ExDoc.Markdown behaviour

Adapter behaviour and conveniences for converting Markdown to HTML.

ExDoc is compatible with any markdown processor that implements the functions defined in this module. The markdown processor can be changed via the `:markdown_processor` option in your `mix.exs` or via the `:markdown_processor` configuration in the `:ex_doc` configuration.

ExDoc supports the following Markdown parsers out of the box:

- Earmark
- Cmark

ExDoc uses Earmark by default.

Summary

Functions

Callbacks

assets(atom)

Assets specific to the markdown implementation

before_closing_body_tag(atom)

Literal content to be written to the file just before the closing body tag

before_closing_head_tag(atom)

Literal content to be written to the file just before the closing head tag

configure(any)

A function that accepts configuration options and configures the markdown processor

to_html(arg0, arg1)

Converts markdown into HTML

Functions

get_markdown_processor() </>

Gets the current markdown processor set globally.

pretty_codeblocks(bin) </>

Helper to handle plain code blocks (`...`) with and without language specification and indentation code blocks.

put_markdown_processor(processor) </>

Changes the markdown processor globally.

```
to_html(text, opts \\ [])
```

Converts the given markdown document to HTML.

Callbacks

```
assets(atom)
assets(atom()) :: [{String.t(), String.t()}]
```

Assets specific to the markdown implementation.

This callback takes the documentation format (`:html` or `:epub`) as an argument and must return a list of pairs of the form `{basename, content}` where:

- `basename` – relative path that will be written inside the `doc/` directory.
- `content` – is a binary with the full contents of the file that will be written to `basename`.

EPUB Documentation Gotchas

Generating HTML documentation is simple, and it works exacly as you would expect for a webpage. The EPUB file format, on the other hand, may cause some surprise.

Apparently, an EPUB file expects all assets to have a unique name *when discarding the file extension*.

This creates problems if you include, for example, the files `custom.js` and `custom.css`. Because the filename without the extension is equal (`custom`), you will get an unreadable EPUB. It's possible to go around this limitation by simply giving the files unique names:

- `custom.js` becomes `custom-js.js` *and*
- `custom.css` becomes `custom-css.css`

Example

```
def callback assets(_) do
  [{"dist/custom-css.css", custom_css_content()},
   {"dist/custom-js.js", custom_js_content()}]
end
```

before_closing_body_tag(atom) </>

```
before_closing_body_tag(atom()) :: String.t()
```

Literal content to be written to the file just before the closing body tag.

This callback takes the documentation format (`:html` or `:epub`) as an argument and returns a literal string. It is useful when the markdown processor needs to a include extra JavaScript.

Example

```
def callback before_closing_body_tag(_) do
  # Include the Javascript specified in the assets/1 callback
  ~S(<script src="dist/custom-js.js"></script>)
end
```

before_closing_head_tag(atom) </>

```
before_closing_head_tag(atom()) :: String.t()
```

Literal content to be written to the file just before the closing head tag.

This callback takes the documentation format (`:html` or `:epub`) as an argument and returns a literal string. It is useful when the

markdown processor needs to a include extra CSS.

Example

```
def callback before_closing_head_tag(_) do
  # Include the CSS specified in the assets/1 callback
  ~S(<link rel="stylesheet" href="dist/custom-css.css"/>)
end
```

configure(any) </>

```
configure(any()) :: :ok
```

A function that accepts configuration options and configures the markdown processor.

It is run once when `:ex_doc` is loaded, and the return value is discarded. Modules that implement this behaviour will probably store the options somewhere so that they can be accessed when needed.

The format of the options as well as what the function does with them is completely up to the module that implements the behaviour.

to_html(arg0, arg1) </>

```
to_html(String.t(), Keyword.t()) :: String.t()
```

Converts markdown into HTML.