Jump start

Set up

You can install Projbook to any project but we recommand creating a dedicated documentation project. It will allow you to centralize all the documentation without contaminating the code projects.

First of all, install Projbook using nuget in the project you want the documentation generated from. Alternatively, you can install Projbook.Core if you don't have any PDF needs.

It will create a default configuration `projbook.json` from where you can configure your documentation project, define your templates and your page content:

- Projbook.Example
    - projbook.json

```
{
  "title": "Projbook landing page",
  "description": "Everything you ever dreamed to know!",
  "template": "index-template.html",
  "output": "index.html",
  "icon": "https://raw.githubusercontent.com/defrancea/Projbook/master/resources/Projbook.png",
  "configurations": [
    {
      "title": "Sample document",
      "description": "Description",
      "icon": "https://raw.githubusercontent.com/defrancea/Projbook/master/resources/Projbook.png",
      "template-html": "template.html",
      "template-pdf": "template-pdf.html",
      "output-html": "documentation.html",
      "output-pdf": "documentation-pdf-input.html",
      "section-title-base": 1,
      "pages": [
        {
          "title": "First page",
          "path": "Page/page1.md"
        },
        {
          "title": "Second page",
          "path": "Page/page2.md"
        }
      ]
    }
  ]
}
```

You can also find some sample files under `Page`. These are going to be the source of the documentation, it's where you'll spend most of your time writing your documentation referencing your actual code source. The syntax of these is markdown and we'll detail later how to reference and extract your source code.

- Projbook.Example

Installing a visual studio Markdown Editor will make the markdown writing easier.

Default templates can edited in order to customize your rendering.

- Projbook.Example
  - index-template.html
  - template.html
  - template-pdf.html

> Notice that the default template contains some commented code providing a disqus integration. Follow the instructions in the template to enable the same disqus integration as this document.

To generate the documentation you simply need to build the project and find your documentation in your target directory:

- Projbook.Documentation

> It is possible to skip pdf generation using compilation symbols by using `PROJBOOK_NOPDF` in order to speed up Debug build while keeping it for Release builds. Since `Projbook.Core` does not including pdf generation dependencies, this symbol will be forced if you choose it instead of `Projbook`.

Snippet extraction

Projbook extends the markdown syntax in order to define snippet reference. By default, you can specify code block but you need to manually type the content this way:

```
```txt
Some code exmaple
```
```

The first syntax extension allows you to leave the content empty but reference a file. During the document project's build it will find the file content and inject it inside the code block:

```
```txt[Path/To/File.txt]
```
```

Opionally you can specify a pattern used to extract some part of the referenced. This pattern highly depending on the type of content you extract. We'll detail supported syntax and format later.

```
```txt[Path/To/File.txt] <pattern>
```
```

The syntax you define will define two things:

- The syntax highlighting that is going to be applied using prism.js
- The pattern you can apply (will be detailed below)

Ultimately you can extract any text-based content associated with any syntax hightlighting but using syntax-

specific extraction pattern will make the snippet extraction really powerful. As an example, this code block:

```
```csharp[Code/SampleClass.cs] Method(int)
```
```

The whole file content being:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Projbook.Documentation.Code
{
    public class SampleClass
    {
        public SampleClass()
        {
        }

        private void Method(string input)
        {
            Console.WriteLine(input);
        }

        private void Method(int input)
        {
            Console.WriteLine(42 + input);
        }
    }
}
```

Will be rendered as extracting the referenced method:

```
private void Method(int input)
{
    Console.WriteLine(42 + input);
}
```

Extractors

Extraction pattern are implemented as projbook plugins. Below is the documentation of the built in ones.

C# pattern

C# patterns allow referencing any syntax block member like namepsaces, classes, fields, properties, events, indexers, methods. Some options allows wipe block content or extract block content only. It mostly follow

the cref syntax with some extra options and modifications:

- Constructors are matched by `<Constructor>`
- Finalizers are matched by `<Destructor>`
- Indexers are matched by their type like `[int]`
- You can reference events and properties sub blocks by using their name like `ThePropertyName.get` or `TheEventName.add`
- It's possible to match any methods by parameters like `(string, int)` matching any method having these parameters
- The `-` prefix to extract the block content only
- The `=` prefix to extract the block structure only

All example below are using this file content:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Projbook.Documentation.Code
{
    public class SampleClass
    {
        public SampleClass()
        {
        }

        private void Method(string input)
        {
            Console.WriteLine(input);
        }

        private void Method(int input)
        {
            Console.WriteLine(42 + input);
        }
    }
}
```

Member selection will simply apply a pattern matching on the full qualified member name such as you can use either name or a full qualified name to resolve ambiguity. All of these pattern are equivalent in this context:

- `csharp[Path/To/The/File.cs] SampleClass.Method(string)`
- `csharp[Path/To/The/File.cs] Projbook.Documentation.Code.SampleClass.Method(string)`
- `csharp[Path/To/The/File.cs] (string)`

Will produce:

```
private void Method(string input)
{
    Console.WriteLine(input);
}
```

Constructors are matched by `<Constructor>` special name, `csharp[Code/SampleClass.cs] <Constructor>` would extract:

```
public SampleClass()
{
}
```

Thanks to pattern matching, in case of ambigous matching Projbook will extract all matching members and stack them up. Here `Method` having overloads, `csharp[Code/SampleClass.cs] Method` will extract all of them. Note that the member name is not fully qualified but it could if needed or preferred:

```
private void Method(string input)
{
    Console.WriteLine(input);
}

private void Method(int input)
{
    Console.WriteLine(42 + input);
}
```

> This feature is not specific to method name but also apply to parameter extracting all methods having the same parameters and can even be combined with options (see below for more details).

