Basic Concepts

Introduction

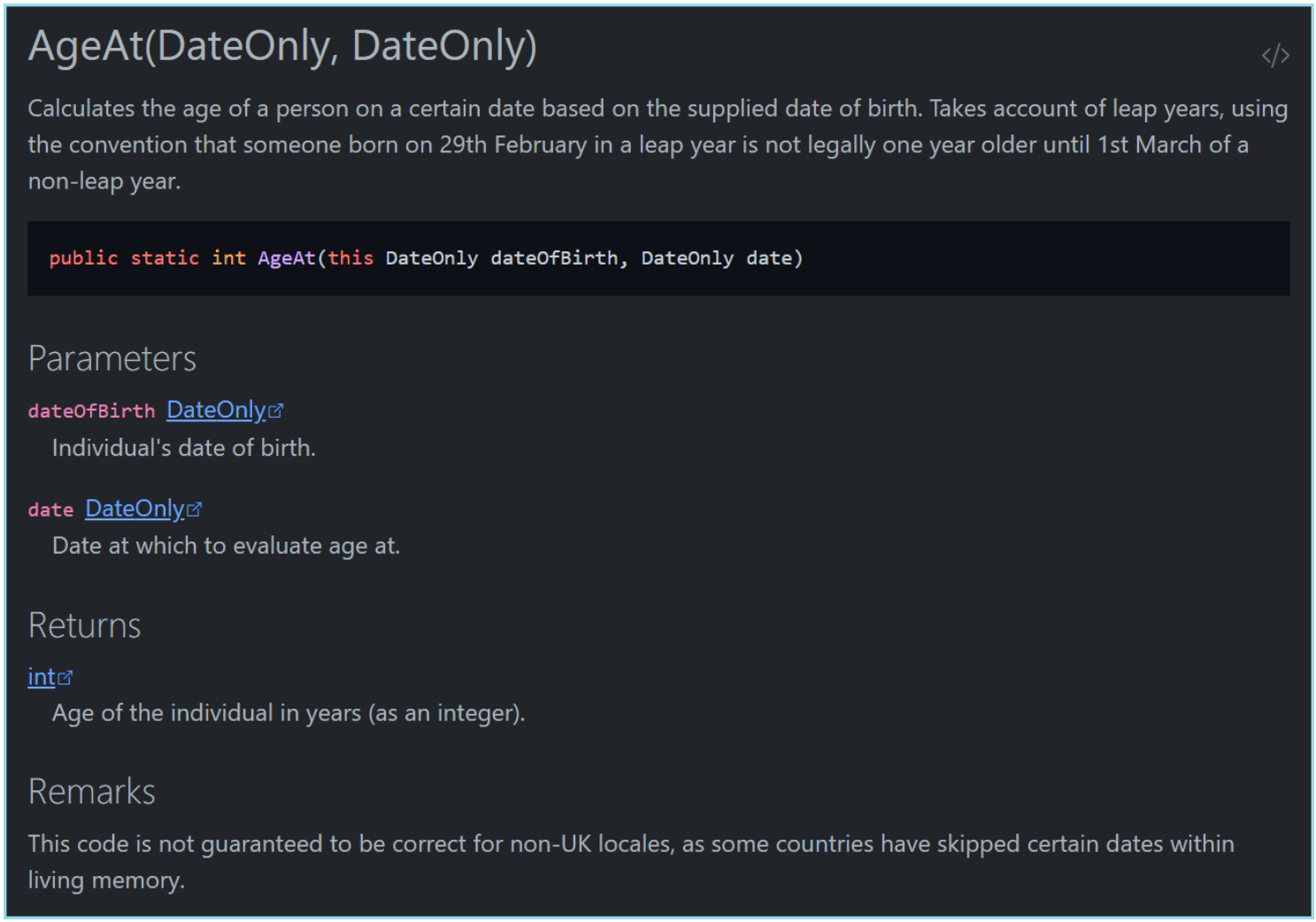
Docfx is a powerful tool but easy to use for most regular use cases, once you understand the basic concepts.

Docfx can be used as a static site generator, but the real value of the tool is in bringing together static documentation pages and .NET API documentation. Docfx supports both C# and VB projects (although currently the output of tool is limited to C# syntax), and relies on the long-established XML comment syntax for C# (and similarly for VB). For example, the following C# code:

```
/// <summary>
/// Calculates the age of a person on a certain date based on the supplied date of
/// using the convention that someone born on 29th February in a leap year is not
/// of a non-leap year.
/// </summary>
/// <param name="dateOfBirth">Individual's date of birth.</param>
/// <param name="date">Date at which to evaluate age at.</param>
/// <returns>Age of the individual in years (as an integer).</returns>
/// <remarks>This code is not guaranteed to be correct for non-UK locales, as some
/// within living memory.</remarks>
public static int AgeAt(this DateOnly dateOfBirth, DateOnly date)
{
    int age = date.Year - dateOfBirth.Year;

    return dateOfBirth > date.AddYears(-age) ? --age : age;
}
```

can be used to generate output like this:



Static documentation pages are prepared using Markdown (slightly enhanced to support specific features). Markdown content can also be injected into the generated API documentation using a feature called 'Overwrites'.

Once the API documentation has been parsed from the source code, it is compiled along with the Markdown content into a set of HTML pages which can be published on a website. It is also possible to compile the final output into one or more PDFs for offline use.

Docfx is a command-line tool that can be invoked directly, or as a .NET Core CLI tool using the `dotnet` command, but it can also be invoked from source code using the `Docset.Build` method in the `Docfx` namespace. It is configured using a JSON configuration file, `docfx.json` which has sections for different parts of the build process.

Consuming .NET projects

The most common use case for processing .NET projects is to specify one or more .csproj files in the `docfx.json` file:

```
{
  "metadata": [
    {
      "src": [
        {
          "files": [
            "src/MyProject.Abc/*.csproj",
            "src/MyProject.Xyz/*.csproj"
          ],
          "src": "path/to/csprojs"
        }
      ],
      "dest": "api"
    }
  ],
  //...
}
```

Although Docfx can build a documentation website in one step, it's helpful to understand the separate steps the tool uses to generate its output.

The first step is called the ***metadata*** step and can be completed using the following command line:

```
docfx metadata path/to/docfx.json
```

This command reads all the source files specified by the projects listed in `docfx.json` and searches for XML documentation entries. Note that this step does not use `.xml` compiler output but rather uses the Roslyn compiler to navigate the supplied codebase. The output of this step is a set of YAML files that are stored in the `dest` folder specified in `docfx.json`.

Here's an example of the (partial) output from the above code example:

```
### YamlMime:ManagedReference
items:
- uid: MyProject.Extensions.DateOnlyExtensions.AgeAt(System.DateOnly,System.DateOnly)
  commentId: M:MyProject.Extensions.DateOnlyExtensions.AgeAt(System.DateOnly,System.Dat
  id: AgeAt(System.DateOnly,System.DateOnly)
  isExtensionMethod: true
  parent: MyProject.Extensions.DateOnlyExtensions
  langs:
  - csharp
  - vb
```

```
  name: AgeAt(DateOnly, DateOnly)
  nameWithType: DateOnlyExtensions.AgeAt(DateOnly, DateOnly)
  fullName: MyProject.Extensions.DateOnlyExtensions.AgeAt(System.DateOnly, System.DateO
  type: Method
  source:
    remote:
      path: src/MyProject/Extensions/DateOnlyExtensions.cs
      branch: main
      repo: https://github.com/MyUser/MyProject.git
    id: AgeAt
    path: ../../MyProject/src/MyProject/Extensions/DateOnlyExtensions.cs
    startLine: 63
  assemblies:
  - MyProject.Common
  namespace: MyProject.Extensions
  summary: >-
    Calculates the age of a person on a certain date based on the supplied date of birt
```

For the most part, it isn't important to know too much about the output of the `metadata` step, except where you want to make reference to entities from your Markdown content. When doing so, you need to reference the relevant `uid` from the YAML file. However, as you can see, the `uid` is the same as the full signature of the entity or method including the namespace.

It's also worth knowing that the `metadata` step generates `toc.yml`, a table-of-contents file for the input source code, grouped by .NET namespace. This is the only auto-generated table-of-contents file; all other `toc.yml` must be manually created/edited.

> **NOTE**
>
> In additional to using `.csproj` files for input, it is also possible to generate the intermediate YAML output from compiled `.dll` (or `.exe`) and `.xml` files. See .NET API Docs for further details.

Documentation Build Process

The next step is called the ***build*** step and can be completed using the following command line:

```
docfx build path/to/docfx.json
```

(You can append `--serve` to this step and Docfx will start a local web server so you can preview the final output.)

Internally, there are many parts to this step, but in short, Docfx does the following during the `build` step:

- resolve all cross-references
- convert the YAML content from the `metadata` step into a structured data format, for passing onto the template engine
- convert all Markdown content into HTML
- apply templates and themes

Conversion of Markdown to HTML is achieved using the Markdig CommonMark-compliant Markdown processor.

Template and theme processing is the one part of Docfx that is not coded in C#; instead the Jint JavaScript interpreter is used to run a set of JavaScript scripts; this approach allows an extra level of customisation of the build process as Docfx provides a way to override the default scripts using the template section of the `docfx.json` file:

```
{
  "build": {
    //...
    "output": "_site",
    "template": [
      "default",
      "modern",
      "templates/mytemplate"
    ]
  }
}
```

In this example, Docfx first searches the `templates\mytemplate` folder, then the `modern` folder, then `default` folder for each `.css` or `.js` file. Note that `default` and `modern` templates are included with Docfx and included in the Docfx installation packaged alongside the Docfx executable.

(The embedded templates can be exported using the command

```
docfx template export default -o path/for/exported_templates
```

where `default` is the name of the template being exported. The command `docfx template list` can be used to list the embedded templates within Docfx.)

Quick Start

Build your technical documentation site with docfx. Converts .NET assembly, XML code comment, REST API Swagger files and markdown into rendered HTML pages, JSON model or PDF files.

Create a New Website

In this section we will build a simple documentation site on your local machine.

> Prerequisites
>
> - Familiarity with the command line
> - Install .NET SDK 6.0 or higher

Make sure you have .NET SDK installed, then open a terminal and enter the following command to install the latest docfx:

```
dotnet tool update -g docfx
```

To create a new docset, run:

```
docfx init --quiet
```

This command creates a new docset under the `docfx_project` directory. To build the docset, run:

```
docfx docfx_project/docfx.json --serve
```

Now you can preview the website on http://localhost:8080.

To preview your local changes, save changes then run this command in a new terminal to rebuild the website:

```
docfx docfx_project/docfx.json
```

Publish to GitHub Pages

Docfx produces static HTML files under the `_site` folder ready for publishing to any static site hosting servers.

To publish to GitHub Pages:

1. Enable GitHub Pages.
2. Upload `_site` folder to GitHub Pages using GitHub actions.

This example uses `peaceiris/actions-gh-pages` to publish to the `gh-pages` branch:

```
# Your GitHub workflow file under .github/workflows/

jobs:
  publish-docs:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout
      uses: actions/checkout@v3
    - name: Dotnet Setup
      uses: actions/setup-dotnet@v3
      with:
        dotnet-version: 7.x

    - run: dotnet tool update -g docfx
    - run: docfx docfx_project/docfx.json

    - name: Deploy
      uses: peaceiris/actions-gh-pages@v3
      with:
        github_token: ${{ secrets.GITHUB_TOKEN }}
        publish_dir: docs/_site
```

Use the NuGet Library

You can also use docfx as a NuGet library:

```
<PackageReference Include="Docfx.App" Version="2.70.0" />
```

Then build a docset using:

```
await Docfx.Docset.Build("docfx.json");
```

See API References for additional APIs.

Next Steps

- Write Articles
- Organize Contents
- Configure Website
- Add .NET API Docs

Markdown

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

Markdown Extensions

Docfx supports additional markdown syntax that provide richer content. These syntax are specific to docfx and won't be rendered elsewhere like GitHub.

To use a custom markdown extension:

1. Use docfx as a NuGet library:

```
<PackageReference Include="Docfx.App" Version="2.70.0" />
```

2. Configure the markdig markdown pipeline:

```
var options = new BuildOptions
{
    // Enable custom markdown extensions here
    ConfigureMarkdig = pipeline => pipeline.UseCitations(),
}

await Docset.Build("docfx.json", options);
```

Here is a list of markdown extensions provided by docfx by default.

YAML header

Also referred to as YAML Front Matter, the YAML header is used to annotate a Markdown file with various metadata elements. It should appear at the top of the document. Here's an example:

```
---
uid: fileA
---

# This is fileA
...
```

In this example, the UID provides a unique identifier for the file and is intended to be unique inside a project. If you define duplicate UID for two files, the resolve result is undetermined.

For API reference files, the UID is auto generated by mangling the API's signature. For example, the System.String class's UID is `System.String`. You can open a generated YAML file to lookup the value of its UID.

> **NOTE**
> Conceptual Markdown file doesn't have UID generated by default. So it cannot be cross referenced unless you give it a UID.

See the list of predefined metadata for applicable options for inclusion in the YAML header.

Alerts

Alerts are block quotes that render with colors and icons that indicate the significance of the content.

The following alert types are supported:

```
> [!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.
```

They look like this in rendered page:

> **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.

Custom Alerts

You can define custom alerts with the `build.markdownEngineProperties.alerts` property in `docfx.json` and use it in markdown files. The key specifies the markdown keyword without the surrounding `[!`, `]` symbols. The value is the CSS class names:

```
{
  "build": {
    "markdownEngineProperties": {
      "alerts": {
        "TODO": "alert alert-secondary"
      }
    }
  }
}
```

```
> [!TODO]
> This is a custom TODO section
```

The above custom alert looks like this in rendered page:

> **TODO**
> This is a custom TODO section

DocFX allows you to customize the display of alert titles in your documentation. By default, alert titles are displayed as the keyword in upper case. To change this behavior, you can create a custom template and use a `token.json` file to define your custom alert titles:

1. **Create a custom template**: Follow the steps in the [Custom Template Guide](create a custom template) to create your own template.
2. **Create a `token.json` file**: In your custom template folder, create a new file named `token.json`. This file will be used to define your custom alert titles. The format should be as follows:

```
{
  "todo": "MY TODO"
}
```

In this example, the key is the alert keyword in **lower case** (e.g., "todo"), and the value is the custom display title of the alert (e.g., "MY TODO").

Video

You can embed a video in your page by using the following Markdown syntax:

```
> [!Video embed_link]
```

Example:

```
> [!Video https://www.youtube.com/embed/Sz1lCeedcPI]
```

This will be rendered as:

Image

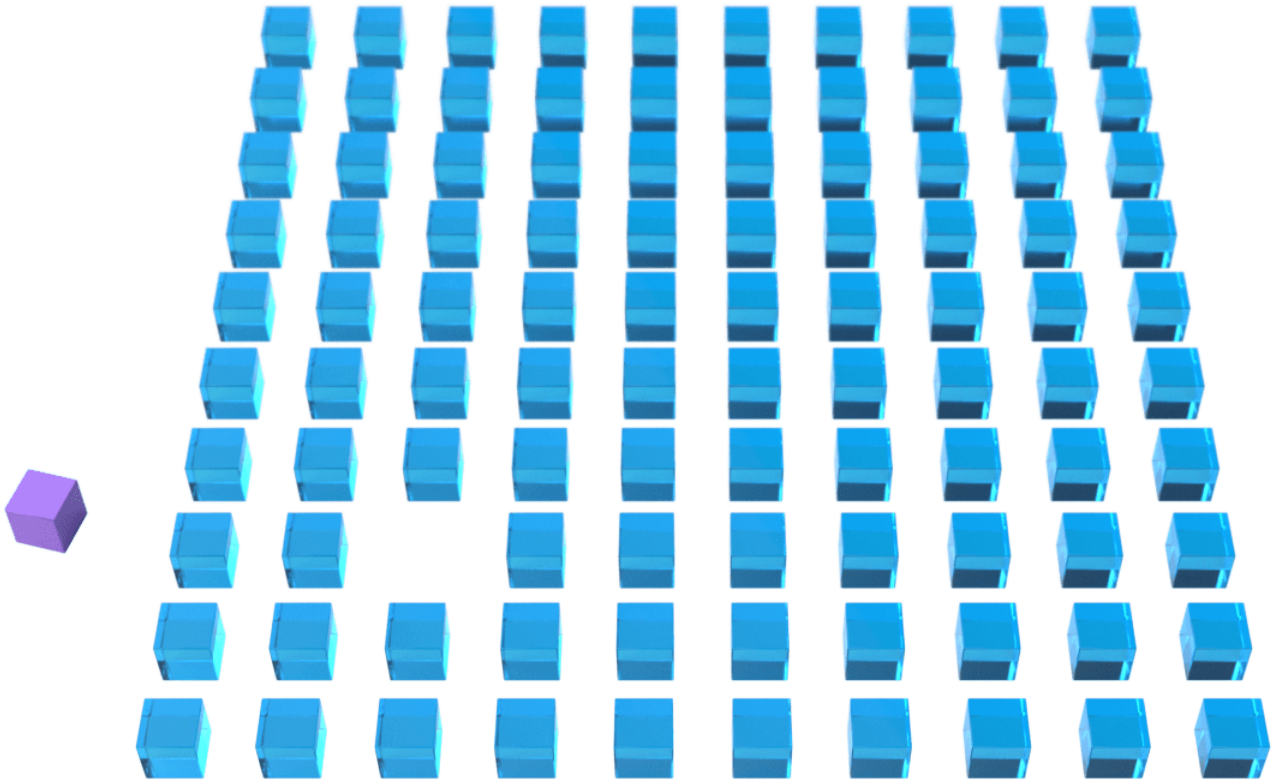
You can embed a image in your page by using the following Markdown syntax:

```
![ <alt-text> ]( <image-link> )
```

Example:

```
![alt-text](https://learn.microsoft.com/en-us/media/learn/not-found/learn-not-found-lig
```

This will be rendered as:



Math Expressions

Docfx supports LaTeX formatted math expressions within markdown using MathJax.

> **NOTE**
> Math expressions is only supported in the `modern` template.

To include a math expression inline with your text, delimit the expression with a dollar symbol $.

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

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

To add a math expression as a block, start a new line and delimit the expression with two dollar symbols $$.

```
**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( \
```

The Cauchy-Schwarz Inequality

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

Mermaid Diagrams

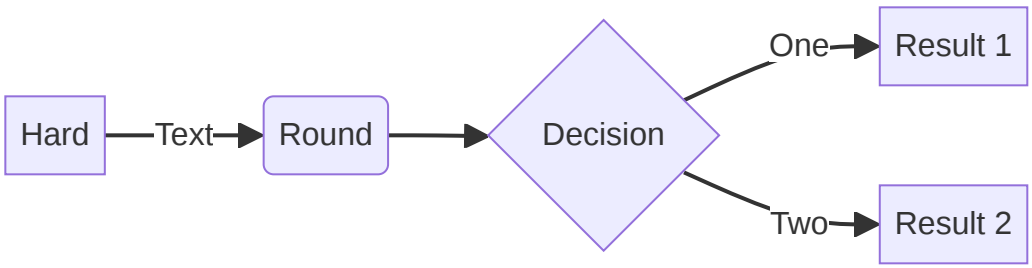
You can embed mermaid diagrams using markdown code block:

Example:

````
```mermaid
flowchart LR

A[Hard] -->|Text| B(Round)
B --> C{Decision}
C -->|One| D[Result 1]
C -->|Two| E[Result 2]
```
````

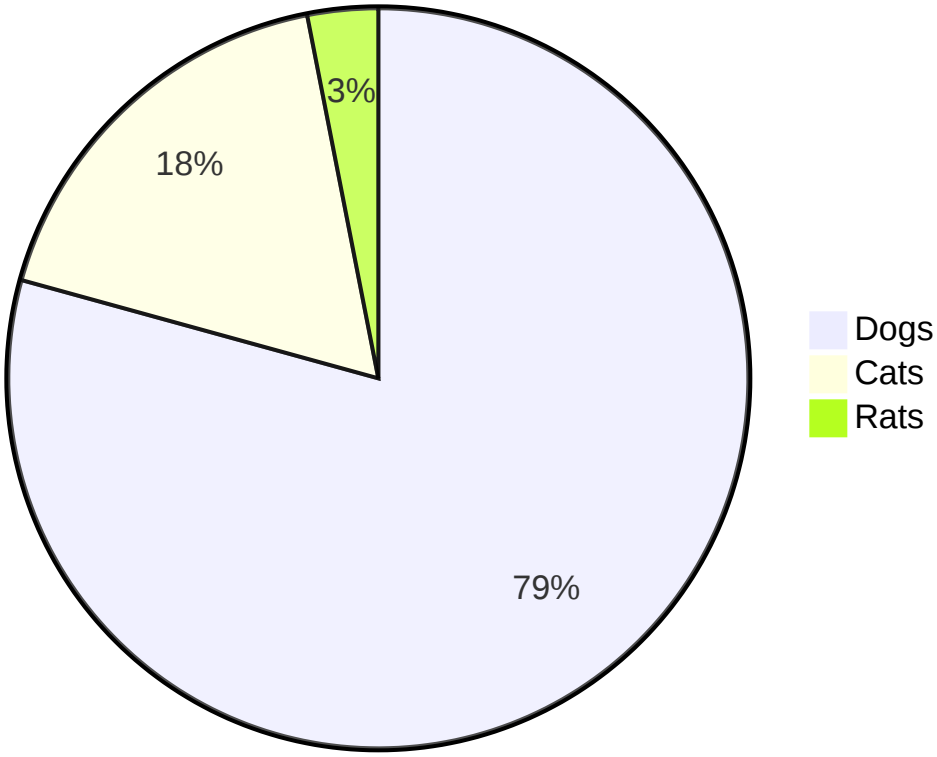
This will be rendered as:



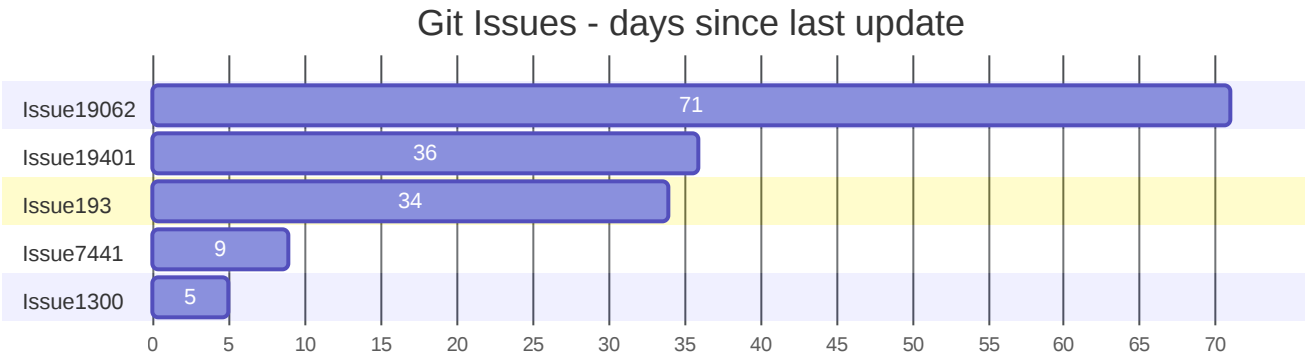
> **NOTE**
> Mermaid diagrams is only supported in the `modern` template.

