Jump start

Set up

1. Create a documentation project
2. Add Projbook using nuget
3. Update your pages in markdown format in `Page` folder
4. Update `projbook.json` to match your files:

```
{
  "title": "Projbook",
  "template-html": "template.html",
  "output-html": "projbook.html",
  "template-pdf": "template-pdf.html",
  "output-pdf": "projbook-pdf.html",
  "section-title-base": 1,
  "pages": [
    {
      "title": "Jump start",
      "path": "Page/jumpstart.md"
    },
    {
      "title": "Template",
      "path": "Page/template.md"
    },
    {
      "title": "Reference",
      "path": "Page/reference.md"
    }
  ]
}
```

5. Build the documentation project
6. The generated documentation will be available in your `TargetDir`

Extract csharp file content

For the following examples we're going to use the following file as snippet source:

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

namespace Projbook.Documentation.Code
{
   public class SampleClass
   {
      private void Method(string input)
      {
         Console.WriteLine(input);
      }

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

Extracting file content is doable by simply using a regular code block syntax and add the extraction rule between `[]`. Extraction rule syntax include the target file, the member to extract and the extraction options.

File selector

While using code block it's possible to reference a csharp file using `csharp[Path/To/The/File.cs]`.

If nothing else is defined, the whole file content will be extracted:

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

namespace Projbook.Documentation.Code
{
   public class SampleClass
   {
      private void Method(string input)
      {
         Console.WriteLine(input);
      }

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

Using csharp as syntax allows to extract specific members (see below), however any syntax could be used for extracting a file content. It's valid to extract the same file as txt using `txt[Code/SampleClass.cs]` but the syntax highlighting will be lost:

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

namespace Projbook.Documentation.Code
{
    public class SampleClass
    {
        private void Method(string input)
        {
            Console.WriteLine(input);
        }

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

Member selector

The second part of the extraction rule is the member name, you can either use the raw name or the full qualified name. All of following extraction rule will extract the same content:

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

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

Aggregate members

Thanks to the partial member 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);
}
```

Extraction options

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.

Block structure

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
{
    // ...
}
```

Block content

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);
```

Combine with member aggregation

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)
{
    // ...
}
```

Extract Xml file content

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.1.0.6\build\Projbook.props"
  Condition="Exists('..\packages\Projbook.1.0.6\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.1.0.6\build\Projbook.targets"
  Condition="Exists('..\packages\Projbook.1.0.6\build\Projbook.targets')" />
```

Template

HTML and PDF templates

There are two template file 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 file is 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`

Default templates using bootstrap can be directly used as it without any changes but can be modified or entirely rewritte if needed. You can use html template, pdf or both but at least one must be definied.

Syntax

Templates use html with [Razor syntax](http://www.asp.net/web-pages/overview/getting-started/introducing-razor-syntax-(c) having relevant information about documentation as model.

Model

The model is usable using the `@Model` variable using razor and contains top level member:

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

Pages

The `Page` class contains following members that can be used in 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; }
```

Sections

The `Section` class contains following members that can be used in templates

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

Default template

See default template.html and template-pdf.html

Reference

CSharp member matching

Here is listed all possible member matching. All of them can use partial or fully qualified name, in case of multiple matching Projbook will aggregate all matching members. The member matching never consider memeber type and will apply on any member simply based on the name. The member matching follows this syntax `csharp[<fileName>] <optionalOption><member>`.

- `<fileName>`: Any file name in the current documentation project, it's possible to reach code content outside of the document project by adding a project reference from the documentation project to another solution's project.
- `<optionalOption>`: Options member, see options.
- `<member>`: The member to extract, see the following section for details on all supported member matching

Namespaces

- `csharp[File.cs] MyNamespace`: Match the namespace `MyNameSpace`
- `csharp[File.cs] My.Namespace`: Match the namespace `My.NameSpace` or the sub namespace `Namespace` of `My` namespace

Classes

- `csharp[File.cs] ClassName`: Match any class nammed `ClassName`
- `csharp[File.cs] ClassName.SubClassName`: Match any class nammed `SubClassName` located inside `ClassName`

Methods

- `csharp[File.cs] ClassName.MethodName`: Match any method nammed `MethodName` of the class `ClassName`
- `csharp[File.cs] MethodName`: Match any method nammed `MethodName`
- `csharp[File.cs] MethodName(string, int)`: Match any method nammed `MethodName` with two parameters of type string and int
- `csharp[File.cs] (string, int)`: Match any method with two parameters of type string and int

Properties

- `csharp[File.cs] ClassName.PropertyName`: Match any property nammed `PropertyName` of the class `ClassName`
- `csharp[File.cs] Property`: Match any property nammed `PropertyName`
- `csharp[File.cs] Property.get`: Match the getter of any property nammed `PropertyName`
- `csharp[File.cs] get`: Match any getter

Indexers

- `csharp[File.cs] ClassName.[int]`: Match any indexer using the type `int` of the class `ClassName`
- `csharp[File.cs] [int]`: Match any indexer using the type `int`

Events

- `csharp[File.cs] ClassName.EventName` : Match any event nammed `EventName` of the class `ClassName`
- `csharp[File.cs] EventName` : Match any property nammed `EventName`
- `csharp[File.cs] EventName.add` : Match the adder of any event nammed `EventName`
- `csharp[File.cs] add` : Match any adder

Constructors

- `csharp[File.cs] ClassName.<Constructor>` : Match any constructor of the class `ClassName`
- `csharp[File.cs] <Constructor>` : Match any constructor
- `csharp[File.cs] <Constructor>(string, int)` : Match any constructor with two parameters of type string and int

Destructors

- `csharp[File.cs] ClassName.<Destructor>` : Match any destructor of the class `ClassName`
- `csharp[File.cs] <Destructor>` : Match any destructor

Generics

- `csharp[File.cs] ClassName{T}` : Match any class nammed `ClassName` with a type parameter `T`
- `csharp[File.cs] ClassName{T, U}` : Match any class nammed `ClassName` with two type parameter `T` and `U`
- `csharp[File.cs] ClassName.MethodName{T}` : Match any method nammed `MethodName` of the class `ClassName` with a type parameter `T`
- `csharp[File.cs] ClassName.MethodName{T}(T)` : Match any method nammed `MethodName` of the class `ClassName` with a type parameter `T` having a parameter of type `T`
- `csharp[File.cs] ClassName{T}.MethodName{U}(U)` : Match any method nammed `MethodName` of the class `ClassName` having a type parater `T` with a type parameter `U` having a parameter of type `U`

Options

- `=` : Code structure only including the member with all motifier, documentation and signature but replace the code block content by `// ...`
- `-` : Code content only, it will remove the code structure and will produce what's located inside the code block only

Xml member matching

CSharp matching simply follow Projbook's syntax for member matching but use XPath as member selector. The member matching follows this syntax `csharp[<fileName>] <xpath>` .

- `<fileName>` : Any file name in the current documentation project, it's possible to reach code content outside of the document project by adding a project reference from the documentation project to another solution's project.
- `<xpath>` : The XPath query performing the member selection, see XPath reference

Source example

Have a look to this documentation pages as example source