During the extraction process Projbook can process snippet content in extracting block structure or the code block. This is doable by adding an option prefix to the extraction rule.

The `=` char represents the top and the bottom of a code block. With `csharp[Code/SampleClass.cs] =SampleClass` Projbook will perform a member extraction and will replace the code content by `// ...`:

```
public class SampleClass
{
    // ...
}
```

The `-` char represents the content of a code block. With `csharp[Code/SampleClass.cs] -Method(int)` Projbook will perform a member extraction isolating the code content:

```
Console.WriteLine(42 + input);
```

You can combine rules for extracting and processing many member with options. The rule `csharp[Code/SampleClass.cs] =Method` will find any matching member with name `Method`, stack them up and remove the code content by `// ...`:

```
private void Method(string input)
{
    // ...
}

private void Method(int input)
{
    // ...
}
```

Xml pattern

It is also possible to extract xml content by using XPath as query language, for example, we can export all Import tag in the Projbook's documentation project by using `xml[Projbook.Documentation.csproj] //Import` :

```
<Import
  Project="..\packages\Projbook.1.1.0-cr2\build\Projbook.props"
  Condition="Exists('..\packages\Projbook.1.1.0-cr2\build\Projbook.props')" />

<Import
  Project="$(SolutionDir)build\Projbook.settings"
  Condition="Exists('$(SolutionDir)build\Projbook.settings')" />

<Import
  Project="$(MSBuildToolsPath)\Microsoft.CSharp.targets" />

<Import
  Project="..\packages\Projbook.1.1.0-cr2\build\Projbook.targets"
  Condition="Exists('..\packages\Projbook.1.1.0-cr2\build\Projbook.targets')" />
```

File system pattern

A special syntax `fs` for **file system** is handled and rendered as a tree, simply uses Projbook references to target a path in the file system with this syntax and it will be rendered as a jstree:

```
```fs[Page]
```
```

Being rendered as:

- Page
    - extractor.md
    - jumpstart.md
    - plugin.md
    - template.md

Patterns can be file search pattern such as `*.cs` or `my-file.txt` like this:

```fs[.] *.md*
```

Being rendered as:

- Projbook.Documentation

It's also possible to combine patterns using `|` or `;` chars such as the following extraction rule:

```fs[.] *.md|*template*.html
```

Produces:

- Projbook.Documentation
  - index-template.html
  - template.html
  - template-pdf.html

Templates

Default templates are using bootstrap can be directly used as it without any changes but can be modified or entirely rewritte if needed. Templates use html with Razor syntax having relevant information about documentation as model.

HTML and PDF templates

There are two template file per generation configuration that you can define in `projbook.json`:

- `template-html`: Used as template for HTML generation.
- `template-pdf`: Used as template for PDF generation.

By default the generated files are going to have the same name as the template one with the `-generated` sufix but you can define your own using:

- `output-html`: The output file for `template-html`
- `output-pdf`: The output file for `template-pdf`

> You can use html template, pdf or both but at least one must be definied.

The `@Model` variable contains two top level member:

- `Model.Title`: The documentation title from the configuration
- `Model.Pages`: An array of Page (mode details below)

Here is the `Page` class members exposed in the template templates:

```
/// <summary>
/// The page id.
/// </summary>
public string Id { get; private set; }
```

```
/// <summary>
/// The page title.
/// </summary>
public string Title { get; private set; }
```

```
/// <summary>
/// The pre section content.
/// </summary>
public string PreSectionContent { get; private set; }
```

```
/// <summary>
/// The page sections.
/// </summary>
public Section[] Sections { get; private set; }
```

Here is the `Section` class members exposed in the template templates:

```
/// <summary>
/// The section id.
/// </summary>
public string Id { get; private set; }
```

```
/// <summary>
/// The section level.
/// </summary>
public int Level { get; private set; }
```

```
/// <summary>
/// The section title.
/// </summary>
public string Title { get; private set; }
```

```
/// <summary>
/// The section content.
/// </summary>
public string Content { get; private set; }
```

Index template

The home page is generated using its own template that you can change using the following properties:

- `template` : The index template with `index-template/html` as a default value.

- `output` : The output file name with `index.html` as a default value.

From this template you can use `@Model.IndexConfiguration` containing the following member:

```
/// <summary>
/// The index title.
/// </summary>
public string Title { get; set; }
```

```
/// <summary>
/// The index description.
/// </summary>
public string Description { get; set; }
```

```
/// <summary>
/// The index output.
/// </summary>
public string Icon { get; set; }
```

```
/// <summary>
/// The index configurations.
/// </summary>
public Configuration[] Configurations { get; set; }
```

Each Configuration configurations contains:

```
/// <summary>
/// The document title.
/// </summary>
public string Title { get; set; }
```

```
/// <summary>
/// The document description.
/// </summary>
public string Description { get; set; }
```

```
/// <summary>
/// The document icon.
/// </summary>
public string Icon { get; set; }
```

> Projbook default templates use a ready to go preset bootstrap-based content but you're free to edit template to matches your needs or your project theme.