There are plenty of other diagrams supported by mermaid such as:

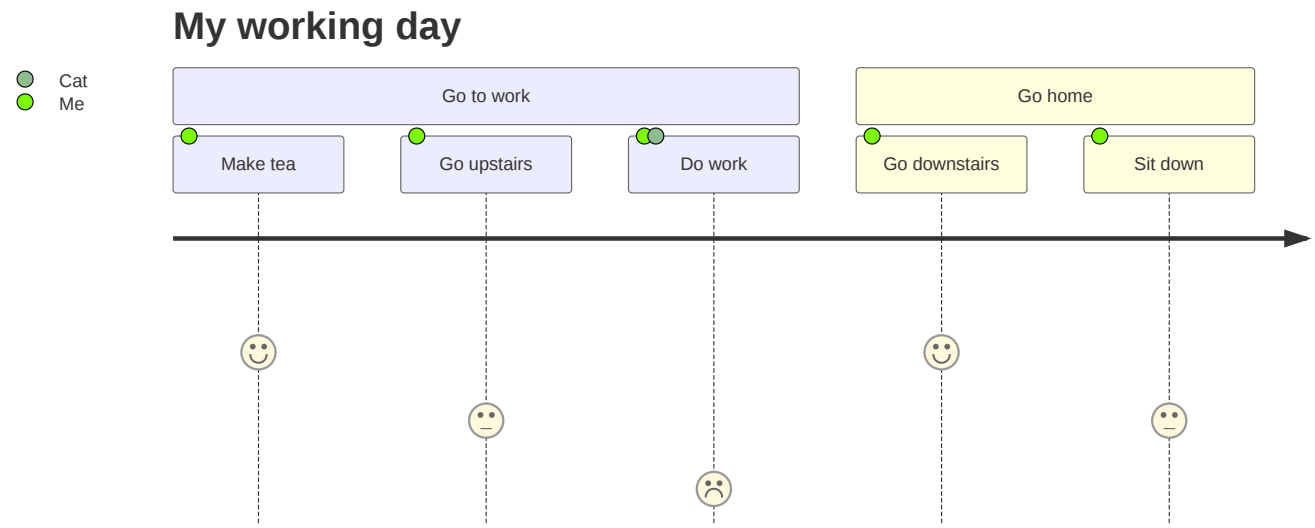
Pie chart



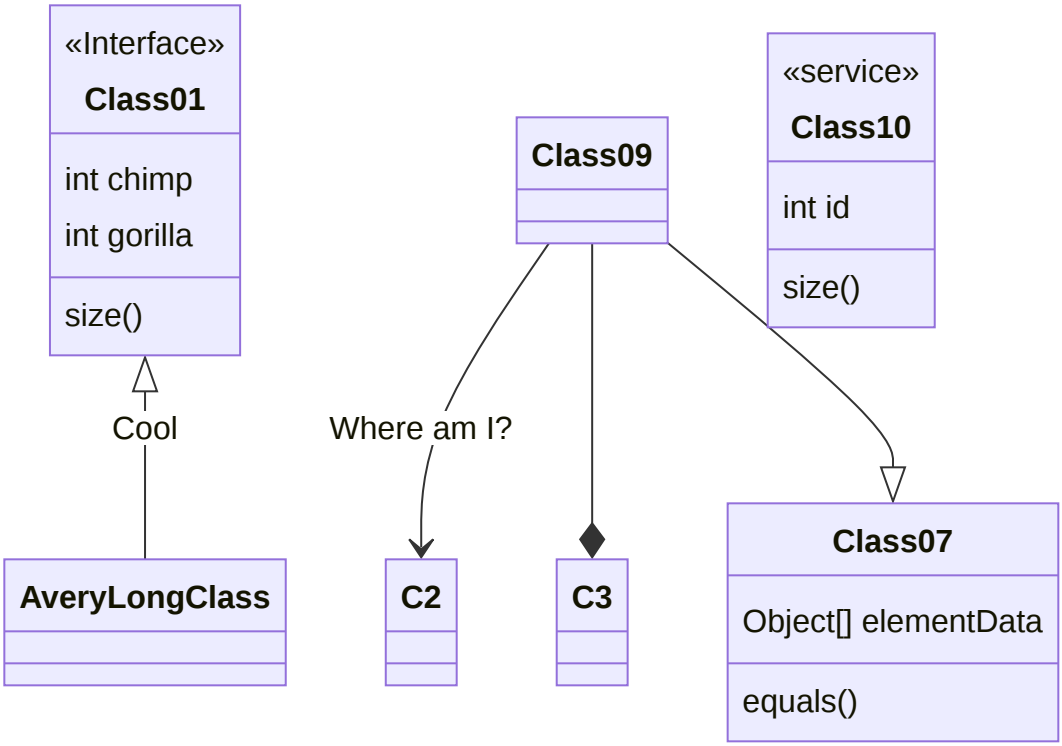
Bar chart



User Journey diagram



Class diagram



Include Markdown Files

Where markdown files need to be repeated in multiple articles, you can use an include file. The includes feature replace the reference with the contents of the included file at build time.

You can reuse a common text snippet within a sentence using inline include:

```
Text before [!INCLUDE [<title>](<filepath>)] and after.
```

Or reuse an entire Markdown file as a block, nested within a section of an article. Block include is on its own line:

```
[!INCLUDE [<title>](<filepath>)]
```

Where `<title>` is the name of the file and `<filepath>` is the relative path to the file.

Example:

```
[!INCLUDE [my-markdown-block](../../includes/my-markdown-block.md)]
```

Included markdown files needs to be excluded from build, they are usually placed in the `/includes` folder.

Code Snippet

There are several ways to include code in an article. The code snippet syntax replaces code from another file:

```
[!code-csharp[](Program.cs)]
```

You can include selected lines from the code snippet using region or line range syntax:

```
[!code-csharp[](Program.cs#region)]
[!code-csharp[](Program.cs#L12-L16)]
```

Code snippets are indicated by using a specific link syntax described as follows:

```
[!code-<language>[](<filepath><query-options>)]
```

Where `<language>` is the syntax highlighting language of the code and `<filepath>` is the relative path to the markdown file.

Highlight Selected Lines

Code Snippets typically include more code than necessary in order to provide context. It helps readability when you highlight the key lines that you're focusing on. To highlight key lines, use the `highlight` query options:

```
[!code-csharp[](Program.cs?highlight=2,5-7,9-)]
```

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);
        }
    }
}
```

Tabs

Tabs enable content that is multi-faceted. They allow sections of a document to contain variant content renderings and eliminates duplicate content.

Here's an example of the tab experience:

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...

---
```

Differences introduced by DFM syntax

> [!WARNING]
> Please note that DFM introduces more syntax to support more functionalities. When GFM does not support them, preview the Markdown file inside *GFM Preview* can lead to different results.

Text after block extension

Some block extension in DFM cannot be recognized in GFM. In GFM, it would be treated as a part of paragraph. Then, following content would be treated as a part of paragraph.

For example:

```
> [!NOTE]
>     This is code.
```

In GFM, it will be rendered as a paragraph with content `[!NOTE] This is code.` in blockquote.
In DFM, it will be rendered as a code in note.

Table of Contents

A table of contents (TOC) defines the structure of a set of documents.

YAML TOC

To add a TOC, create a file named `toc.yml`. Here's the structure for a simple YAML TOC:

```yaml
items:
- name: Tutorial
  items:
  - name: Introduction
    href: tutorial.md
  - name: Step 1
    href: step-1.md
  - name: Step 2
    href: step-2.md
  - name: Step 3
    href: step-3.md
```

The YAML document is a tree of TOC nodes, each of which has these properties:

- `name`: The display name for the TOC node.
- `href`: The path the TOC node leads to. Optional because a node can exist just to parent other nodes.
- `items`: If a node has children, they're listed in the items array.
- `uid`: The uid of the article. Can be used instead of `href`.
- `expanded`: Expand children on load, only works if the template is `modern`.

Nested TOCs

To nest a TOC within another TOC, set the `href` property to point to the `toc.yml` file that you want to nest. You can also use this structure as a way to reuse a TOC structure in one or more TOC files.

Consider the following two `toc.yml` files:

toc.yml:

```yaml
items:
- name: Overview
  href: overview.md
```

```yaml
- name: Reference
  href: api/toc.yml
```

api/toc.yml:

```yaml
items:
- name: System.String
  href: system.string.yml
- name: System.Float
  href: system.float.yml
```

This structure renders as follows:

```
Overview
Reference
├─ System.String
├─ System.Float
```

Reference TOCs

To reference another TOC without embeding it to a parent TOC using nested TOCs, set the `href` property to point to the directory that you want to reference and end the string with `/`, this will generate a link pointing to the first article in the referenced TOC.

Consider the following folder structure:

```
toc.yml
├─ System
    ├─ toc.yml
├─ System.Collections
    ├─ toc.yml
```

toc.yml:

```yaml
- name: System
  href: System/
- name: System.Collections
  href: System.Collections/
```

This structure renders as follows:

```
System # Link to the first article in System/toc.yml
System.Collections  # Link to the first article in System.Collections/toc.yml
```

Navigation Bar

The `toc.yml` file in the `docfx.json` folder will be used to fill the content of the navigation bar at the top of the page. It usually uses Reference TOCs to navigate to child pages.

The following example creates a navigation bar with two *Docs* and *API* entries:

```
toc.yml
├─ docs
    ├─ toc.yml
├─ api
    ├─ toc.yml
```

toc.yml:

```
- name: Docs
  href: docs/
- name: API
  href: api/
```

Config

Docfx uses `docfx.json` as the config file for the site. Most docfx commands operate in a directory containing `docfx.json`.

The `build` config determines what files are included in the site:

```
{
  "build": {
    "content": [
      { "files": "**/*.{md,yml}", "exclude": "**/include/**" }
    ],
    "resource": [
      { "files": "**/images/**" }
    ]
  }
}
```

The `content` config defines glob patterns of files that are transformed to HTML by the build process. It is usually the markdown files and auto-generated API YAML files.

The `resource` config defines static resources copied to output as is.

URL Management

URL is determined by the file path relative to `docfx.json`. Docfx uses “Ugly URLs”: a file named `docs/urls.md` is accessible from the `docs/urls.html` URL.

To customize URL pattern for a directory, use the `src` property to remove the directory name from the URL, and use the `dest` property to insert an URL prefix:

```
{
  "build": {
    "content": [
      { "files": "**/*.{md,yml}", "src": "articles", "dest": "docs" }
    ]
  }
}
```

In this example, files in the `articles` directory uses `docs` as the base URL: The `articles/getting-started/installation.md` file is accessible by the `docs/getting-started/installation.html` URL.

URL Redirects

The `redirect_url` metadata is a simple way to create redirects in your documentation. This metadata can be added to a Markdown file in your project, and it will be used to redirect users to a new URL when they try to access the original URL:

```
---
redirect_url: [new URL]
---
```

Replace [new URL] with the URL that you want to redirect users to. You can use any valid URL, including relative URLs or external URLs.

Metadata

Metadata are attributes attached to an file. It helps shape the look and feel of a page and provides extra context to the article.

To add metadata to an article, use "YAML Front Matter" markdown extension syntax:

```
---
title: a title
description: a description
---
```

Some metadata attributes are consistent across a set of content. Use the `globalMetadata` property in `docfx.json` to apply the same metadata to all articles:

```
{
  "build": {
    "globalMetadata": {
      "_appTitle": "My App"
    }
  }
}
```

To apply identical metadata values to a folder or a set of content, use the `fileMetadata` config:

```
{
  "build": {
    "fileMetadata": {
```

```
      "_appTitle": {
        "articles/dotnet/**/*.md": ".NET",
        "articles/typescript/**/*.md": "TypeScript"
      }
    }
  }
}
```

When the same metadata key is defined in multiple places, YAML Front Matter takes precedence over `fileMetadata` which in turn takes precedence over `globalMetadata`.

Sitemap

Docfx produces a sitemap.xml about the pages on your site for search engines like Google to crawl your site more efficiently.

The `sitemap` option in `docfx.json` controls how sitemaps are generated:

```
{
  "build": {
    "sitemap": {
      "baseUrl": "https://dotnet.github.io/docfx",
      "priority": 0.1,
      "changefreq": "monthly"
    }
  }
}
```

Where:

- `baseUrl` is the base URL for the website. It should start with `http` or `https`. For example, `https://dotnet.github.io/docfx`.
- `lastmod` is the date of last modification of the page. If not specified, docfx sets the date to the build time.
- `changefreq` determines how frequently the page is likely to change. Valid values are `always`, `hourly`, `daily`, `weekly`, `monthly`, `yearly`, `never`. Default to `daily`.
- `priority` is the priority of this URL relative to other URLs on your site. Valid values range from 0.0 to 1.0. Default to `0.5`
- `fileOptions` is a per file config of the above options. The key is the file glob pattern and value is the sitemap options.

Template

Template defines the appearance of the website.

Docfx ships several built-in templates. We recommend using the modern template that matches the look and feel of this site. It supports dark mode, more features, rich customization options and.

Use the modern template by setting the `template` property to `["default", "modern"]`:

```
{
  "build": {
    "template": [
      "default",
      "modern"
    ]
  }
}
```

Additional templates are available at the Template Gallery.

Template Metadata

The easiest way of customizing the the appearance of pages is using metadata. Here is a list of predefined metadata:

Modern Template | Default Template

Name	Type	Description
`_appTitle`	string	A string append to every page title.
`_appName`	string	The name of the site displayed after logo.
`_appFooter`	string	The footer HTML.
`_appLogoPath`	string	Path to the app logo.
`_appLogoUrl`	string	URL for the app logo.
`_appFaviconPath`	string	Favicon URL path.
`_enableSearch`	bool	Whether to show the search box.

Name	Type	Description
`_noindex`	bool	Whether to include in search results
`_disableContribution`	bool	Whether to show the "*Edit this page*" button.
`_gitContribute`	object	Defines the `repo` and `branch` property of git links.
`_gitUrlPattern`	string	URL pattern of git links.
`_disableNewTab`	bool	Whether to render external link indicator icons and open external links in a new tab.
`_disableNextArticle`	bool	Whether to show the previous and next article link.
`_disableTocFilter`	bool	Whether to show the table of content filter box.
`_googleAnalyticsTagId`	string	Enables Google Analytics web traffic analysis.
`_lang`	string	Primary language of the page. If unset, the `<html>` tag will not have `lang` property.
`_layout`	string	Determines the layout of the page. Supported values are `landing` and `chromeless`.

TIP

Docfx produces the right git links for major CI pipelines including GitHub, GitLab, Azure Pipelines, AppVeyor, TeamCity, Jenkins. `_gitContribute` and `_gitUrlPattern` are optional on these platforms.

Custom Template

To build your own template, create a new folder and add it to `template` config in `docfx.json`:

Modern Template | Default Template

```
{
  "build": {
    "template": [
```

```
      "default",
      "modern",
      "my-template" // <-- Path to custom template
    ]
  }
}
```

Add your custom CSS file to `my-template/public/main.css` to customize colors, show and hide elements, etc. This is an example stylesheet that adjust the font size of article headers.

```
/* file: my-template/public/main.css */
article h1 {
  font-size: 40px;
}
```

You can also use CSS variables to adjust the templates. There are many predefined CSS variables in Bootstrap that can be used to customize the site:

```
/* file: my-template/public/main.css */
body {
  --bs-link-color-rgb: 66, 184, 131 !important;
  --bs-link-hover-color-rgb: 64, 180, 128 !important;
}
```

The `my-template/public/main.js` file is the entry JavaScript file to customize docfx site behaviors. This is a basic setup that changes the default color mode to dark and adds some icon links in the header:

```
/* file: my-template/public/main.js */
export default {
  defaultTheme: 'dark',
  iconLinks: [
    {
      icon: 'github',
      href: 'https://github.com/dotnet/docfx',
      title: 'GitHub'
    },
    {
      icon: 'twitter',
      href: 'https://twitter.com',
      title: 'Twitter'
```

```
      }
    ]
  }
```

You can also configure syntax highlighting options using the `configureHljs` option:

```
export default {
  configureHljs: (hljs) => {
    // Customize hightlight.js here
  },
}
```

See this example on how to enable `bicep` syntax highlighting.

More customization options are available in the docfx options object.

.NET API Docs

Docfx converts XML documentation comments into rendered HTML documentations.

Generate .NET API Docs

To add API docs for a .NET project, add a `metadata` section before the `build` section in `docfx.json` config:

```
{
  "metadata": {
    "src": [{
      "files": ["**/bin/Release/**.dll"],
      "src": "../"
    }],
    "dest": "api"
  },
  "build": {
    "content": [{
      "files": [ "api/*.yml" ]
    }]
  }
}
```

Docfx generates .NET API docs in 2 stages:

1. The *metadata* stage uses the `metadata` config to produce .NET API YAML files at the `metadata.dest` directory.

> **NOTE**
> The `Docset.Build` method does not run the *metadata* stage, invoke the `DotnetApiCatalog.GenerateManagedReferenceYamlFiles` method to run the *metadata* stage before the *build* stage.

2. The *build* stage transforms the generated .NET API YAML files specified in `build.content` config into HTML files.

These 2 stages can run independently with the `docfx metadata` command and the `docfx build` command. The `docfx` root command runs both `metadata` and `build`.

> **NOTE**
>
> Glob patterns in docfx currently does not support crawling files outside the directory containing `docfx.json`. Use the `metadata.src.src` property

Docfx supports several source formats to generate .NET API docs:

Generate from assemblies

When the file extension is `.dll` or `.exe`, docfx produces API docs by reflecting the assembly and the side-by-side XML documentation file.

This approach is build independent and language independent, if you are having trouble with msbuild or using an unsupported project format such as `.fsproj`, generating docs from assemblies is the recommended approach.

Docfx examines the assembly and tries to load the reference assemblies from within the same directory or the global systems assembly directory. In case an reference assembly fails to resolve, use the `references` property to specify a list of additional reference assembly path:

```
{
  "metadata": {
    "src": [{
      "files": ["**/bin/Release/**.dll"],
      "src": "../"
    }],
    "dest": "api",
    "references": [
      "path-to-reference-assembly.dll"
    ]
  },
}
```

If source link is enabled on the assembly and the `.pdb` file exists along side the assembly, docfx shows the "View Source" link based on the source URL extract from source link.

Generate from projects or solutions

When the file extension is `.csproj`, `.vbproj` or `.sln`, docfx uses `MSBuildWorkspace` to perform a design-time build of the projects before generating API docs.

In order to successfully load an MSBuild project, .NET Core SDK must be installed and available globally. The installation must have the necessary workloads and components to support the projects you'll be loading.

Run `dotnet restore` before `docfx` to ensure that dependencies are available. Running `dotnet restore` is still needed even if your project does not have NuGet dependencies when Visual Studio is not installed.

To troubleshoot MSBuild load problems, run `docfx metadata --logLevel verbose` to see MSBuild logs.

Docfx build the project using `Release` config by default, additional MSBuild properties can be specified with `properties`.

If your project targets multiple target frameworks, docfx internally builds each target framework of the project. Try specify the `TargetFramework` MSBuild property to speed up project build:

```
{
  "metadata": {
    "src": [{
      "files": ["**/bin/Release/**.dll"],
      "src": "../"
    }],
    "dest": "api",
    "properties": {
      "TargetFramework": "net6.0"
    }
  },
}
```

Generate from source code

When the file extension is `.cs` or `.vb`, docfx uses the latest supported .NET Core SDK installed on the machine to build the source code using `Microsoft.NET.Sdk`. Additional references can be specified in the `references` config:

```
{
  "metadata": {
    "src": [{
      "files": ["**/bin/Release/**.dll"],
      "src": "../"
    }],
    "dest": "api",
```

```
    "references": [
      "path-to-reference-assembly.dll"
    ]
  },
}
```

Customization Options

There are several options available for customizing .NET API pages that are tailored to your specific needs and preferences. To customize .NET API pages for DocFX, you can use the following options:

- `memberLayout`: This option determines whether type members should be on the same page as containing type or as dedicated pages. Possible values are:
  - `samePage`: Type members are on the same page as containing type.
  - `separatePages`: Type members are on dedicated pages.
- `namespaceLayout`: This option determines whether namespace node in TOC is a list or nested. Possible values are:
  - `flattened`: Namespace node in TOC is a list.
  - `nested`: Namespace node in TOC is nested.

Supported XML Tags

Docfx supports Recommended XML tags for C# documentation comments.

> **WARNING**
> Docfx parses XML documentation comment as markdown by default, writing XML documentation comments using markdown may cause rendering problems on places that do not support markdown, like in the Visual Studio intellisense window.

To disable markdown parsing while processing XML tags, set `shouldSkipMarkup` to `true`:

```
{
  "metadata": {
    "src": [{
      "files": ["**/bin/Release/**.dll"],
      "src": "../"
    }],
    "dest": "api",
```

```
    "shouldSkipMarkup": true
  }
}
```

Filter APIs

Docfx shows only the public accessible types and methods callable from another assembly. It also has a set of default filtering rules that excludes common API patterns based on attributes such as `[EditorBrowsableAttribute]`.

To disable the default filtering rules, set the `disableDefaultFilter` property to `true`.

To show private methods, set the `includePrivateMembers` config to `true`. When enabled, internal only langauge keywords such as `private` or `internal` starts to appear in the declaration of all APIs, to accurately reflect API accessibility.

There are two ways of customizing the API filters:

Custom with Code

To use a custom filtering with code:

1. Use docfx .NET API generation as a NuGet library:

```
<PackageReference Include="Docfx.Dotnet" Version="2.62.0" />
```

2. Configure the filter options:

```
var options = new DotnetApiOptions
{
    // Filter based on types
    IncludeApi = symbol => ...

    // Filter based on attributes
    IncludeAttribute = symbol => ...
}

await DotnetApiCatalog.GenerateManagedReferenceYamlFiles("docfx.json", options);
```

The filter callbacks takes an `ISymbol` interface and produces an `SymbolIncludeState` enum to choose between include the API, exclude the API or use the default filtering behavior.

The callbacks are raised before applying the default rules but after processing type accessibility rules. Private types and members cannot be marked as include unless

`includePrivateMembers` is true.

Hiding the parent symbol also hides all of its child symbols, e.g.:

- If a namespace is hidden, all child namespaces and types underneath it are hidden.
- If a class is hidden, all nested types underneath it are hidden.
- If an interface is hidden, explicit implementations of that interface are also hidden.

Custom with Filter Rules

To add additional filter rules, add a custom YAML file and set the `filter` property in `docfx.json` to point to the custom YAML filter:

```
{
  "metadata": {
    "src": [{
      "files": ["**/bin/Release/**.dll"],
      "src": "../"
    }],
    "dest": "api",
    "filter": "filterConfig.yml" // <-- Path to custom filter config
  }
}
```

The filter config is a list of rules. A rule can include or exclude a set of APIs based on a pattern. The rules are processed sequentially and would stop when a rule matches.

Filter by UID

Every item in the generated API docs has a `UID` (a unique identifier calculated for each API) to filter against using regular expression. This example uses `uidRegex` to excludes all APIs whose uids start with `Microsoft.DevDiv` but not `Microsoft.DevDiv.SpecialCase`.

```
apiRules:
- include:
    uidRegex: ^Microsoft\.DevDiv\.SpecialCase
- exclude:
    uidRegex: ^Microsoft\.DevDiv
```

Filter by Type

This example exclude APIs whose uid starts with `Microsoft.DevDiv` and type is `Type`:

```
apiRules:
- exclude:
    uidRegex: ^Microsoft\.DevDiv
    type: Type
```

Supported value for `type` are:

- `Namespace`
- `Class`
- `Struct`
- `Enum`
- `Interface`
- `Delegate`
- `Event`
- `Field`
- `Method`
- `Property`
- `Type`: a `Class`, `Struct`, `Enum`, `Interface` or `Delegate`.
- `Member`: a `Field`, `Event`, `Method` or `Property`.

API filter are hierarchical, if a namespace is excluded, all types/members defined in the namespace would also be excluded. Similarly, if a type is excluded, all members defined in the type would also be excluded.

Filter by Attribute

This example excludes all APIs which have `AttributeUsageAttribute` set to `System.AttributeTargets.Class` and the `Inherited` argument set to `true`:

```
apiRules:
- exclude:
  hasAttribute:
    uid: System.AttributeUsageAttribute
    ctorArguments:
    - System.AttributeTargets.Class
```

```
    ctorNamedArguments:
      Inherited: "true"
```

Where the `ctorArguments` property specifies a list of match conditions based on constructor parameters and the `ctorNamedArguments` property specifies match conditions using named constructor arguments.

Add REST API docs

Docfx generates REST API documentation from Swagger 2.0 files.

To add REST API docs, include the swagger JSON file to the `build` config in `docfx.json`:

```
{
  "metadata": {
    "src": [ "../src/**/bin/Release/**.dll" ],
    "dest": "api"
  },
  "build": {
    "content": [{
      "files": [ "**/*.swagger.json" ] // <-- Include swagger JSON files
    }]
  }
}
```

Each swagger file produces one output HTML file.

Organize REST APIs using Tags

APIs can be organized using the Tag Object. An API can be associated with one or more tags. Untagged APIs are put in the *Other apis* section.

This example defines the `Basic` and `Advanced` tags and organize APIs using the two tags. The `x-bookmark-id` property specifies the URL fragment for the tag.

```
{
  "swagger": "2.0",
  "info": {
    "title": "Contacts",
    "version": "1.6"
  },
  "host": "microsoft.com",
  "basePath": "/docfx",
  "schemes": [
    "https"
  ],
  "tags": [
    {
      "name": "Basic",
      "x-bookmark-id": "BasicBookmark",
      "description": "Basic description"
```

```
    },
    {
      "name": "Advanced",
      "description": "Advanced description"
    }
  ],
  "paths": {
    "/contacts": {
      "get": {
        "operationId": "get_contacts",
        "tags": [
          "Basic",
          "Advanced"
        ]
      },
      "set": {
        "operationId": "set_contacts",
        "tags": [
          "Advanced"
        ]
      },
      "delete": {
        "operationId": "delete_contacts"
      }
    }
  }
}
```

The above example produces the following layout:

```
Basic
├─ get_contacts
Advanced
├─ get_contacts
├─ set_contacts
Other APIs
├─ delete_contacts
```

Links and Cross References

Markdown provides a syntax to create hyperlinks. For example, the following syntax:

```
[bing](http://www.bing.com)
```

Will render to:

```
<a href="http://www.bing.com">bing</a>
```

Here the url in the link could be either absolute url pointing to another website (`www.bing.com` in the above example), or a relative url pointing to a local resource on the same server (for example, `about.html`).

When working with large documentation project that contains multiple files, it is often needed to link to another Markdown file using the relative path in the source directory. Markdown spec doesn't have a clear definition of how this should be supported. What's more, there is also a common need to link to another file using a "semantic" name instead of its file path. This is especially common in API reference docs, for example, you may want to use `System.String` to link to the topic of `String` class, without knowing it's actually located in `api/system/string.html`, which is auto generated.

In this document, you'll learn the functionalities DocFX provides for resolving file links and cross reference, which will help you to reference other files in an efficient way.

Link to a file using relative path

In DocFX, you can link to a file using its relative path in the source directory. For example,

You have a `file1.md` under root and a `file2.md` under `subfolder/`:

```
/
|- subfolder/
|  \- file2.md
\- file1.md
```

You can use relative path to reference `file2.md` in `file1.md`:

```
[file2](subfolder/file2.md)
```

DocFX converts it to a relative path in output folder structure:

```
<a href="subfolder/file2.html">file2</a>
```

You can see the source file name (`.md`) is replaced with output file name (`.html`).

> **NOTE**
>
> DocFX does not simply replace the file extension here (`.md` to `.html`), it also tracks the mapping between input and output files to make sure source file path will resolve to correct output path. For example, if in the above case, `subfolder` is renamed to `subfolder2` using file mapping in `docfx.json`, in output html, the link url will also resolve to `subfolder2/file2.html`.

Relative path vs. absolute path

It's recommended to always use relative path to reference another file in the same project. Relative path will be resolved during build and produce build warning if the target file does not exist.

> **TIP**
>
> A file must be included in `docfx.json` to be processed by DocFX, so if you see a build warning about a broken link but the file actually exists in your file system, go and check whether this file is included in `docfx.json`.

You can also use absolute path (path starts with `/`) to link to another file, but DocFX won't check its correctness for you and will keep it as-is in the output HTML. That means you should use the output file path as absolute path. For example, in the above case, you can also write the link as follows:

```
[file2](/subfolder/file2.html)
```

Sometimes you may find it's complicated to calculate relative path between two files. DocFX also supports paths that start with `~` to represent a path relative to the root directory of your project (i.e., where `docfx.json` is located). This kind of path will also be validated and resolved during build. For example, in the above case, you can write the following links in `file2.md`:

```
[file1](~/file1.md)

[file1](../file1.md)
```

Both will resolve to `../file1.html` in output html.

> **WARNING**
> Automatic link doesn't support relative path. If you write something like `<file.md>`, it will be treated as an HTML tag rather than a link.

Links in file includes

If you use file include to include another file, the links in the included file are relative to the included file. For example, if `file1.md` includes `file2.md`:

```
[!include[file2](subfolder/file2.md)]
```

All links in `file2.md` are relative to the `file2.md` itself, even when it's included by `file1.md`.

> **NOTE**
> Please note that the file path in include syntax is handled differently than Markdown link. You can only use relative path to specify location of the included file. And DocFX doesn't require included file to be included in `docfx.json`.

> **TIP**
> Each file in `docfx.json` will build into an output file. But included files usually don't need to build into individual topics. So it's not recommended to include them in `docfx.json`.

Links in inline HTML

Markdown supports inline HTML. DocFX also supports to use relative path in inline HTML. Path in HTML link (`<a>`), image (`<img>`), script (`<script>`) and css (`<link>`) will also be resolved if they're relative path.

Using cross reference

Besides using file path to link to another file, DocFX also allows you to give a file a unique identifier so that you can reference this file using that identifier instead of its file path. This is useful in the following cases:

1. A path to a file is long and difficult to memorize or changes frequently.
2. API reference documentation which is usually auto generated so it's difficult to find its file path.
3. References to files in another project without needing to know the project's file structure.

The basic syntax for cross referencing a file is:

```
<xref:id_of_another_file>
```

This is similar to automatic link syntax in Markdown but with a `xref` scheme. This link will build into:

```
<a href="path_of_another_file">title_of_another_file</a>
```

As you can see, one benefit of using cross reference is that you don't need to specify the link text and DocFX will automatically resolve it for you.

> **NOTE**
>
> Title is extracted from the first heading of the Markdown file. Or you can also specify title using title metadata.

Define UID

The unique identifier of a file in DocFX is called a UID. For a Markdown file, you can specify its UID by adding a UID metadata in the YAML header. For example, the following Markdown defines a UID "fileA".

```
---
uid: fileA
---
```

```
# This is fileA
...
```

> **NOTE**
> UID is supposed to be unique inside a project. If you define duplicate UID for two files, the resolve result is undetermined.

For API reference files, UID is auto generated by mangling the API's signature. For example, the System.String class's UID is `System.String`. You can open a generated YAML file to lookup the value of its UID.

> **NOTE**
> Conceptual Markdown file doesn't have UID generated by default. So it cannot be cross referenced unless you give it a UID.

Different syntax of cross reference

Besides the auto link, we also support some other ways to use cross references:

Markdown link

In Markdown link, you can also use `xref` in link url:

```
[link_text](xref:uid_of_another_file)
```

This will resolve to:

```
<a href="path_of_another_file">link_text</a>
```

In this case, DocFX won't resolve the link text for you because you already specified it, unless the `link_text` is empty.

Shorthand form

You can also use `@uid_to_another_file` to quickly reference another file. There are some rules for DocFX to determine whether a string following `@` are UID:

1. The string after `@` must start with `[A-Za-z]`, and end with:

    - Whitespace or line end
    - Punctuation (`[.,;:!?`~]`) followed by whitespace or line end
    - Two or more punctuations (`[.,;:!?`~]`)
2. A string enclosed by a pair of quotes (`'` or `"`)

The render result of `@` form is same as the auto link form. For example, `@System.String` is the same as `<xref:System.String>`.

> **WARNING**
>
> Since `@` is a common character in a document, DocFX doesn't show a warning if a UID isn't found for a shorthand form xref link. Warnings for missing links are shown for auto links and Markdown links.

Using hashtag in cross reference

Sometimes you need to link to the middle of a file (an anchor) rather than jump to the beginning of a file. DocFX also allows you to do that.

In Markdown link or auto link, you can add a hashtag (`#`) followed by the anchor name after UID. For example:

```
<xref:uid_to_file#anchor_name>

[link_text](xref:uid_to_file#anchor_name)

@uid_to_file#anchor_name
```

Will all resolve to `url_to_file#anchor_name` in output HTML.

The link text still resolves to the title of the whole file. If it's not what you need, you can specify your own link text.

> **NOTE**
>
> Hashtag in `xref` is always treated as separator between file name and anchor name. That means if you have `#` in UID, it has to be encoded to `%23`.
>
> The `xref` format follows the URI standard so that all reserved characters should be encoded.

Link to overwrite files

Overwrite file itself doesn't build into individual output file. It's merged with the API reference item model to build into a single file. If you want to link to the content inside an overwrite file (for example, an anchor), you cannot use the path to the overwrite file. Instead, you should either cross reference its UID, or link to the YAML file that contains the API.

For example, you have String class which is generated from `system.string.yml`, then you have a `string.md` that overwrites its conceptual part which contains a `compare-strings` section. You can use one of the following syntax to link to this section:

```
[compare strings](xref:System.String#compare-strings)

[compare strings](system.string.yml#compare-strings)
```

Both will render to:

```
<a href="system.string.html#compare-strings">compare strings</a>
```

Cross reference between projects

Another common need is to reference topics from an external project. For example, when you're writing the documentation for your own .NET library, you'll want to add some links that point to types in .NET base class library. DocFX gives you two ways to achieve this functionality: by exporting all UIDs in a project into a map file to be imported in another project, and through cross reference services.

> **TIP**
>
> Docfx automatically resolves .NET base class library types and other types published to https://learn.microsoft.com by default, without cross reference map or cross reference service. This process does not require network access.

Cross reference map file

When building a DocFX project, there will be an `xrefmap.yml` generated under output folder. This file contains information for all topics that have UID defined and their corresponding urls. The format of `xrefmap.yml` looks like this:

```
references:
- uid: uid_of_topic
  name: title_of_topic
  href: url_of_topic.html
  fullName: full_title_of_topic
- ...
```

It's a YAML object that contains following properties:

1. `references`: a list of topic information, each item contains following properties:
   - `uid`: UID to a conceptual topic or API reference
   - `name`: title of the topic
   - `href`: url to the topic, which is an absolute url or relative path to current file (`xrefmap.yml`)
   - `fullName`: doesn't apply to conceptual, means the fully qualified name of API. For example, for String class, its name is `String` and fully qualified name is `System.String`. This property is not used in link title resolve for now but reserved for future use.

> **TIP**
> The topic is not necessarily a file, it can also be a section inside a file. For example, a method in a class. In this case its url could be an anchor in a file.

Using cross reference map

Once you import a cross reference map file in your DocFX project, all UIDs defined in that file can be cross referenced.

To use a cross reference map, add a `xref` config to the `build` section of `docfx.json`:

```
{
  "build": {
    "xref": [
      "<path_to_xrefmap>"
    ],
    ...
  }
}
```

The value of `xref` could be a string or a list of strings that contain the path/url to cross reference maps.

> **NOTE**
> DocFX supports reading cross reference map from a local file or a web location. It's recommended to deploy `xrefmap.yml` to the website together with topic files so that others can directly use its url in `docfx.json` instead of downloading it to local.

Advanced: more options for cross reference

You can create a cross link with following options:

- `text`: the display text when the cross reference has been resolved correctly.

  e.g.: `@"System.String?text=string"` will be resolved as @"System.String?text=string".

- `alt`: the display text when the cross reference does not have a `href` property.

  e.g.: `<xref href="System.Collections.Immutable.ImmutableArray`1?alt=ImmutableArray"/>` will be resolved as ImmutableArray.

- `displayProperty`: the property of display text when the cross reference is has resolved correctly.

  e.g.: `<a href="xref:System.String?displayProperty=fullName"/>` will be resolved as .

- `altProperty`: the property of display text when the cross reference does not have a `href` property.

  e.g.: `<xref href="System.Collections.Immutable.ImmutableArray`1" altProperty="name"/>` will be resolved as System.Collections.Immutable.ImmutableArray`1.

- `title`: the title of link.

  e.g.: `[](xref:System.String?title=String+Class)` will be resolved as .

Create PDF Files

Docfx produces PDF files based on the TOC structure.

Install wkhtmltopdf

To build PDF files, first install `wkhtmltopdf` by downloading the latest binary from the official site or install using chocolatey: `choco install wkhtmltopdf`.

Make sure the `wkhtmltopdf` command is added to `PATH` environment variable and is available in the terminal.

PDF Config

Add a `pdf` section in `docfx.json`:

```
{
        "pdf": {
                "content": [
                        {
                                "files": [ "**/*.{md,yml}" ]
                        }
                ],
                "wkhtmltopdf": {
                        "additionalArguments": "--enable-local-file-access"
                }
        }
}
```

Most of the config options are the same as `build` config. The `wkhtmltopdf` config contains additional details to control `wkhtmltopdf` behavior:

- `filePath`: Path to `wkhtmltopdf.exe`.
- `additionalArguments`: Additional command line arguments passed to `wkhtmltopdf`. Usually needs `--enable-local-file-access` to allow access to local files.

Running `docfx` command againt the above configuration produces a PDF file for every TOC included in the `content` property. The PDF files are placed under the `_site_pdf` folder based on the TOC name.

See this sample on an example PDF config.

Add Cover Page

A cover page is the first PDF page before the TOC page.

To add a cover page, add a `cover.md` file alongside `toc.yml`. The content of `cover.md` will be rendered as the PDF cover page.

Config Reference

The `docfx.json` file indicates that the directory is the root of a docfx project.

```
{
  "build": { },
  "metadata": { },
  "pdf": { }
}
```

build

Configuration options that are applied for `docfx build` command:

```
{
  "build": {
    "content": ["**/*.{md|yml}"],
    "resource": ["**/media/**"],
    "globalMetadata": {
      "_appTitle": "My App"
    }
  }
}
```

content

Specifies an array of content files to include in the project. Supports File Mappings

```
{
  "build": {
    "content": ["**/*.{md,yml}"]
  }
}
```

resource

Specifies an array of resource files to include in the project. Supports File Mappings.

```
{
  "build": {
    "resources": ["**/*.png"]
```

```
  }
}
```

`overwrite`

Contains all the conceptual files that contain yaml headers with `uid` values and is intended to override the existing metadata `yml` files. Supports File Mappings.

`globalMetadata`

Contains metadata that will be applied to every file, in key-value pair format. For example, you can define `"_appTitle": "This is the title"` in this section, and when applying template `default`, it will be part of the page title as defined in the template.

```
{
  "build": {
    "globalMetadata": {
      "_appTitle": "DocFX website",
      "_enableSearch": "true"
    }
  }
}
```

See Predefined Metadata section for a list of predefined metadata.

`fileMetadata`

Specifies metadata associated with a particular file in order of metadata name, file glob patterns and metadata value:

```
{
  "build": {
    "fileMetadata": {
        "priority": {
            "**.md": 2.5,
            "spec/**.md": 3
        },
        "keywords": {
            "obj/docfx/**": ["API", "Reference"],
            "spec/**.md": ["Spec", "Conceptual"]
        },
        "_noindex": {
            "articles/**/article.md": true
        }
```

```
    }
  }
}
```

See Predefined Metadata section for a list of predefined metadata.

globalMetadataFiles

Set `globalMetadata` from external files.

```
{
  "build": {
    "globalMetadataFiles":  ["global1.json", "global2.json"]
  }
}
```

fileMetadataFiles

Set `fileMetadata` from external files.

```
{
  "build": {
    "fileMetadataFiles": ["file1.json", "file2.json"],
  }
}
```

template

The templates applied to each file in the documentation. Specify a string or an array. The latter ones will override the former ones if the name of the file inside the template collides. If omitted, the embedded `default` template will be used.

Templates are used to transform YAML files generated by `docfx` to human-readable pages. A page can be a markdown file, an html file or a plain text file. Each YAML file will be transformed to one page and be exported to the output folder preserving its relative path to `src`. For example, if pages are in HTML format, a static website will be generated in the output folder.

```
{
  "build": {
    "template": "custom",
```

```
  }
}
```

```
{
  "build": {
    "template": ["default", "my-custom-template"],
  }
}
```

> **NOTE**
>
> Docfx provides several builtin has embedded templates: `default`, `default(zh-cn)`, `pdf.default`, `statictoc` and `common`. Please avoid using these as template folder name.

`theme`

The themes applied to the documentation. Theme is used to customize the styles generated by `template`. It can be a string or an array. The latter ones will override the former ones if the name of the file inside the template collides. If omitted, no theme will be applied, the default theme inside the template will be used.

Theme is to provide general styles for all the generated pages. Files inside a theme will be generally copied to the output folder. A typical usage is, after YAML files are transformed to HTML pages, well-designed CSS style files in a Theme can then overwrite the default styles defined in template, e.g. `main.css`.

`xref`

Specifies the urls of xrefmap used by content files. Currently, it supports following scheme: http, https, file.

`exportRawModel`

If set to true, data model to run template script will be extracted in `.raw.json` extension.

`rawModelOutputFolder`

Specifies the output folder for the raw model. If not set, the raw model will appear in the same folder as the output documentation.

`exportViewModel`

If set to true, data model to apply template will be extracted in `.view.json` extension.

viewModelOutputFolder

Specifies the output folder for the view model. If not set, the view model will appear in the same folder as the output documentation.

dryRun

If set to true, the template will not be applied to the documents. This option is always used with `--exportRawModel` or `--exportViewModel` so that only raw model files or view model files will be generated.

maxParallelism

Sets the max parallelism. Setting 0 (default) is the same as setting to the count of CPU cores.

markdownEngineProperties

Sets the parameters for the markdown engine, value is a JSON object.

keepFileLink

If set to true, docfx does not dereference (i.e., copy) the file to the output folder, instead, it saves a `link_to_path` property inside `manifest.json` to indicate the physical location of that file. A file link will be created by incremental build and copy resource file.

sitemap

Specifies the options for generating sitemap.xml file:

```
{
  "build": {
    "sitemap":{
        "baseUrl": "https://dotnet.github.io/docfx",
        "priority": 0.1,
        "changefreq": "monthly",
        "fileOptions":{
            "**/api/**.yml": {
                "priority": 0.3,
                "lastmod": "2001-01-01",
            },
            "**/GettingStarted.md": {
                "baseUrl": "https://dotnet.github.io/docfx/conceptual",
                "priority": 0.8,
```

```
                "changefreq": "daily"
            }
        }
    }
  }
}
```

Generated sitemap.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>https://dotnet.github.io/docfx/api/System.String.html</loc>
    <lastmod>2001-01-01T00:00:00.00+08:00</lastmod>
    <changefreq>monthly</changefreq>
    <priority>0.3</priority>
  </url>
  <url>
    <loc>https://dotnet.github.io/docfx/conceptual/GettingStarted.html</loc>
    <lastmod>2017-09-21T10:00:00.00+08:00</lastmod>
    <changefreq>daily</changefreq>
    <priority>0.3</priority>
  </url>
  <url>
    <loc>https://dotnet.github.io/docfx/ReadMe.html</loc>
    <lastmod>2017-09-21T10:00:00.00+08:00</lastmod>
    <changefreq>monthly</changefreq>
    <priority>0.1</priority>
  </url>
</urlset>
```

`baseUrl`

Specifies the base url for the website to be published. It MUST begin with the protocol (such as http) and end with a trailing slash. For example, `https://dotnet.github.io/docfx/`. If the value is not specified, sitemap.xml will NOT be generated.

`lastmod`

Specifies the date of last modification of the file. If not specified, docfx automatically set the value to the time the file is built.

`changefreq`

Specifies the value of changefreq in sitemap.xml. Valid values are `always`, `hourly`, `daily`, `weekly`, `monthly`, `yearly`, `never`. If not specified, the default value is `daily`

`priority`

Specifies the value of priority in sitemap.xml. Valid values between `0.0` and `1.0`. If not specified, the default value is `0.5`

`fileOptions`

This property can be used when some specific files have different sitemap settings. It is a set of key-value pairs, where key is the *glob* pattern for input files, and value is the sitemap options. Order matters and the latter matching option overwrites the former ones.

`disableGitFeatures`

Disable fetching Git related information for articles. Set to `true` if fetching Git related information is slow for huge Git repositories. Default value is `false`.

`metadata`

Configuration options that are applied for `docfx metadata` command:

```
{
  "metadata": [
    {
      "src": [
        {
          "files": ["**/*.csproj"],
          "exclude": [ "**/bin/**", "**/obj/**" ],
          "src": "../src"
        }
      ],
      "dest": "api"
    }
  ]
}
```

`src`

Specifies the source projects using File Mappings.

`output`

Specifies the output folder of the generated metadata files relative to `docfx.json` directory. The `docfx metadata --output <outdir>` command line argument overrides this value.

outputFormat

Specifies the generated output file format.

- `mref` (default): output as ManagedReference YAML files.
- `markdown`: Output as common-mark compliant markdown file.

dest

Specifies the output folder of the generated metadata files relative to `docfx.json` directory. The `docfx metadata --output <outdir>` command line argument prepends this value.

shouldSkipMarkup

If set to true, DocFX would not render triple-slash-comments in source code as markdown.

filter

Specifies the filter configuration file, please go to How to filter out unwanted apis attributes for more details.

disableDefaultFilter

Disables the default filter configuration file.

disableGitFeatures

Disables generation of view source links.

properties

Specifies an optional set of MSBuild properties used when interpreting project files. These are the same properties that are passed to msbuild via the `/property:name=value` command line argument.

```
{
  "metadata": [
    {
      "properties": {
          "TargetFramework": "netstandard2.0"
      }
    }
```

```
    ]
}
```

> **NOTE**
> Make sure to specify `"TargetFramework": <one of the frameworks>` in your docfx.json when the project is targeting for multiple platforms.

noRestore

Do not run `dotnet restore` before building the projects.

namespaceLayout

Specifies how namespaces in TOC are organized:

- `flattened` (default): Renders namespaces as a single flat list.
- `nested`: Renders namespaces in a nested tree form.

memberLayout

Specifies how member pages are organized:

- `samePage` (default): Places members in the same page as their containing type.
- `separatePages`: Places members in separate pages.

allowCompilationErrors

When enabled, continues documentation generation in case of compilation errors.

EnumSortOrder

Specifies how enum members are sorted:

- `alphabetic` (default): Sort enum members in alphabetic order.
- `declaringOrder`: Sort enum members in the order as they are declared in the source code.

pdf

Configuration options that are applied for `docfx pdf` command:

name

Specifies the prefix of the generated PDF files, e.g. PDF generated from `testproject\toc.yml` is named as `{name}.pdf`, `testproject\api\toc.yml` is named as `{name}_api.pdf`. If not specified, the value of `name` is the folder name `testproject`.

`generatesAppendices`

If specified, an `appendices.pdf` file is generated containing all the not-in-TOC articles.

`keepRawFiles`

If specified, the intermediate html files used to generate the PDF are not deleted after the PDF has been generated.

`wkhtmltopdf`

Contains additional options specific to wkhtmltopdf which is used internally to generate the PDF files.

`filePath`

The path and file name of a wkhtmltopdf.exe compatible executable.

`additionalArguments`

Additional arguments that should be passed to the wkhtmltopdf executable. For example, pass `--enable-local-file-access` if you are building on a local file system. This will ensure that the supporting *.js and *.css files are loaded when rendering the HTML being converted to PDF.

`coverTitle`

The name of the bookmark to use for the cover page. If omitted, "Cover Page" will be used.

`tocTitle`

The name of the bookmark to use for the "Table of Contents". If omitted, "Table of Contents" will be used.

`outline`

The type of outline to use. Valid values are `NoOutline`, `DefaultOutline`, `WkDefaultOutline`. If not specified, the default value is `DefaultOutline`. If `WkDefaultOutline` is specified, `--outline` is passed to wkhtmltopdf; otherwise `--no-outline` is passed to wkhtmltopdf.

`noStdin`

Do not use `--read-args-from-stdin` for the wkhtmltopdf. Html input file names are set using the command line. It has been introduced to use in the Azure pipeline build. Can cause maximum allowed arguments length overflow if too many input parts (like Appendices, TocTitle, CoverPageTitle) were set for certain html source file.

`excludeDefaultToc`

If true, excludes the table of contents (generated by DocFX) in the PDF file.

File Mappings

In the short-hand form, these filenames are resolved relative to the directory containing the `docfx.json` file:

```
{
  "build": {
    "content": ["**/*.md", "TOC.yml"]
  }
}
```

In the expanded form, the `files` are resolved relative to `src` directory, or the directory containing the `docfx.json` file in absence of the `src` property:

```
{
  "build": {
    "content": [
      {"files": "docs/**/*.md", "dest": "docs"},
      {"files": ["**/*.yml", "*.md"], "exclude": ["**/*Private*"], "src": "../api", "de
    ]
  }
}
```

`files`

The file or file array, supports glob patterns.

`exclude`

The files to be excluded, supports glob patterns.

`src`

Specifies the source directory relative to the `docfx.json` directory, supports relative directory outside of the `docfx.json` directory such as `..`.

`dest`

The folder name for the generated files.

Glob Patterns

- `*`: Matches 0 or more charactors in a single path portion.
- `?`: Matches 1 character in a signle path portion.
- `**`: Matches 0 or more directories and subdirectories.
- `{}`: Expands the comma-delimited sections within the braces into a set.

Predefined Metadata

These are the standard metadata predefined by docfx. They are supported by builtin templates and SHOULD be supported by 3rd-party templates:

`_appTitle`

A string suffix appended to the title of every page.

`_appName`

The name of the site displayed after the logo.

`_appFooter`

The footer text. Shows Docfx's Copyright text if not specified.

`_appLogoPath`

Logo file's path from output root. Will show DocFX's logo if not specified. Remember to add file to resource.

`_appFaviconPath`

Favicon file's path from output root. Will show DocFX's favicon if not specified. Remember to add file to resource.

`_enableSearch`

Indicate whether to show the search box on the top of page.

`_enableNewTab`

Indicate whether to open a new tab when clicking an external link. (internal link always shows within the current tab)

_disableNavbar

Indicate whether to show the navigation bar on the top of page.

_disableBreadcrumb

Indicate whether to show breadcrumb on the top of page.

_disableToc

Indicate whether to show table of contents on the left of page.

_disableAffix

Indicate whether to show the affix bar on the right of page.

_disableContribution

Indicate whether to show the `View Source` and `Improve this Doc` buttons.

_gitContribute

Customize the `Improve this Doc` URL button for public contributors. Use `repo` to specify the contribution repository URL. Use `branch` to specify the contribution branch. Use `apiSpecFolder` to specify the folder for new overwrite files. If not set, the git URL and branch of the current git repository will be used.

_gitUrlPattern

Choose the URL pattern of the generated link for `View Source` and `Improve this Doc`. Supports `github` and `vso` currently. If not set, DocFX will try speculating the pattern from domain name of the git URL.

_noindex

File(s) specified are not returned in search results

Commandline Reference

Introduction

`docfx` is used to generate documentation for programs. It has the ability to:

1. Extract language metadata for programing languages as defined in Metadata Format Specification. Currently `C#` and `VB` are supported (although see note below regarding `VB` support). The language metadata is saved in `YAML` format as described in [YAML 1.2][1].
2. Look for available conceptual files as provided and link them with existing programs using the syntax described in Metadata Yaml Format. Supported conceptual files are *plain text* files, *html* files, and *markdown* files.
3. Generate documentation to a. Visualize language metadata, with extra **content** provided by linked conceptual files using the syntax described in Metadata Yaml Format. b. Organize and render available conceptual files which can be easily cross-referenced with language metadata pages. Docfx supports **Docfx Flavored Markdown(DFM)** for writing conceptual files. **DFM** supports all *Github Flavored Markdown(GFM)* syntax with 2 exceptions when resolving list. It also adds several new features including *file inclusion*, *cross reference*, and *yaml header*. For detailed description about DFM, please refer to DFM.

> **NOTE**
> Although Docfx is able to process `VB` projects and individual `VB` source code files and extract metadata from them, the documentation output from Docfx is always in `C#` format, i.e. types and member signatures, etc., are shown in `C#` format and not `VB` format.

Currently generating documentation to a *client only* **website** is supported. The generated **website** can be easily published to whatever platform such as *Github Pages* and *Azure Website* with little extra effort.

Generating offline documentation such as **PDF** is also supported.

Syntax

```
docfx <command> [<args>]
```

Run `docfx --version` to get the version of the docfx.

Run `docfx --help` or `docfx -h` to get a list of all available commands and options. Run `docfx <command> --help` or `docfx <command> -h` to get help on a specific command.

Commands

Init command `docfx init`

`docfx init` helps generate an `docfx.json` file.

Extract language metadata command `docfx metadata`

Syntax

```
docfx metadata [<projects>] [--property <n1>=<v1>;<n2>=<v2>]
```

Layout

```
|-- <metadata folder>
      |-- api
      |     |-- <namespace>.yml
      |     |-- <class>.yml
      |-- toc.yml
      |-- index.yml
```

Optional arguments for the `docfx metadata` command

- **`<projects>` argument (optional)**

  `<projects>` specifies the projects to have metadata extracted. There are several approaches to extract language metadata.

  1. From a supported file or file list Supported file extensions include `.csproj`, `.vbproj`, `.sln`, `project.json`, `dll` assembly file, `.cs` source file and `.vb` source file.

     ```
     Multiple files are separated by whitespace, e.g. `docfx metadata Class1.cs a.csproj

     > [!Note]
     > Glob pattern is **NOT** supported in command line options.
     ```

  2. From *docfx.json* file, as described in **Section3**.

  3. If the argument is not specified, `docfx` will try reading `docfx.json` under current directory.

The default output folder is `_site/` folder if it is not specified in `docfx.json` under current directory.

- **`--shouldSkipMarkup` command option**

  If adding option `--shouldSkipMarkup` in metadata command, it means that DocFX would not render triple-slash-comments in source code as markdown.

  e.g. `docfx metadata --shouldSkipMarkup`

- **`--property <n1>=<v1>;<n2>=<v2>` command option**

An optional set of MSBuild properties used when interpreting project files. These are the same properties that are passed to msbuild via the /property:=;= command line argument. For example: `docfx metadata --property TargetFramework=net48` generates metadata files with .NET framework 4.8. This command can be used when the project supports multiple `TargetFrameworks`.

Generate documentation command `docfx build`

Syntax

```
docfx build [-o:<output_path>] [-t:<template folder>]
```

`docfx build` generates documentation for current folder.

If `toc.yml` or `toc.md` is found in current folder, it will be rendered as the top level TABLE-OF-CONTENT. As in website, it will be rendered as the top navigation bar. Path in `toc.yml` or `toc.md` are relative to the TOC file.

> **NOTE**
>
> `homepage` is not supported in `toc.md`. If `href` is referencing a **folder**, it must end with `/`.

toc.yml syntax

`toc.yml` is an array of items. Each item can have following properties:

Property	Description
name	**Required**. The title of the navigation page.

Property	Description
href	**Required**. A folder or a file *UNDER* the current folder. A folder must end with `/`. If referencing a folder, a TOC.md file inside the folder will be rendered as a second level TABLE-OF-CONTENT. As in website, it will be rendered as a sidebar.
homepage	The default content shown when no article is selected.

TOC.yml Sample

```
- name: Home
  href: articles/Home.md
- name: Roslyn Wiki
  href: roslyn_wiki/
- name: Roslyn API
  href: api_roslyn/
  homepage: homepages/roslyn_language_features.md
```

TOC.md Sample

```
## [Home](articles/Home.md)
## [Roslyn Wiki](roslyn_wiki/)
## [Roslyn API](api_roslyn/)
```

Optional arguments for the `docfx build` command

- **`<output_path>` argument (optional)**

  The default output folder is `_site/` folder

- **`<template folder>` argument (optional)**

  If specified, use the template from template folder

Template Folder Structure

```
|-- <template folder>
      |-- index.html
      |-- styles
      |     |-- docascode.css
      |     |-- docascode.js
      |-- template
```

```
|     |-- toc.html
|     |-- navbar.html
|     |-- yamlContent.html
|-- favicon.ico
|-- logo.ico
```

Generate PDF documentation command `docfx pdf`

Syntax

```
docfx pdf [<config_file_path>] [-o:<output_path>]
```

`docfx pdf` generates PDF for the files defined in config file, if config file is not specified, `docfx` tries to find and use `docfx.json` file under current folder.

> **NOTE**
>
> Prerequisite: We leverage wkhtmltopdf to generate PDF. Download wkhtmltopdf and save the executable folder path to **%PATH%**. Or just install wkhtmltopdf using chocolatey: `choco install wkhtmltopdf`

Current design is that each TOC file generates a corresponding PDF file. Walk through Walkthrough: Generate PDF Files to get start.

If `cover.md` is found in a folder, it will be rendered as the cover page.

Environment Variables

DOCFX_KEEP_DEBUG_INFO

If set true. Keep following debug info in output HTML.

- `sourceFile`
- `sourceStartLineNumber`
- `sourceEndLineNumber`
- `jsonPath`
- `data-raw-source`
- `nocheck`

DOCFX_SOURCE_BRANCH_NAME

Used to override git branch name.

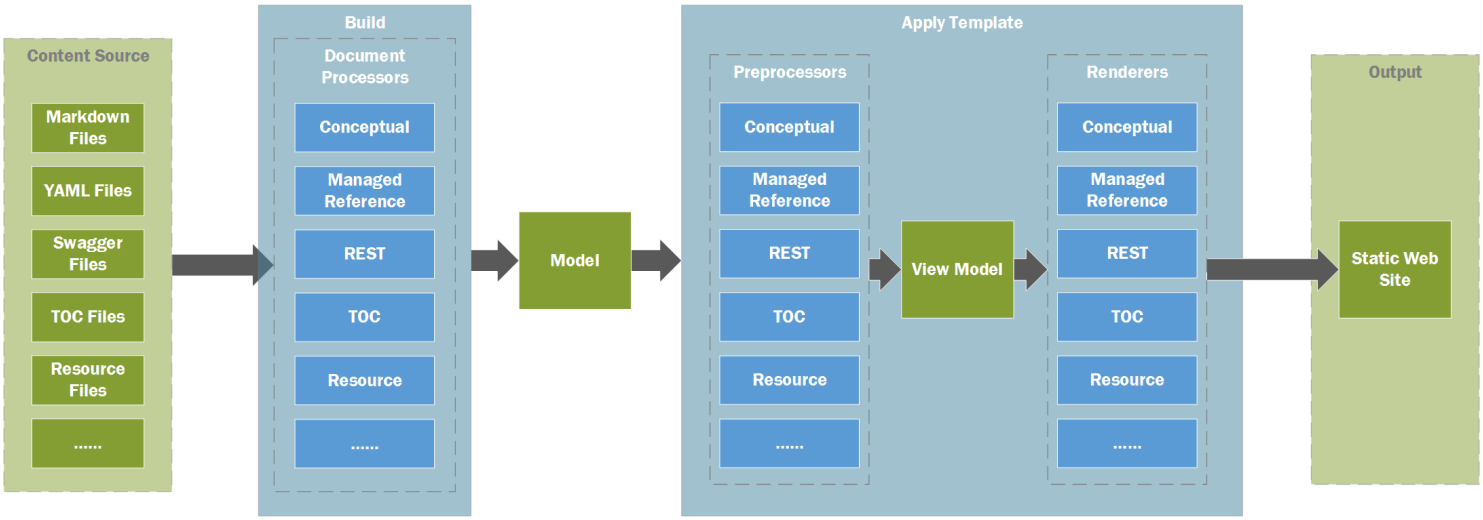
DOCFX_NO_CHECK_CERTIFICATE_REVOCATION_LIST

Used to disable CRL check. This setting is intended to be used on offline environment.

Introduction to the DocFX Template System

The DocFX template system provides a flexible way of defining and using templates to control how the final output files are rendered. These files provide the **content** used to publish a DocFx-generated web site. Note that this is different than the HTML templates discussed in Walkthrough Advanced: Customize Your Website, which are used to control the **styling** applied to the web site.

As the following DocFX workflow shows,



DocFX loads the set of files and transforms them into different data models using different types of *Document Processor*s. Afterwards, the template system loads these data models, and transforms them into output files based on the *document type* of the data model.

Each file belongs to a *document type*. For example, the document type for Markdown files is `conceptual`, and the document type for `toc.md` files is `Toc`.

For a specific *Template*, each *document type* can have several *Renderer*s. For a specific file, the template system picks the corresponding *Renderer*s to render the input data model into output files.

Renderer

*Renderer*s are files written in Mustache. It is used to transform the input data model into output files.

Naming rule for a *Renderer* file

The naming rule for a *Renderer* file is: `<document_type>.<output_extension>[.primary].tmpl`.

- `<document_type>` is the *document type* current *Renderer* responsible to.

- `<output_extension>` defines the extension of the output files going through current *Renderer*. For example, `conceptual.html.tmpl` transforms `file1.md` into output file `file1.html`, and `toc.json.tmpl` transforms `toc.md` into output file `toc.json`.
- `[.primary]` is optional. It is used when there are multiple *Renderer*s with different extension for one particular document type. The output file transformed by the `.primary` *Renderer* is used as the file to be linked. The below example describes the behavior in detail.

Here is an example.

The following template contains two Mustache *Renderer* files for `conceptual` document type:

```
/- some_template/
    |- conceptual.html.primary.tmpl
    \- conceptual.mta.json.tmpl
```

There are two Markdown files `A.md` and `B.md`, the content for `A.md` is:

```
[Link To B](B.md)
```

The template system produces two output files for `A.md`: `A.html` and `A.mta.json`, and also two output files for `B.md`: `B.html` and `B.mta.json`. According to `conceptual.html.primary.tmpl`, `.html` is the **primary** output file, the link from `A.md` to `B.md` is resolved to `B.html` instead of `B.mta.json`, which is to say, the content of `A.md` is transformed to:

```
<a href="B.html">Link To B</a>
```

> **NOTE**
>
> If no `primary` *Renderer* is defined, DocFX randomly picks one *Renderer* as the primary one, and the result is unpredictable.

Renderer in Mustache syntax

Introduction to Mustache

Mustache is a logic-less template syntax containing only *tag*s. It works by expanding tags in a template using values provided in a hash or object. *Tag*s are indicated by the double mustaches. `{{name}}` is a tag, it tries to find the **name** key in current context, and replace it with the value of **name**. mustache.5 lists the syntax of Mustache in detail.

Naming rule

*Renderer*s in Mustache syntax **MUST** end with `.tmpl` extension.

Mustache Partials

Mustache Partials is also supported in the template system. **Partials** are common sections of *Renderer* that can be shared by multiple *Renderer* files. **Partials MUST** end with `.tmpl.partial`.

For example, inside a *Template*, there is a **Partial** file `part.tmpl.partial` with content:

```
Inside Partial
{{ name }}
```

To reuse this **Partial** file, *Renderer* file uses the following syntax:

```
Inside Renderer
{{ >part }}
```

It has the same effect with the following *Renderer* file:

```
Inside Renderer
Inside Partial
{{ name }}
```

Extended syntax for Dependencies

When rendering the input data model into output files, for example, html files, the html file may rely on other files to display correctly. For example, the html file dependents on stylesheet file `main.css`. We call such file `main.css` a *Dependency* to the *Renderer*.

DocFX introduces the following syntax to define the dependency for the *Renderer*:

```
{{!include('<file_name>')}}
```

`docfx` copies these dependencies to output folder preserving its relative path to the *Renderer* file.

> **TIP**
>
> Mustache is logic-less, and for a specific `{{name}}` tag, Mustache searches its context and its parent context recursively. So most of the time *Preprocessor File* is used to re-define the data model used by the Mustache *Renderer*.

Extended syntax for Master page

In most cases templates with different document types share the same layout and style. For example, most of the pages can share navbar, header, or footer.

DocFX introduces the following syntax to use a master page:

```
{{!master('<master_page_name>')}}
```

Inside the master page, the following syntax is used for pages to place their content body:

```
{{!body}}
```

For example, with the following master page `_master.html`:

```
<html>
    <head></head>
    <body>
        {{!body}}
    <body>
</html>
```

A template `conceptual.html.tmpl` as follows:

```
{{!master('_master.html')}}
Hello World
```

renders as the same as:

```
<html>
    <head></head>
    <body>
        Hello World
    <body>
</html>
```

Preprocessor

*Renderer*s take the input data model produced by the document processor and render them into output files. Sometimes the input data model is not exactly what the *Renderer*s want. The template system introduces the concept of *Preprocessor* to transform the input data model into exactly what the *Renderers* want. We call the data model returned by the *Preprocessor* the *View Model*. The *View Model* is the data model applied to the *Renderers*.

Naming rule for *Preprocessor*

The naming of the *Preprocessor* follows the naming of the *Renderer*, with file extension changes to `.js`: `<renderer_file_name_without_extension>.js`.

If a *Preprocessor* has no corresponding *Renderer*, it still needs to be executed. For example, to run `exports.getOptions` function, it should be named as `<document_type>.tmpl.js`.

Syntax for *Preprocessor*

*Preprocessor*s are JavaScript files following ECMAScript 5.1 standard. The template system uses Jint as JavaScript Engine, and provides several additional functions for easy debugging and integration.

Module

Preprocessor leverages the concept of *Module* as similar to the Module in Node.js. The syntax of Module in *Preprocessor* is a *subset* of the one in Node.js. The advantage of the Module concept is that the *Preprocessor* script file can also be run in Node.js. The Module syntax in *Preprocessor* is simple,

1. To export function property from one Module file `common.js`:

   ```
   exports.util = function () {}
   ```

2. To use the exported function property inside `common.js`:

   ```
   var common = require('./common.js');
   // call util
   common.util();
   ```

> **NOTE**
> Only relative path starting with `./` is supported.

Log

You can call the following functions to log messages with different error levels: `console.log`, `console.warn` or `console.warning` and `console.err`.

Function Signature

A *Preprocessor* file is also considered as a Module. It **MUST** export the function property with the signature required by DocFx's prescriptive interop pattern.

There are two functions defined.

Function 1: `exports.getOptions`

Function property `getOptions` takes the data model produced by document processor as the input argument, and the return value must be an object with the following properties:

Property Name	Type	Description
isShared	bool	Defines whether the input data model can be accessed by other data models when `transform`. By default the value is `false`. If it is set to `true`, the data model will be stored into Globally Shared Properties.

A sample `exports.getOptions` defined in `toc.tmpl.js` is:

```
exports.getOptions = function (model) {
    return {
        isShared: true;
    };
}
```

Function 2: `exports.transform`

Function property `transform` takes the data model produced by document processor (described in further detail in The Input Data Model) as the input argument, and returns the *View Model*. *View Model* is the exact model to apply the corresponding *Renderer*.

A sample `exports.transform` for `conceptual.txt.js` is:

```
exports.transform = function (model) {
    model._title = "Hello World"
    return model;
}
```

If `conceptual.txt.tmpl` is:

```
{{{_title}}}
```

Then Markdown file `A.md` is transformed to `A.txt` with content:

```
Hello World
```

> **TIP**
>
> For each file, the input data model can be exported to a JSON file by calling `docfx build --exportRawModel`. And the returned *View Model* can be exported to a JSON file by calling `docfx build --exportViewModel`. The output files are stored in the DocFx destination subdirectory, which defaults to `<project-name>\_site\`.

The Input Data Model

The input data model used by `transform` not only contains properties extracted from the content of the file, but also system generated properties and globally shared properties.

System Generated Properties

System generated property names start with underscore `_`, as listed in the following table:

Name	Description
_rel	The relative path of the root output folder from current output file. For example, if the output file is `a/b/c.html` from root output folder, then the value is `../../`.
_path	The path of current output file starting from root output folder.
_navPath	The relative path of the root TOC file from root output folder, if exists. The root TOC file stands for the TOC file in root output folder. For example, if the output file is html file, the value is `toc.html`.
_navRel	The relative path from current output file to the root TOC file, if exists. For example, if the root TOC file is `toc.html` from root output folder, the value is empty.
_navKey	The original file path of the root TOC file starting with `~/`. `~/` stands for the folder where `docfx.json` is in, for example, `~/toc.md`.
_tocPath	The relative path of the TOC file that current output file belongs to from root output folder, if current output file is in that TOC file. If current output file is not defined in any TOC file, the nearest TOC file is picked.
_tocRel	The relative path from current output file to its TOC file. For example, if the TOC file is `a/toc.html` from root output folder, the value is `../`.

Name	Description
_tocKey	The original file path of the TOC file starting with `~/`. `~/` stands for the folder where `docfx.json` is in, for example, `~/a/toc.yml`.

> **NOTE**
>
> Users can also override system generated properties by using *YAML Header*, `fileMetadata` or `globalMetadata`.

Globally Shared Properties

Globally shared properties are stored in `__global` key for every data model. Its initial value is read from `global.json` inside the *Template* if the file exists. If a data model has `isShared` equal to `true` with the above `getOptions` function property, it is stored in `__global._shared` with the original path starting with `~/` as the key.

How-to: Create A Custom Template

Templates are organized as a zip package or a folder. The file path (without the `.zip` extension) of the zip package or the path of the folder is considered to be the template name.

Quickstart

Let's create a template to transform Markdown files into a simple html file.

Step 1. Create a template folder

Create a folder for the template, for example, `c:/docfx_howto/simple_template`.

Step 2. Add *Renderer* file

Create a file `conceptual.html.primary.tmpl` under the template folder with the following content:

```
{{{conceptual}}}
```

Now a simple custom template is created.

You may notice that DocFX reports a warning message saying that: *Warning: [Build Document.Apply Templates]There is no template processing document type(s): Toc*. It is because our custom template only specifies how to handle document with type `conceptual`.

In the documentation project, run `docfx build docfx.json -t c:/docfx_howto/simple_template --serve`. The `-t` command option specifies the template name(s) used by the current build.

Open http://localhost:8080 and you can see a simple web page as follows:

Add *Preprocessor* file

Step 3. Add *Preprocessor* file

Sometimes the input data model is not exactly what *Renderer* wants, you may want to add some properties to the data model, or modify the data model a little bit before applying the *Renderer* file. This can be done by creating a *Preprocessor* file.

Create a file `conceptual.html.primary.js` under the template folder with the following content:

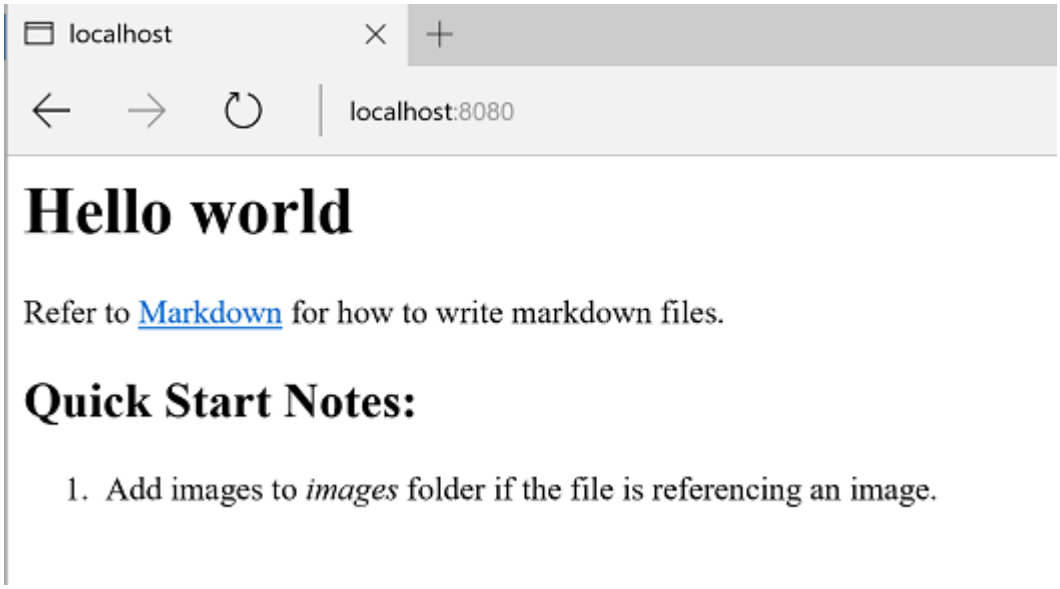
```
exports.transform = function (model) {
    model._extra_property = "Hello world";
    return model;
}
```

Update the file `conceptual.html.primary.tmpl` with the following content:

```
<h1>{{_extra_property}}</h1>
{{{conceptual}}}
```

In the documentation project, run `docfx build docfx.json -t c:/docfx_howto/simple_template --serve`.

Open http://localhost:8080 and you can see `_extra_property` is added to the web page.



Merge template with `default` template

DocFX contains some embedded template resources that you can refer to directly. You can use `docfx template list` to list available templates provided by DocFX.

Take `default` template as an example.

Run `docfx template export default`. It exports what's inside `default` template into the folder `_exported_templates`. You can see that there are sets of *Preprocessor* and *Renderer* files to deal with different types of documents.

DocFX supports specifying multiple templates for a documentation project. That allows you to leverage the `default` template for handling other types of documents, together with your custom template.

When dealing with multiple templates, DocFX merges the files inside these templates.

The principle for merging is: if a file name collides then the file in the latter template overwrites the one in the former template.

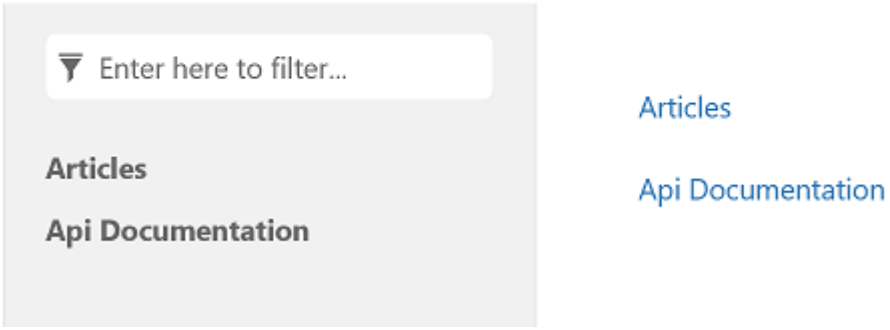
For example, you can merge `default` template and your custom template by calling `docfx build docfx.json -t default,c:/docfx_howto/simple_template`. Multiple templates are split by a comma `,` in the command line. Or you can define it in `docfx.json` by:

```
"build": {
    "template": [
        "default",
        "c:/docfx_howto/simple_template"
    ]
}
```

In the documentation project, run `docfx build docfx.json -t default,c:/docfx_howto/simple_template --serve`.

Now the warning message *There is no template processing document type(s): Toc* disappears because the default template contains *Renderer* to handle TOC files.

Open http://localhost:8080/toc.html and you can see a toc web page.



> **TIP**
>
> Run `docfx template export default` to view what's inside the default template.

> **NOTE**
>
> It is possible that DocFX updates its embedded templates when a new version is released. So please make sure to re-export the template if you overwrite or are dependent on it in your custom template.

Extension for *Preprocessor* file

If you want to modify some properties based on DocFX `default` template's *Preprocessor*, you can use *Preprocessor* extension file to achieve this.

For example, if you want to add a property to the managed reference's data model after `default` template's *Preprocessor*, you can update the file `ManagedReference.extension.js` in your custom template with the following content:

```
/**
 * This method will be called at the start of exports.transform in ManagedReference.htm
 */
exports.preTransform = function (model) {
  return model;
```

```
}

/**
 * This method will be called at the end of exports.transform in ManagedReference.html.
 */
exports.postTransform = function (model) {
  model._extra_property = "Hello world";
  return model;
}
```

Compared with modifying `ManagedReference.html.primary.js` directly, you needn't worry about merging your custom templates with DocFX's embedded templates when DocFX updates.

Doc-as-Code: Metadata Format Specification

0. Introduction

0.1 Goals and Non-goals

1. The goal of this document is to define a general format to describe language metadata for programming languages.
2. The language metadata is designed to be language agnostic and support multiple programming language in a single metadata file.
3. The main user scenario for language metadata is to generate reference documentation, so this document will discuss how to optimize metadata format for documentation rendering.
4. This document does **NOT** discuss details of metadata format implementation of a specific programming language.

0.2 Terminology

The key words **MUST**, **MUST NOT**, **REQUIRED**, **SHALL**, **SHALL NOT**, **SHOULD**, **SHOULD NOT**, **RECOMMENDED**, **MAY**, and **OPTIONAL** in this document are to be interpreted as described in RFC 2119.

Words in *italic* indicate they are terms previously defined in this document.

1. Items and Identifiers

1.1 Items

Item is the basic unit of metadata format. From a documentation perspective, each *item* represents a "section" in the reference documentation. This "section" is the minimum unit that you can cross reference to, or customize in layout and content.

> When implementing the metadata format for your own language, you can decide which elements are *items*. For example, usually namespaces, classes, and methods are *items*. However, you can also make smaller elements such as parameters be items if you want them to be referenceable and customizable.

Items can be hierarchical. One *item* can have other *items* as *children*. For example, in C#, namespaces and classes can have classes and/or methods as *children*.

1.2 Identifiers

Each *item* has an identifier (ID) which is unique under its parent.

As we're targeting to support multiple languages, there are no restrictions as to which characters are not allowed in identifiers. However, to make identifiers easier to recognize and resolve in Markdown, it's not **RECOMMENDED** to have whitespaces in identifiers. Markdown processor **MAY** implement some algorithm to tolerate whitespaces in handwritten Markdown. (Leading and trailing spaces **MUST** be removed from identifier.)

Identifier **MUST** be treated as case-sensitive when comparing equality.

Each *item* has a unique identifier (UID) which is globally unique. A *UID* is defined as follows:

1. If an *item* does not have a *parent*, its *UID* is its *ID*.
2. Otherwise, its *UID* is the combination of the *UID* of its *parent*, a separator and the *ID* of the *item* itself.

Valid separators are `.`, `:`, `/` and `\`.

For example, for a class `String` under namespace `System`, its ID is `String` and UID is `System.String`.

> Given the above definition, an *item*'s UID **MUST** starts with the *UID* of its parent (and any of its ancestors) and ends with the *ID* of itself. This is useful to quickly determine whether an *item* is under another *item*.

1.3 Alias

Identifier could be very long, which makes it difficult to write by hand in Markdown. For example, it's easy to create a long *ID* in C# like this:

```
Format(System.IFormatProvider,System.String,System.Object,System.Object)
```

We can create short *alias* for *items* so that they can be referenced easily.

Alias is same as *ID*, except:

1. It doesn't have to be unique.
2. One *item* can have multiple *aliases*.

> It's not **RECOMMENDED** to create an *alias* that has nothing to do with an *item*'s *ID*.
> Usually an *item*'s *alias* is part of its *ID* so it's easy to recognize and memorize.
> For example, for the case above, we usually create an alias `Format()`.

We can easily get a "global" alias for an *item* by replacing the *ID* part of its *UID* with its alias.

2. File Structure

2.1 File Format

You can use any file format that can represent structural data to store metadata. However, we recommend using YAML or JSON. In this document, we use YAML in examples, but all YAML can be converted to JSON easily.

2.2 File Layout

A metadata file consists of two parts: An "item" section and a "reference" section. Each section is a list of objects and each object is a key-value pair (hereafter referred to as "property") list that represents an *item*.

2.3 Item Section

Though *items* can be hierarchical, they are flat in an *item* section. Instead, each *item* has a "children" *property* indicating its *children* and a "parent" *property* indicating its parent.

An *item* object has some basic properties:

Property	**Description**
uid	**REQUIRED**. The *unique identifier* of the *item*.
children	**OPTIONAL**. A list of *UIDs* of the *item*'s children. Can be omitted if there are no *children*.
parent	**OPTIONAL**. The *UID* of the *item*'s parent. If omitted, metadata parser will try to figure out its *parent* from the *children* information of other *items* within the same file.

Here is an example of a YAML format metadata file for C# Object class:

```yaml
items:
- uid: System.Object
  parent: System
  children:
  - System.Object.Object()
  - System.Object.Equals(System.Object)
  - System.Object.Equals(System.Object,System.Object)
  - System.Object.Finalize()
```

```
    - System.Object.GetHashCode()
    - System.Object.GetType()
    - System.Object.MemberwiseClone()
    - System.Object.ReferenceEquals()
    - System.Object.ToString()
  - uid: System.Object.Object()
    parent: System.Object
  - uid: System.Object.Equals(System.Object)
    parent: System.Object
  - uid: System.Object.Equals(System.Object,System.Object)
    parent: System.Object
  - uid: System.Object.Finalize()
    parent: System.Object
  - uid: System.Object.GetHashCode()
    parent: System.Object
  - uid: System.Object.GetType()
    parent: System.Object
  - uid: System.Object.MemberwiseClone()
    parent: System.Object
  - uid: System.Object.ReferenceEquals()
    parent: System.Object
  - uid: System.Object.ToString()
    parent: System.Object
references:
...
```

> *Items* **SHOULD** be organized based upon how they will display in documentation. For example, if you want all members of a class be displayed in a single page, put all members in a single metadata file.

2.3 Item Object

In additional to the *properties* listed in last section, *item object* also has some **OPTIONAL** *properties*:

Property	**Description**
id	The *identifier* of the *item*.
alias	A list of *aliases* of the *item*.
name	The display name of the *item*.

Property	Description
fullName	The full display name of the *item*. In programming languages, it's usually the full qualified name.
type	The type of the *item*, such as class, method, etc.
url	If it's a relative URL, then it's another metadata file that defines the *item*. If it's an absolute URL, it means the *item* is coming from an external library, and the URL is the documentation page of this *item*. If omitted, the URL is the location of the current file.
source	The source code information of the *item*. It's an object that contains following *properties*: 1. repo: the remote Git repository of the source code. 2. branch: the branch of the source code. 3. revision: the Git revision of the source code. 4. path: the path to the source code file where the *item* is defined. 5. startLine: the start line of the *item* definition. 6. endLine: the end line of the *item* definition.

Here is an example of a C# Dictionary class:

```yaml
- uid: System.Collections.Generic.Dictionary`2
  id: Dictionary`2
  alias:
  - Dictionary
  parent: System.Collections.Generic
  name: Dictionary<TKey, TValue>
  fullName: System.Collections.Generic.Dictionary<TKey, TValue>
  type: class
  url: System.Collections.Generic.Dictionary`2.yml
  source:
    repo: https://github.com/dotnet/netfx.git
    branch: master
    revision: 5ed47001acfb284a301260271f7d36d2bb014432
    path: src/system/collections/generic/dictionary.cs
    startLine: 1
    endLine: 100
```

2.4 Custom Properties

Besides the predefined *properties*, *item* can have its own *properties*. One restriction is *property* name **MUST NOT** contains dots, as dot in *property* name will have special meaning (described in later section).

2.5 Reference Section

The reference section also contains a list of *items*. These *items* serve as the references to *items* in the *item section* and won't show up in documentation. Also, a reference *item* doesn't need to have full *properties*, it just contains necessary information needed by its referrer (for example, name or URL).

In metadata file, all *items* **MUST** be referenced by *UID*.

> It's **RECOMMENDED** to include all referenced *items* in reference section. This makes the file self-contained and easy to render at runtime.
>
> Many programming languages have the concept of "template instantiation". For example, in C#, you can create a new type `List<int>` from `List<T>` with argument `int`. You can create a reference for "template instances". For example, for a class inherited from `List<int>`:

```
items:
- uid: NumberList
  inherits:
  - System.Collections.Generic.List<System.Int32>
references:
- uid: System.Collections.Generic.List`1<System.Int32>
  link: @"System.Collections.Generic.List`1"<@"System.Int32">
- uid: System.Collections.Generic.List`1
  name: List
  url: system.collections.generic.list`1.yml
- uid: System.Int32
  name: int
  url: system.int32.yml
```

2.6 Multiple Language Support

An *item* may need to support multiple languages. For example, in .NET, a class can be used in C#, VB, managed C++ and F#. Different languages may have differences in *properties*. For example, a list of string is displayed as `List<string>` in C#, while `List(Of string)` in VB.

To support this scenario, we introduce a concept of language context to allow defining different *property* values in different languages.

If a *property* name is in the form of `property_name.language_name`, it defines the value of `property_name` under `language_name`. For example:

```
- uid: System.Collections.Generic.Dictionary`2
  name.csharp: Dictionary<TKey, TValue>
  name.vb: Dictionary(Of TKey, TValue)
```

This means the name of dictionary is `Dictionary<TKey, TValue>` in C# and `Dictionary(Of TKey, TValue)` in VB.

The following *properties* **SHALL NOT** be overridden in language context: uid, id, alias, children, and parent.

3. Work with Metadata in Markdown

3.1 YAML Metadata Section

In a Markdown file, you can also define *items* using the same metadata syntax. The metadata definition **MUST** be in YAML format and enclosed by triple-dash lines (`---`). Here is an example:

```
---
uid: System.String
summary: String class
---

This is a **string** class.
```

You can have multiple YAML sections inside a single Markdown file, but in a single YAML section, there **MUST** be only one *item*.

The YAML metadata section does not have to contain all *properties*. The only *property* that **MUST** appear is "uid", which is used to match the same *item* in metadata file.

The most common scenario for using YAML section is to specify which *item* the markdown doc belongs to. But you can also overwrite *item property* by defining one with the same name in YAML section. In the above example, the *property* "summary" will overwrite the same one in metadata.

As with language context, the following *properties* **SHALL NOT** be overridden: uid, id, alias, children, and parent.

You **SHALL NOT** define new *item* in Markdown.

3.2 Reference Items in Markdown

To cross reference an *item*, you can use URI with `xref` scheme. You can either use standard link or automatic link with the above URI. For example, to cross reference `System.String`:

```
[System.String](xref:System.String)

<xref:System.String>
```

> Since *item* reference is a URI, special characters (like `#`, `?`) **MUST** be encoded.

We also introduce a shorthand markdown syntax to cross reference easily:

If a string starts with `@`, and followed by a string enclosed by quotes `'` or double quotes `"`, it will be treated as an *item* reference. The string inside `""` or `''` is the *UID* of the *item*. Here is one example:

```
@"System.String"
```

> Markdown processor **MAY** implement some algorithm to allow omit curly braces if *ID* is simple enough. For example, For reference like `@"int"`, we may also want to allow `@int`.

When rendering references in Markdown, they will expand into a link with the *item*'s name as link title. You can also customize the link title using the standard syntax of Markdown:

```
[Dictionary](xref:System.Collections.Generic.Dictionary`2)<[String](xref:System.String)
```

Will be rendered to: Dictionary<String, String>

Besides *UID*, we also allow referencing items using *ID* and *alias*, in the Markdown processor, the below algorithm **SHOULD** be implemented to resolve references.

Check whether the reference matches:

1. Any *identifier* of current *item*'s children.
2. Any *alias* of current *item*'s children.
3. Any *identifier* of current *item*'s silbings.
4. Any *alias* of current *item*'s silbings.
5. A *UID*.
6. A *global alias*.

.NET API Docs YAML Format

This document describes the YAML file format to represent .NET API docs.

The first line of the YAML file is the magic header `### YamlMime:ManagedReference`.

1. Items

The following .NET elements are defined as *items* in metadata:

1. Namespaces
2. Types, including class, struct, interface, enum, delegate
3. Type members, including field, property, method, event

Other elements such as parameters and generic parameters are not standalone *items*, they're part of other *items*.

2. Identifiers

2.1 Unique Identifiers

For any *item* in .NET languages, its *UID* is defined by concatenating its *parent*'s *UID* and its own *ID* with a dot. The *ID* for each kind of item is defined in following sections. The basic principle here is to make *ID* format close to source code and easy for human reading.

UID is similar to the document comment id, which is started with type prefix, for example, `T:`, or `M:`, but *UID* do not.

There **MUST NOT** be any whitespace between method name, parentheses, parameters, and commas.

2.2 Spec Identifiers

Spec identifier is another form of *UID*. It can spec a generic type with type arguments (for example, for parameters, return types or inheritances) and these *UID*s are unique in one yaml file.

It is a simple modified Unique Identifiers, when it contains generic type arguments, it will use `{Name}` instead `` `N ``. For type parameter, it will be `{Name}`. And it also supports array and pointer.

> Example 2.2 Spec Identifier
>
> C#:

```
namespace Foo
{
   public class Bar
   {
      public unsafe List<String> FooBar<TArg>(int[] arg1, byte* arg2, TArg arg3, Li
      {
          return null;
      }
   }
}
```

YAML:

```
references:
- uid: System.Collections.Generic.List{System.String}
- uid: System.Int32[]
- uid: System.Byte*
- uid: {TArg}
- uid: System.Collections.Generic.List{{TArg}[]}
```

3. Namespaces

For all namespaces, they are flat, e.i. namespaces do not have the parent namespace. So for any namespace, *ID* is always same with its *UID*.

Example 3 Namespace

C#:

```
namespace System.IO
{
}
```

YAML:

```
uid: System.IO
id: System.IO
name: System.IO
fullName: System.IO
```

The children of namespace are all the visible types in the namespace.

4. Types

Types include classes, structs, interfaces, enums, and delegates. They have following properties: summary, remarks, syntax, namespace, assemblies, inheritance. The *parents* of types are namespaces. The *children* of types are members.

ID

ID for a type is also its *name*.

> Example 4 Type
>
> C#:
>
> ```
> namespace System
> {
>     public class String {}
>     public struct Boolean {}
>     public interface IComparable {}
>     public enum ConsoleColor {}
>     public delegate void Action();
> }
> ```
>
> YAML:
>
> ```
> - uid: System.String
>   id: String
>   name.csharp: String
>   fullName.csharp: System.String
> - uid: System.Boolean
>   id: Boolean
>   name.csharp: Boolean
>   fullName.csharp: System.String
> - uid: System.IComparable
>   id: IComparable
>   name.csharp: IComparable
>   fullName.csharp: System.IComparable
> - uid: System.ConsoleColor
>   id: ConsoleColor
>   name.csharp: ConsoleColor
>   fullName.csharp: System.ConsoleColor
> - uid: System.Action
>   id: Action
> ```

```
  name.csharp: Action
  fullName.csharp: System.Action
```

4.1 ID for Nested Types

For nested types, *ID* is defined by concatenating the *ID* of all its containing types and the *ID* of itself, separated by a dot.

The parent type of a nested type is its containing namespace, rather than its containing type.

> Example 4.1 Nested type
>
> C#:
>
> ```
> namespace System
> {
>     public class Environment
>     {
>         public enum SpecialFolder {}
>     }
> }
> ```
>
> YAML:
>
> ```
> uid: System.Environment.SpecialFolder
> id: Environment.SpecialFolder
> name.csharp: Environment.SpecialFolder
> fullName.csharp: System.Environment.SpecialFolder
> ```

4.2 Inheritance

Only class contains inheritance, and the inheritance is a list of spec id.

> Example 4.2 Inheritance
>
> C#:
>
> ```
> namespace System.Collections.Generic
> {
>     public class KeyedByTypeCollection<TItem> : KeyedCollection<Type, TItem>
>     {
> ```

```
    }
}
```

YAML:

```
uid : System.Collections.Generic.KeyedByTypeCollection`1
inheritance:
- System.Collections.ObjectModel.KeyedCollection{System.Type,{TItem}}
- System.Collections.ObjectModel.Collection{{TItem}}
- System.Object
```

4.3 Syntax

The syntax part for type contains declaration, and descriptions of type parameters for different languages. For delegates, it also contains descriptions of parameters and a return type.

5. Members

Members include fields, properties, methods, and events. They have the following properties: summary, remarks, exceptions, and syntax. The parents of members are types. Members never have children, and all parameter types or return types are spec id.

5.1 Constructors

The *ID* of a constructor is defined by `#ctor`, followed by the list of the *UIDs* of its parameter types: When a constructor does not have parameter, its *ID* **MUST NOT** end with parentheses.

The syntax part for constructors contains a special language declaration, and descriptions of parameters.

> Example 5.1 Constructor
>
> C#:
>
> ```
> namespace System
> {
>     public sealed class String
>     {
>         public String();
>         public String(char[] chars);
> ```

```
    }
}
```

YAML:

```
- uid: System.String.#ctor
  id: #ctor
  name.csharp: String()
  fullName.csharp: System.String.String()
- uid: System.String.#ctor(System.Char[])
  id: #ctor(System.Char[])
  name.csharp: String(Char[])
  fullName.csharp: System.String.String(System.Char[])
```

5.2 Methods

The *ID* of a method is defined by its name, followed by the list of the *UIDs* of its parameter types:

```
method_name(param1,param2,...)
```

When a method does not have parameter, its *ID* **MUST** end with parentheses.

The syntax part for method contains a special language declaration, and descriptions of type parameters for generic method, descriptions of parameters and return type.

Example 5.2 Method

C#:

```
namespace System
{
    public sealed class String
    {
        public String ToString();
        public String ToString(IFormatProvider provider);
    }
}
```

YAML:

```
- uid: System.String.ToString
  id: ToString
  name.csharp: ToString()
  fullName.csharp: System.String.ToString()
- uid: System.String.ToString(System.IFormatProvider)
  id: ToString(System.IFormatProvider)
  name.csharp: ToString(IFormatProvider)
  fullName.csharp: System.String.ToString(System.IFormatProvider)
```

5.2.1 Explicit Interface Implementation

The *ID* of an explicit interface implementation (EII) member **MUST** be prefixed by the *UID* of the interface it implements and replace `.` to `#`.

> Example 2.6 Explicit interface implementation (EII)
>
> C#:

```
namespace System
{
    using System.Collections;

    public sealed class String : IEnumerable
    {
        IEnumerator IEnumerable.GetEnumerator();
    }
}
```

> YAML:

```
- uid: "System.String.System#Collections#IEnumerable#GetEnumerator"
  id: "System#Collections#IEnumerable#GetEnumerator"
  name.csharp: IEnumerable.GetEnumerator()
  fullName.csharp: System.String.System.Collections.IEnumerable.GetEnumerator()
```

5.4 Operator Overloads

The *IDs* of operator overloads are same with the metadata name (for example, `op_Equality`). The names of operator overloads are similar to MSDN, just remove `op_` from the metadata name of the method. For instance, the name of the equals (`==`) operator is `Equality`.

Type conversion operator can be considered a special operator whose name is the UID of the target type, with one parameter of the source type. For example, an operator that converts from string to int should be `Explicit(System.String to System.Int32)`.

The syntax part for methods contains a special language declaration, descriptions of parameters and return type.

> Example 5.4 Operator overload

```
namespace System
{
    public struct Decimal
    {
        public static implicit operator Decimal(Char value);
    }

    public sealed class String
    {
        public static bool operator ==(String a, String b);
    }
}
```

YAML:

```
- uid: System.Decimal.op_Implicit(System.Char)~System.Decimal
  id: op_Implicit(System.Char)~System.Decimal
  name.csharp: Implicit(Char to Decimal)
  fullName.csharp: System.Decimal.Implicit(System.Char to System.Decimal)
- uid: System.String.op_Equality(System.String,System.String)
  id: op_Equality(System.String,System.String)
  name.csharp: Equality(String,String)
  fullName.csharp: System.String.Equality(System.String,System.String)
```

Please check overloadable operators for all overloadable operators.

5.5 Field, Property or Event

The *ID* of field, property or event is its name. The syntax part for field contains a special language declaration and descriptions of field type. For property, it contains a special language declaration, descriptions of parameters, and return type. For event, it contains a special language declaration and descriptions of event handler type.

> Example 5.5 Field, Property and Event

C#:

```
namespace System
{
    public sealed class String
    {
        public static readonly String Empty;
        public int Length { get; }
    }

    public static class Console
    {
        public static event ConsoleCancelEventHandler CancelKeyPress;
    }
}
```

YAML:

```
- uid: System.String.Empty
  id: Empty
  name.csharp: Empty
  fullName.csharp: System.String.Empty
- uid: System.String.Length
  id: Length
  name.csharp: Length
  fullName.csharp: System.String.Length
- uid: System.Console.CancelKeyPress
  id: CancelKeyPress
  name.csharp: CancelKeyPress
  fullName.csharp: System.Console.CancelKeyPress
```

5.6 Indexer

Indexer operator's name is metadata name, by default, it is `Item`, with brackets and parameters.

Example 5.6 Indexer

```
namespace System.Collections
{
    public interface IList
    {
        object this[int index] { get; set; }
```

```
    }
}
```

YAML:

```
- uid: "System.Collections.IList.Item[System.Int32]"
  id: "Item[System.Int32]"
  name.csharp: Item[Int32]
  fullName.csharp: System.Collections.IList.Item[System.Int32]
```

6. Generics

The *ID* of a generic type is its name with followed by `` `n ``, `n` and the count of generic type count, which is the same as the rule for document comment ID. For example, `Dictionary`2`.

The *ID* of a generic method uses postfix `` ``n ``, `n` is the count of in method parameters, for example, `System.Tuple.Create``1(``0)`.

Example 2.7 Generic

```
namespace System
{
    public static class Tuple
    {
        public static Tuple<T1> Create<T1>(T1 item1);
        public static Tuple<T1, T2> Create<T1, T2>(T1 item1, T2 item2);
    }
}
```

YAML:

```
- uid: System.Tuple.Create``1(``0)
  id: Create``1(``0)
  name.csharp:  Create<T1>(T1)
  fullName.csharp: System.Tuple.Create<T1>(T1)
- uid: System.Tuple.Create``2(``0,``1)
  id: Create``2(``0,``1)
  name.csharp:  Create<T1,T2>(T1,T2)
  fullName.csharp: System.Tuple.Create<T1,T2>(T1,T2)
```

7. Reference

The reference contains the following members: name, fullName, summary, isExternal, href, and more.

The *UID* in reference can be a *Spec Id*, then it contains one more member: spec. The *spec* in reference is very like a list of lightweight references, it describes how to compose the generic type in some special language.

> Example 7 *spec* for references
>
> YAML:

```
references:
- uid: System.Collections.Generic.Dictionary{System.String,System.Collections.Gener
  name.csharp: Dictionary<String, List<Int32>>
  fullName.csharp: System.Collections.Generic.Dictionary<System.String, System.Coll
  spec.csharp:
  - uid: System.Collections.Generic.Dictionary`2
    name: Dictionary
    fullName: System.Collections.Generic.Dictionary
    isExternal: true
  - name: <
    fullName: <
  - uid: System.String
    name: String
    fullName: System.String
    isExternal: true
  - name: ', '
    fullName: ', '
  - uid: System.Collections.Generic.List`1
    name: List
    fullName: System.Collections.Generic.List
    isExternal: true
  - name: <
    fullName: <
  - uid: System.Int32
    name: Int32
    fullName: System.Int32
    isExternal: true
  - name: '>'
    fullName: '>'
  - name: '>'
    fullName: '>'
```

Overwrite Files

Introduction

DocFX supports processing Markdown files, as well as structured data model in YAML or JSON format.

We call Markdown files *Conceptual File*s, and the structured data model files *Metadata File*s.

Current supported *Metadata File*s include:

1. YAML files presenting managed reference model following Metadata Format for .NET Languages.
2. Swagger JSON files presenting Swagger REST API model following Swagger Specification Version 2.0.

Inside DocFX, both *Conceptual File*s and *Metadata File*s are represented as *Model*s with different properties. Details on *Model* structure for these files are described in Data model inside DocFX section.

DocFX introduces the concept of *Overwrite File* to modify or add properties to *Model*s without changing the input *Conceptual File*s and *Metadata File*s.

The format of *Overwrite Files*

*Overwrite File*s are Markdown files with multiple *Overwrite Section*s starting with YAML header block. A valid YAML header for an *Overwrite Section MUST* take the form of valid YAML set between triple-dashed lines and start with property `uid`. Here is a basic example of an *Overwrite Section*:

```
---
uid: microsoft.com/docfx/Contacts
some_property: value
---
Further description for `microsoft.com/docfx/Contacts`
```

Each *Overwrite Section* is transformed to *Overwrite Model* inside DocFX. For the above example, the *Overwrite Model* represented in YAML format is:

```
uid: microsoft.com/docfx/Contacts
some_property: value
conceptual: <p><b>Content</b> in Markdown</p>
```

Anchor `*content`

`*content` is the keyword invented and used specifically in *Overwrite File*s to represent the Markdown content following YAML header. We leverage Anchors syntax in YAML specification for `*content`.

The value for `*content` is always transformed from Markdown content to HTML. When `*content` is not used, the Markdown content below YAML header will be set to `conceptual` property; When `*content` is used, the Markdown content below YAML header will no longer be set to `conceptual` property. With `*content`, we can easily add Markdown content to any properties.

```
---
uid: microsoft.com/docfx/Contacts
footer: *content
---
Footer for `microsoft.com/docfx/Contacts`
```

In the above example, the value for `*content` is `<p>Footer for <code>microsoft.com/docfx/Contacts</code></p>`, and the *Overwrite Model* represented in YAML format is:

```
uid: microsoft.com/docfx/Contacts
footer: <p>Footer for <code>microsoft.com/docfx/Contacts</code></p>
```

`uid` for an *Overwrite Model* stands for the Unique IDentifier of the *Model* it will overwrite. So it is allowed to have multiple *Overwrite Section*s with YAML Header containing the same `uid`. For one *Overwrite File*, the latter *Overwrite Section* overwrites the former one with the same `uid`. For different *Overwrite File*s, the order of overwrite is **Undetermined**. So it is suggested to have *Overwrite Sections* with the same `uid` in the same *Overwrite File*.

When processing *Conceptual File*s and *Metadata File*s, *Overwrite Model*s with the same `uid` are applied to the processed *Model*s. Different *Model*s have different overwrite principles, Overwrite principles section describes the them in detail.

Apply *Overwrite File*s

Inside `docfx.json`, `overwrite` is used to specify the *Overwrite File*s.

Overwrite principles

As a general principle, `uid` is always the key that an *Overwrite Model* find the *Model* it is going to overwrite. So a *Model* with no `uid` defined will never get overwritten.

Different types of files produce different *Model*s. The quickest way to get an idea of what the *Model* looks like is to run:

```
docfx build --exportRawModel
```

`--exportRawModel` exports *Model* in JSON format with `.raw.json` extension.

The basic principle of *Overwrite Model* is:

1. It keeps the same data structure as the *Model* it is going to overwrite
2. If the property is defined in *Model*, please refer Data model inside DocFX for the specific overwrite behavior for a specific property.
3. If the property is not defined in *Model*, it is added to *Model*

Data model inside DocFX

Managed reference model

Key	**Type**	**Overwrite behavior**
uid	uid	Merge key.
assemblies	string[]	Ignore.
attributes	Attribute[]	Ignore.
children	uid[]	Ignore.
documentation	Source	Merge.
example	string[]	Replace.
exceptions	Exception[]	Merge keyed list.
fullName	string	Replace.
fullName.	string	Replace.
id	string	Replace.
implements	uid[]	Ignore.

Key	Type	Overwrite behavior
inheritance	uid[]	Ignore.
inheritedMembers	uid[]	Ignore.
isEii	boolean	Replace.
isExtensionMethod	boolean	Replace.
langs	string[]	Replace.
modifiers.	string[]	Ignore.
name	string	Replace.
name.	string	Replace.
namespace	uid	Replace.
overridden	uid	Replace.
parent	uid	Replace.
platform	string[]	Replace.
remarks	markdown	Replace.
see	LinkInfo[]	Merge keyed list.
seealso	LinkInfo[]	Merge keyed list.
source	Source	Merge.
syntax	Syntax	Merge.
summary	markdown	Replace.
type	string	Replace.

Source

Property	Type	Overwrite behavior
base	string	Replace.

Property	Type	Overwrite behavior
content	string	Replace.
endLine	integer	Replace.
id	string	Replace.
isExternal	boolean	Replace.
href	string	Replace.
path	string	Replace.
remote	GitSource	Merge.
startLine	integer	Replace.

GitSource

Property	Type	Overwrite behavior
path	string	Replace.
branch	string	Replace.
repo	url	Replace.
commit	Commit	Merge.
key	string	Replace.

Commit

Property	Type	Overwrite behavior
committer	User	Replace.
author	User	Replace.
id	string	Replace.
message	string	Replace.

User

Property	Type	Overwrite behavior
name	string	Replace.
email	string	Replace.
date	datetime	Replace.

Exception

Property	Type	Overwrite behavior
type	uid	Merge key.
description	markdown	Replace.
commentId	string	Ignore.

LinkInfo

Property	Type	Overwrite behavior
linkId	uid or href	Merge key.
altText	markdown	Replace.
commentId	string	Ignore.
linkType	enum(`CRef` or `HRef`)	Ignore.

Syntax

Property	Type	Overwrite behavior
content	string	Replace.
content.	string	Replace.
parameters	Parameter[]	Merge keyed list.
typeParameters	Parameter[]	Merge keyed list.
return	Parameter	Merge.

Parameter

Property	Type	Overwrite behavior
id	string	Merge key.
description	markdown	Replace.
attributes	Attribute[]	Ignore.
type	uid	Replace.

Attribute

Property	Type	Overwrite behavior
arguments	Argument[]	Ignore.
ctor	uid	Ignore.
namedArguments	NamedArgument[]	Ignore.
type	uid	Ignore.

Argument

Property	Type	Overwrite behavior
type	uid	Ignore.
value	object	Ignore.

NamedArgument

Property	Type	Overwrite behavior
name	string	Ignore.
type	string	Ignore.
value	object	Ignore.

REST API model

Key	Type	Overwrite behavior
children	REST API item model	Overwrite when *uid* of the item model matches
summary	string	Overwrite
description	string	Overwrite

REST API item model

Key	Type	Overwrite behavior
uid	string	Key

Conceptual model

Key	Type	Overwrite behavior
title	string	Overwrite
rawTitle	string	Overwrite
conceptual	string	Overwrite

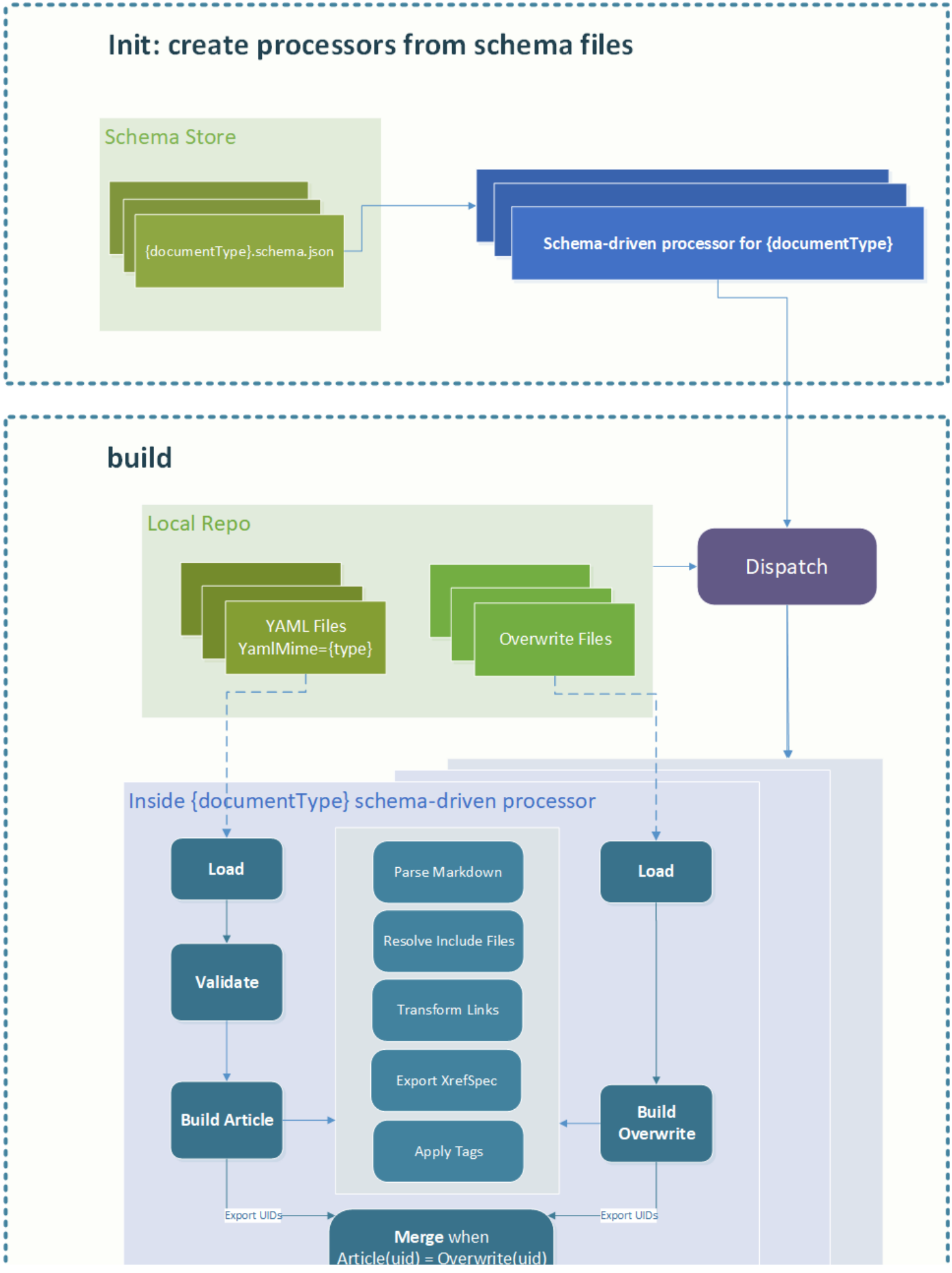
Schema-driven Document Processor(SDP) Design Spec

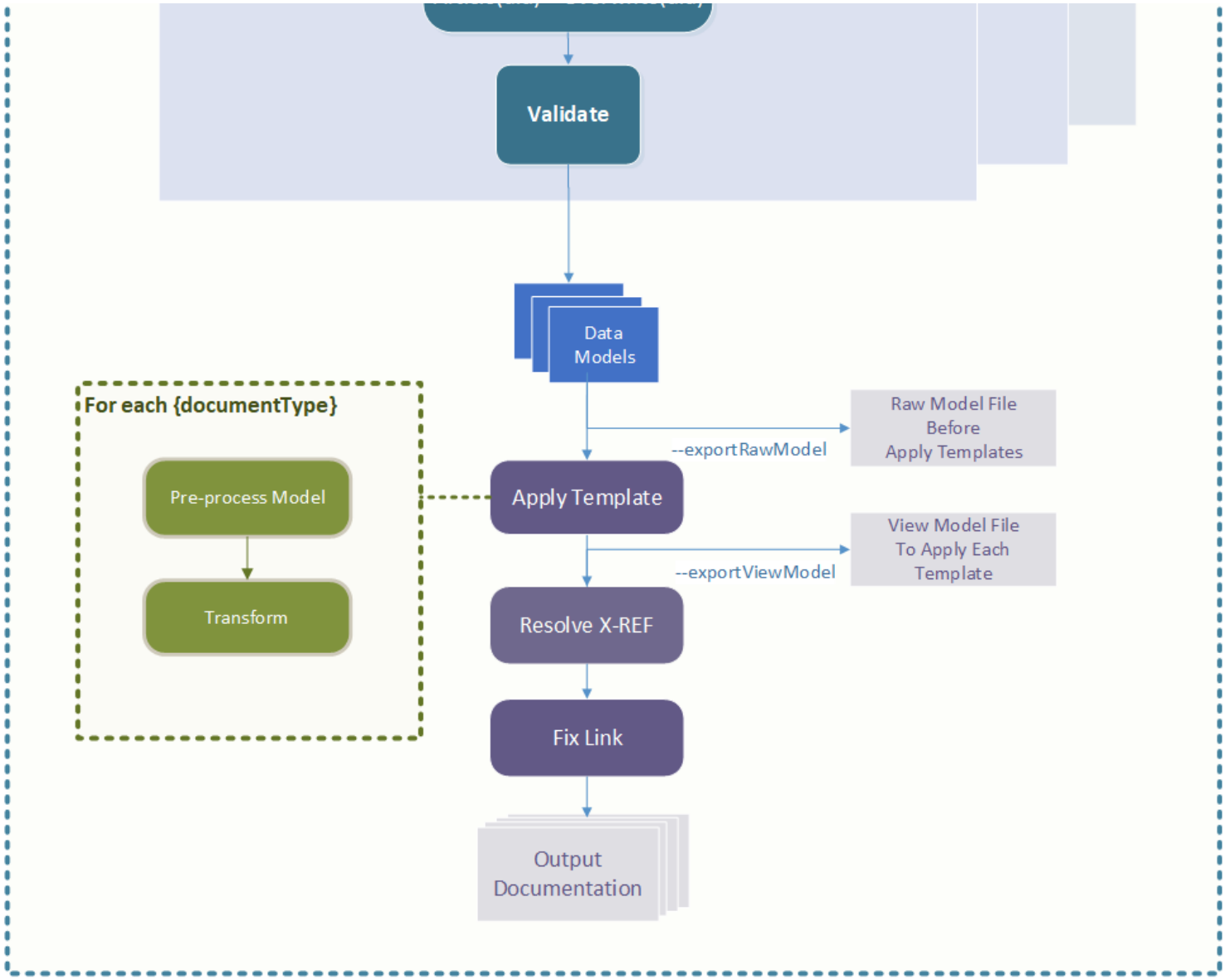
1. Overview

DocFX supports different document processors to handle different kinds of input. With a new data model introduced in, a new document processor is required to support that model, even most of the code logic is the same for these processors. With this situation considered, a Schema-driven Document Processor (abbreviated to *SDP* below) is introduced to simplify the process. Togethor with a well defined DocFX Document Schema, SDP is able to *validate* and *process* a new data model with no extra effort needed.

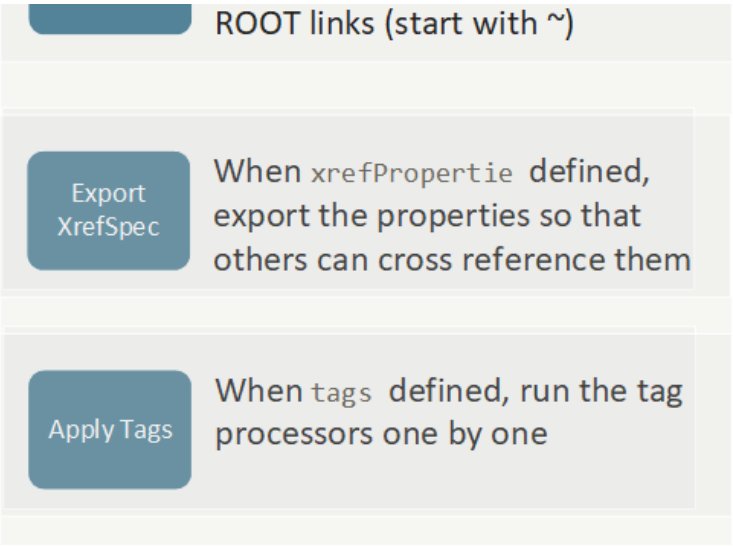
2. Workflow

The workflow for SDP is illustrated below. In general, the schema file, with suggested naming convention, has `documentType` in its name, as `{documentType}`.schema.json (When `title` is defined in the schema file, `title` is considered as the `documentType` for this schema). `docfx` loads the schema files from `schemas` subfolder in `template` folder, and creates processors for these schema files with per schema file per processor. With data models are processed, `docfx` applies templates for that `documentType` to these data model, as details illustrated in Template Introduction and generates output documentation.


Init: create processors from schema files
Schema Store
{documentType}.schema.json
Schema-driven processor for {documentType}
build
Local Repo
YAML Files
YamlMime={type}
Overwrite Files
Dispatch
Inside {documentType} schema-driven processor
Load
Validate
Build Article
Parse Markdown
Resolve Include Files
Transform Links
Export XrefSpec
Apply Tags
Load
Build Overwrite
Export UIDs
Export UIDs
Merge when
Article(uid) = Overwrite(uid)



Parse Markdown	When `contentType==markdown`, transform the content to html
Resolve Include Files	When `reference==file`, replace value with file content
Transform Links	When `contentType==href`, transform the link to URL-TO-
Resolve X-REF	Resolve cross reference with **xref** html tag
Fix Link	Transform all URL-TO-ROOT links (start with ~) to links relative to current file


ROOT links (start with ~)
Export XrefSpec
When xrefPropertie defined, export the properties so that others can cross reference them
Apply Tags
When tags defined, run the tag processors one by one

DocFX Document Schema v1.0 Specification

1. Introduction

DocFX supports different document processors to handle different kinds of input. For now, if the data model changes a bit, a new document processor is needed, even most of the work in processors are the same.

DocFX Document Schema (abbreviated to *THIS schema* below) is introduced to address this problem. This schema is a JSON media type for defining the structure of a DocFX document. This schema is intended to **annotate**, **validate** and **interpret** the document data.

2. Conventions and Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

3. Overview

DocFX Document Schema is in JSON format. It borrows most syntax from JSON Schema, while it also introduces some other syntax to manipulate the data.

3.1 Annotation

THIS schema is a JSON based format for the structure of a DocFX document.

3.2 Validation

JSON schema validation already defines many keywords. This schema starts from supporting limited keyword like `type`, `properties`.

3.3 Interpretation

Besides annotate and validate the input document model, *THIS schema* also defines multiple interpretations for each property of the document model. For example, a property named `summary` contains value in Markdown format, *THIS schema* can define a `markup` interpretation for the `summary` property, so that the property can be marked using DFM syntax.

4. General Considerations

- *THIS schema* leverages JSON schema definition, that is to say, keywords defined in JSON schema keeps its meaning in *THIS schema* when it is supported by *THIS schema*.

5. Detailed Specification

Format

The files describing DocFX document model in accordance with the DocFX document schema specification are represented as JSON objects and conform to the JSON standards. YAML, being a superset of JSON, can be used as well to represent a DocFX document schema specification file.

All field names in the specification are **case sensitive**.

This schema exposes two types of fields. Fixed fields, which have a declared name, and Patterned fields, which declare a regex pattern for the field name. Patterned fields can have multiple occurrences as long as each has a unique name.

By convention, the schema file is suffixed with `.schema.json`.

Data Types

Primitive data types in *THIS schema* are based on JSON schema Draft 6 4.2 Instance

Schema

For a given field, `*` as the starting character in *Description* cell stands for **required**.

Schema Object

This is the root document object for *THIS schema*.

Fixed Field

Field Name	Type	Description
$schema	string	*The version of the schema specification, for example, `https://dotnet.github.io/docfx/schemas/v1.0/schema.json#`.
version	string	*The version of current schema object.
id	string	It is best practice to include an `id` property as an unique identifier for each schema.
title	string	The title of current schema, `LandingPage`, for example. In DocFX, this value can be used to determine what kind of documents apply to this schema, If not specified, file name

Field Name	Type	Description
		before `schema.json` of this schema is used. Note that `.` is not allowed.
description	string	A short description of current schema.
type	string	*The type of the root document model MUST be `object`.
properties	Property Definitions Object	An object to hold the schema of all the properties.
metadata	string	In `json-pointer` format as defined in http://json-schema.org/latest/json-schema-validation.html#rfc.section.8.3.9. The format for JSON pointer is defined by https://tools.ietf.org/html/rfc6901, referencing to the metadata object. Metadata object is the object to define the metadata for current document, and can be also set through `globalMetadata` or `fileMetadata` in DocFX. The default value for metadata is empty which stands for the root object.

Patterned Field

Field Name	Type	Description
^x-	Any	Allows extensions to *THIS schema*. The field name MUST begin with x-, for example, x-internal-id. The value can be null, a primitive, an array or an object.

Property Definitions Object

It is an object where each key is the name of a property and each value is a schema to describe that property.

Patterned Field

Field Name	Type	Description
{name}	Property Object	The schema object for the `{name}` property

Property Object

An object to describe the schema of the value of the property.

Fixed Field

Field Name	Type	Description
title	string	The title of the property.
description	string	A lengthy explanation about the purpose of the data described by the schema.
default	what `type` defined	The default value for current field.
type	string	The type of the root document model. Refer to type keyword for detailed description.
properties	Property Definitions Object	An object to hold the schema of all the properties if `type` for the model is `object`. Omitting this keyword has the same behavior as an empty object.
items	Property Object	An object to hold the schema of the items if `type` for the model is `array`. Omitting this keyword has the same behavior as an empty schema.
reference	string	Defines whether current property is a reference to the actual value of the property. Refer to reference for detailed explanation.
contentType	string	Defines the content type of the property. Refer to contentType for detailed explanation.
tags	array	Defines the tags of the property. Refer to tags for detailed explanation.
mergeType	string	Defines how to merge the property. Omitting this keyword has the same behavior as `merge`. Refer to mergeType for detailed explanation.
xrefProperties	array	Defines the properties of current object when it is cross referenced by others. Each item is the name of the property in the instance. Refer to xrefProperties for detailed description of how to leverage this property.

Patterned Field

Field Name	Type	Description
^x-	Any	Allows extensions to *THIS schema*. The field name MUST begin with x-, for example, x-internal-id. The value can be null, a primitive, an array or an object.

6. Keywords in detail

6.1 type

Same as in JSON schema: http://json-schema.org/latest/json-schema-validation.html#rfc.section.6.25

> The value of this keyword MUST be either a string or an array. If it is an array, elements of the array MUST be strings and MUST be unique.
>
> String values MUST be one of the six primitive types ("null", "boolean", "object", "array", "number", or "string"), or "integer" which matches any number with a zero fractional part.
>
> An instance validates if and only if the instance is in any of the sets listed for this keyword.

6.2 reference

It defines whether current property is a reference to the actual value of the property. The values MUST be one of the following:

Value	Description
`none`	It means the property is not a reference.
`file`	It means current property stands for a file path that contains content to be included.

6.3 contentType

It defines how applications interpret the property. If not defined, the behavior is similar to `default` value. The values MUST be one of the following:

Value	Description
`default`	It means that no interpretion will be done to the property.
`uid`	`type` MUST be `string`. With this value, the property name MUST be `uid`. It means the property defines a unique identifier inside current document model.
`href`	`type` MUST be `string`. It means the property defines a file link inside current document model. Application CAN help to validate if the linked file exists, and update the file link if the linked file changes its output path.
`xref`	`type` MUST be `string`. It means the property defines a UID link inside current document model. Application CAN help to validate if the linked UID exists, and resolve the UID link to the corresponding file output link.
`file`	`type` MUST be `string`. It means the property defines a file path inside current document model. Application CAN help to validate if the linked file exists, and resolve the path to the corresponding file output path. The difference between `file` and `href` is that `href` is always URL encoded while `file` is not.
`markdown`	`type` MUST be `string`. It means the property is in DocFX flavored Markdown syntax. Application CAN help to transform it into HTML format.

6.4 tags

The value of this keyword MUST be an `array`, elements of the array MUST be strings and MUST be unique. It provides hints for applications to decide how to interpret the property, for example, `localizable` tag can help Localization team to interpret the property as *localizable*.

6.5 mergeType

The value of this keyword MUST be a string. It specifies how to merge two values of the given property. One use scenario is how DocFX uses the overwrite files to overwrite the existing values. In the below table, we use `source` and `target` to stands for the two values for merging.

The value MUST be one of the following:

Value	Description
`key`	If `key` for `source` equals to the one for `target`, these two values are ready to merge.

Value	Description
`merge`	The default behavior. For `array`, items in the list are merged by `key` for the item. For `string` or any value type, `target` replaces `source`. For `object`, merge each property along with its own `merge` value.
`replace`	`target` replaces `source`.
`ignore`	`source` is not allowed to be merged.

6.6 xrefProperties

The value of this keyword MUST be an array of `string`. Each `string` value is the property name of current object that will be exported to be cross referenced by others. To leverage this feature, a new `xref` syntax with `template` attribute is support:

```
<xref uid="{uid}" template="{path_of_partial_template}" />
```

For the parital template, the input model is the object containing properties `xrefProperties` defines.

For example, in the sample schema defined by 7. Samples, `"xrefProperties": [ "title", "description" ],`, `title` and `description` are `xrefProperties` for uid `webapp`. A partial template to render this xref, for example, named `partials/overview.tmpl`, looks like:

```
{{title}}: {{{description}}}
```

When someone references this uid using `<xref uid="webapp" template="partials/overview.tmpl"`, `docfx` expand this `xref` into the following html:

```
Web Apps Documentation: <p>This is description</p>
```

In this way, users can not only *cross reference* others to get the target url, but also *cross reference* other properties as they like.

A common usage of this is the **Namespace** page in ManagedReference. The **Namespace** page shows a table of its **Classes** with the `summary` of the **Class**, with the help of `xrefProperties`, the source of truth `summary` is always from **Class**. For the **Namespace** page, it can, for example:

1. Define a `class.tr.tmpl` template: `<tr><td>{{name}}</td><td>{{{summary}}}</td></tr>`
2. The namespace `namespace.tmpl` template, use `xref` to render its children classes:

```
  {{#children}}
    <xref uid="{{uid}}" template="class.tr.tmpl" />
  {{/children}}
```

7. Samples

Here's an sample of the schema. Assume we have the following YAML file:

```
### YamlMime:LandingPage
title: Web Apps Documentation
description: This is description
uid: webapp
metadata:
  title: Azure Web Apps Documentation - Tutorials, API Reference
  meta.description: Learn how to use App Service Web Apps to build and host websites ar
  services: app-service
  author: apexprodleads
  manager: carolz
  ms.service: app-service
  ms.tgt_pltfrm: na
  ms.devlang: na
  ms.topic: landing-page
  ms.date: 01/23/2017
  ms.author: carolz
sections:
- title: 5-Minute Quickstarts
  children:
  - text: .NET
    href: app-service-web-get-started-dotnet.md
  - text: Node.js
    href: app-service-web-get-started-nodejs.md
  - text: PHP
    href: app-service-web-get-started-php.md
  - text: Java
    href: app-service-web-get-started-java.md
  - text: Python
    href: app-service-web-get-started-python.md
  - text: HTML
    href: app-service-web-get-started-html.md
- title: Step-by-Step Tutorials
  children:
  - content: "Create an application using [.NET with Azure SQL DB](app-service-web-tutc
  - content: "[Map an existing custom domain to your application](app-service-web-tutor
  - content: "[Bind an existing SSL certificate to your application](app-service-web-tu
```

In this sample, we want to use the JSON schema to describe the overall model structure. Further more, the `href` is a file link. It need to be resolved from the relative path to the final href. The `content` property need to be marked up as a Markdown string. The `metadata` need to be tagged for further custom operations. We want to use `section`'s `title` as the key for overwrite `section` array.

Here's the schema to describe these operations:

```
{
    "$schema": "https://dotnet.github.io/docfx/schemas/v1.0/schema.json#",
    "version": "1.0.0",
    "id": "https://github.com/dotnet/docfx/schemas/landingpage.schema.json",
    "title": "LandingPage",
    "description": "The schema for landing page",
    "type": "object",
    "xrefProperties": [ "title", "description" ],
    "properties": {
        "metadata": {
            "type": "object",
            "tags": [ "metadata" ]
        },
        "uid": {
            "type": "string",
            "contentType": "uid"
        },
        "sections": {
            "type": "array",
            "items": {
                "type": "object",
                "properties": {
                    "children": {
                        "type": "array",
                        "items": {
                            "type": "object",
                            "properties": {
                                "href": {
                                    "type": "string",
                                    "contentType": "href"
                                },
                                "text": {
                                    "type": "string",
                                    "tags": [ "localizable" ]
                                },
                                "content": {
                                    "type": "string",
```

```
                            "contentType": "markdown"
                        }
                    }
                }
            },
            "title": {
                "type": "string",
                "mergeType": "key"
            }
        }
    },
    "title": {
        "type": "string"
    }
}
}
```

8. Q & A

1. DocFX fills `_global` metadata into the processed data model, should the schema reflect this behavior?
   - Decision: *NOT* include, this schema is for **input model**, use another schema for output model.
2. Is it necessary to prefix `d-` to every field that DocFX introduces in?
   - If keep `d-`
     - Pros:
       1. `d-` makes it straightforward that these keywords are introduced by DocFX
       2. Keywords DocFX introduces in will never duplicate with the one preserved by JSON schema
     - Cons:
       1. `d-` prefix provides a hint that these keywords are not *first class* keywords
       2. Little chance that keywords DocFX defines duplicate with what JSON schema defines, after all, JSON schema defines a finite set of reserved keywords.
       3. For exampleSwagger spec is also based on JSON schema and the fields it introduces in has no prefix.
   - Decision: *Remove* `d-` prefix.

How-to: Build your own type of documentation with a custom plug-in

In this topic we will create a plug-in to convert some simple rich text format files to html documents.

Goal and limitation

1. In scope:
   1. Our input will be a set of rtf files with `.rtf` as the file extension name.
   2. The rtf files will be built as html document.
2. Out of scope:
   1. Picture or other object in rtf files.
   2. Hyperlink in rtf files. (in the advanced tutorial, we will describe how to support hyperlinks in a custom plugin.)
   3. Metadata and title.

Preparation

1. Create a new C# class library targeting `net6.0` or later.
2. Add NuGet package reference to `System.Composition`, `Docfx.Plugins` and `Docfx.Common`.
3. Add a project for converting rtf to html: Clone project MarkupConverter, and reference it.
4. Copy the code file `CSharp/parallel/ParallelExtensionsExtras/TaskSchedulers/StaTaskScheduler.cs` from DotNet Samples

Create a document processor

Responsibility of the document processor

- Declare which file can be handled.
- Load from the file to the object model.
- Provide build steps.
- Report document type, file links and xref links in document.
- Update references.

Create our RtfDocumentProcessor

1. Create a new class (RtfDocumentProcessor.cs) with the following code:

```
[Export(typeof(IDocumentProcessor))]
public class RtfDocumentProcessor : IDocumentProcessor
{
    // todo : implements IDocumentProcessor.
}
```

2. Declare that we can handle the `.rtf` file:

```
public ProcessingPriority GetProcessingPriority(FileAndType file)
{
    if (file.Type == DocumentType.Article &&
        ".rtf".Equals(Path.GetExtension(file.File), StringComparison.OrdinalIgnoreC
    {
        return ProcessingPriority.Normal;
    }
    return ProcessingPriority.NotSupported;
}
```

   Here we declare this processor can handle any `.rtf` file in the article category with normal priority. When two or more processors compete for the same file, DocFX will give it to the higher priority one. *Unexpected*: two or more processor declare for the same file with same priority.

3. Load our rtf file by reading all text:

```
public FileModel Load(FileAndType file, ImmutableDictionary<string, object> metadat
{
    var content = new Dictionary<string, object>
    {
        ["conceptual"] = File.ReadAllText(Path.Combine(file.BaseDir, file.File)),
        ["type"] = "Conceptual",
        ["path"] = file.File,
    };
    var localPathFromRoot = PathUtility.MakeRelativePath(EnvironmentContext.BaseDir

    return new FileModel(file, content)
    {
        LocalPathFromRoot = localPathFromRoot,
    };
}
```

   We use `Dictionary<string, object>` as the data model, similar to how ConceptualDocumentProcessor stores the content of markdown files.

4. Implement `Save` method as follows:

```
public SaveResult Save(FileModel model)
{
    return new SaveResult
    {
        DocumentType = "Conceptual",
        FileWithoutExtension = Path.ChangeExtension(model.File, null),
    };
}
```

5. `BuildSteps` property can provide several build steps for the model. We suggest implementing this in the following manner:

```
[ImportMany(nameof(RtfDocumentProcessor))]
public IEnumerable<IDocumentBuildStep> BuildSteps { get; set; }
```

6. `Name` property is used to display in the log, so give any constant string you like. e.g.:

```
public string Name => nameof(RtfDocumentProcessor);
```

7. Since we don't support hyperlink, keep the `UpdateHref` method empty.

```
public void UpdateHref(FileModel model, IDocumentBuildContext context)
{
}
```

View the final RtfDocumentProcessor.cs

Create a document build step

Responsibility of the build step

- Reconstruct documents via the `Prebuild` method, e.g.: remove some document according to a certain rule.
- Transform document content via `Build` method, e.g.: transform rtf content to html content.
- Transform more content required by all document processed via the `PostBuild` method, e.g.: extract the link text from the title of another document.

- About build order:

  1. For all documents in one processor always `Prebuild` -> `Build` -> `Postbuild`.
  2. For all documents in one processor always invoke `Prebuild` by `BuildOrder`.
  3. For each document in one processor always invoke `Build` by `BuildOrder`.
  4. For all documents in one processor always invoke `Postbuild` by `BuildOrder`.

  e.g.: Document processor *X* has two steps: A (with BuildOrder=1), B (with BuildOrder=2). When *X* is handling documents [D1, D2, D3], the invoke order is as follows:

```
A.Prebuild([D1, D2, D3]) returns [D1, D2, D3]
B.Prebuild([D1, D2, D3]) returns [D1, D2, D3]
Parallel(
  A.Build(D1) -> B.Build(D1),
  A.Build(D2) -> B.Build(D2),
  A.Build(D3) -> B.Build(D3)
)
A.Postbuild([D1, D2, D3])
B.Postbuild([D1, D2, D3])
```

Create our RtfBuildStep:

1. Create a new class (RtfBuildStep.cs), and declare it is a build step for `RtfDocumentProcessor`:

```
[Export(nameof(RtfDocumentProcessor), typeof(IDocumentBuildStep))]
public class RtfBuildStep : IDocumentBuildStep
{
    // todo : implements IDocumentBuildStep.
}
```

2. In the `Build` method, convert rtf to html:

```
private readonly TaskFactory _taskFactory = new TaskFactory(new StaTaskScheduler(1)

public void Build(FileModel model, IHostService host)
{
    string content = (string)((Dictionary<string, object>)model.Content)["conceptua
    content = _taskFactory.StartNew(() => RtfToHtmlConverter.ConvertRtfToHtml(conte
    ((Dictionary<string, object>)model.Content)["conceptual"] = content;
}
```

3. Implement other methods:

```
public int BuildOrder => 0;

public string Name => nameof(RtfBuildStep);

public void Postbuild(ImmutableList<FileModel> models, IHostService host)
{
}

public IEnumerable<FileModel> Prebuild(ImmutableList<FileModel> models, IHostServic
{
    return models;
}
```

View the final RtfBuildStep.cs

Enable plug-in

1. Build our project.
2. Copy the output dll files to:
   - Global: the Docfx executable directory.
   - Non-global: a folder you create with the name `plugins` under a template folder. Then run `DocFX build` command with parameter `-t {template}`.

     Hint: DocFX can merge templates so create a template that only contains the `plugins` folder, then run the command `DocFX build` with parameter `-t {templateForRender},{templateForPlugins}`.

Build document

1. Run command `DocFX init` and set the source article with `**.rtf`.
2. Run command `DocFX build`.

How-to: Add a customized post-processor

We provide the ability to process output files by adding a customized post-processor. In DocFX, the index file for full-text-search is generated by one post-processor named `ExtractSearchIndex`. In this topic, we will show how to add a customized post-processor.

Step0: Preparation

- Create a new C# class library project in `Visual Studio`.
- Add nuget packages:
  - `System.Collections.Immutable` with version 1.3.1
  - `Microsoft.Composition` with version 1.0.31
- Add `Docfx.Plugins` If you are building DocFX from source code, add this reference to the project, otherwise add the nuget package `Docfx.Plugins` with the same version as DocFX.

Step1: Create a new class (MyProcessor.cs) with the following code:

```
[Export(nameof(MyProcessor), typeof(IPostProcessor))]
public class MyProcessor : IPostProcessor
{
    // TODO: implements IPostProcessor
}
```

Step2: Update global metadata

```
public ImmutableDictionary<string, object> PrepareMetadata(ImmutableDictionary<string,
{
    // TODO: add/remove/update property from global metadata
    return metadata;
}
```

In this method, we can update the global metadata before building all the files declared in `docfx.json`. Otherwise, you can just return the metadata from parameters if you don't need to change global metadata.

Using `ExtractSearchIndex` for example, we add `"_enableSearch": true` in global metadata. The default template would then know it should load a search box in the navbar.

Step3: Process all the files generated by DocFX

```
    public Manifest Process(Manifest manifest, string outputFolder)
    {
        // TODO: add/remove/update all the files included in manifest
        return manifest;
    }
```

Input for the method `manifest` contains a list of all files to process, and `outputFolder` specifies the output folder where our static website will be placed. We can implement customized operations here to process all files generated by DocFX.

> **NOTE**
>
> Post-processor aims to process the output files, so the `FileModel` can't be accessed in this phase. If some metadata is needed here, an option is to save it in `FileModel.ManifestProperties` in build phase, then access it through `ManifestItem.Metadata`. Another option is to save it somewhere in output files, like HTML's `<meta>` Tag.

Using `ExtractSearchIndex` for example again, we traverse all HTML files, extract key words from these HTML files and save a file named `index.json` under the `outputFolder`. Finally we return the manifest which is not modified.

Step4: Build your project and copy the output dll files to:

- Global: the folder with name `Plugins` under the folder containing the Docfx executable
- Non-global: the folder with name `Plugins` under a template folder, then run `DocFX build` command with parameter `-t {template}`.

  Hint: DocFX can merge templates, so we can specify multiple template folders as `DocFX build -t {templateForRender},{templateForPlugins}`. Each of the template folders should have a subfolder named `Plugins` with exported assemblies.

Step5: Add your post processor in `docfx.json`

In this step, we need to enable the processor by adding its name in `docfx.json`. Here is an example:

```
{
  "build": {
```

```
    ...
    "postProcessors": ["OutputPDF", "BeautifyHTML", "OutputPDF"]
  }
}
```

As you can see, the `postProcessors` is an array, which means it could have multiple processors. It needs to be pointed out that the order of `postProcessors` written in `docfx.json` is also the order to process output files. In the above example, DocFX will run `OutputPDF` first, then `BeautifyHTML`, and then `OutputPDF` again.

If you want to enable the post processors without changing `docfx.json`, you can use the build command option like `docfx build --postProcessors=OutputPDF,BeautifyHTML,OutputPDF`.

One more thing need to be noted: the build command option `postProcessors` would override the corresponding configuration in `docfx.json`.

🔧 Advanced: Support Hyperlink

In this topic, we will support hyperlinking in rtf files.

Create a hyperlink in the rtf file:

1. Open `foo.rtf` by `Word`.
2. Add a hyperlink in content
3. Set the link target to an existing `bar.rtf`
4. Save the document.

About link

An author can write any valid hyperlink in the document, and then needs to run `DocFX build` to update file links.

What is file link:

1. The hyperlink must be a relative path and not rooted.
   - valid: `foo\bar.rtf`, `../foobar.rtf`
   - invalid: `/foo.rtf`, `c:\foo\bar.rtf`, `http://foo.bar/`, `mailto:foo@bar.foobar`
2. The file must exist.

Why update file link:

The story is:

1. In `foo.rtf`, it has a file link to `bar.rtf`.
2. In document build, `bar.rtf` generates a file with the name `bar.html`.
3. But in `foo.rtf`, the link target is still `bar.rtf`, thus in the output folder we cannot find this file and we will get a broken link.
4. To resolve the broken link, we need to update the link target from `bar.rtf` to `bar.html`.

File link is a relative path, but we cannot track the relative path easily. So we track the *normalized file path* instead.

What is a *normalized file path*:

1. It always starts from the working folder (the folder that contains `docfx.json`), and we write it as `~/`.
2. No `../` or `./` or `//`
3. Replace `\` with `/`.
4. No url encoding. The path must be same as it in the file system.
5. No anchor.

Finally, a valid *normalized file path* looks like: `~/foo/bar.rtf`.

- Pros
  - Same form in different documents when the target is the same file.

    When file structure is:

    ```
    z:\a\b\foo.rtf
    z:\a\b\c\bar.rtf
    z:\a\b\c\foobar.rtf
    ```

    Link target `c/foobar.rtf` in `foo.rtf` and link target `foobar.rtf` in `bar.rtf` is the same file. When the working folder is `z:\a\`, the link target is always `~/b/c/foobar.rtf`.
  - Avoids differences in style when referring to the same file.

    For example, the following hyperlinks target the same file: `a/foo.rtf`, `./a/foo.rtf`, `a/b/../foo.rtf`, `a//foo.rtf`, `a\foo.rtf`
- Cons
  - A folder with the name `~` is not supported.

Prepare

1. Open the rtf plug-in library project in `Visual Studio`.
2. Add nuget packages:
   for plug-in: `Docfx.Utility`
3. Add framework assembly reference: `System.Core`, `System.Web`, `System.Xml.Linq`

Update rtf document processor

1. Following the rules for hyperlink, add a `FixLink` help method:

```
private static void FixLink(XAttribute link, RelativePath filePath, HashSet<string>
{
    string linkFile;
    string anchor = null;
    if (PathUtility.IsRelativePath(link.Value))
    {
        var index = link.Value.IndexOf('#');
        if (index == -1)
```

```
            {
                linkFile = link.Value;
            }
            else if (index == 0)
            {
                return;
            }
            else
            {
                linkFile = link.Value.Remove(index);
                anchor = link.Value.Substring(index);
            }
            var path = filePath + (RelativePath)linkFile;
            var file = (string)path.GetPathFromWorkingFolder();
            link.Value = file + anchor;
            linkToFiles.Add(HttpUtility.UrlDecode(file));
        }
    }
```

`RelativePath` helps us generate the links correctly.

2. Then add `CollectLinksAndFixDocument` method:

```
    private static HashSet<string> CollectLinksAndFixDocument(FileModel model)
    {
        string content = (string)((Dictionary<string, object>)model.Content)["conceptua
        var doc = XDocument.Parse(content);
        var links =
            from attr in doc.Descendants().Attributes()
            where "href".Equals(attr.Name.LocalName, StringComparison.OrdinalIgnoreCase
            select attr;
        var path = (RelativePath)model.File;
        var linkToFiles = new HashSet<string>();
        foreach (var link in links)
        {
            FixLink(link, path, linkToFiles);
        }
        using (var sw = new StringWriter())
        {
            doc.Save(sw);
            ((Dictionary<string, object>)model.Content)["conceptual"] = sw.ToString();
        }
        return linkToFiles;
    }
```

3. Modify `Save` method with report links:

```
public SaveResult Save(FileModel model)
{
    HashSet<string> linkToFiles = CollectLinksAndFixDocument(model);

    return new SaveResult
    {
        DocumentType = "Conceptual",
        ModelFile = model.File,
        LinkToFiles = linkToFiles.ToImmutableArray(),
    };
}
```

View final RtfDocumentProcessor.cs

Test and verify

1. Build project.
2. Copy dll to `Plugins` folder.
3. Modify rtf file, create hyperlink, link to another rtf file, and save.
4. Build with command `DocFX build`.
5. Verify output html file.