Verto Documentation

Release 0.11.0

Hayley vas Waas, Jack Morgan

May 03, 2019

Contents

1 Installing Verto **3**

2 Using Verto **5**

- 2.1 Step 1: Importing Verto . . . 5
- 2.2 Step 2: Creating Verto converter . . . 5
- 2.3 Step 3: Convert Markdown with converter . . . 7
- 2.4 Step 4: Accessing VertoResult data . . . 7
- 2.5 (Optional) Step 5: Clearing Saved Data . . . 8
- 2.6 Configuring Verto converter after creation . . . 8
- 2.7 Full list of package methods . . . 9

3 Using Processors **11**

4 Available Processors **13**

- 4.1 Blockquote . . . 13
- 4.2 Boxed Text . . . 15
- 4.3 Button Link . . . 16
- 4.4 Comment . . . 17
- 4.5 Conditional . . . 18
- 4.6 Embed iframe . . . 21
- 4.7 Glossary Link . . . 22
- 4.8 Heading . . . 24
- 4.9 Image . . . 27
- 4.10 Inline Image . . . 30
- 4.11 Interactive . . . 33
- 4.12 Panel . . . 36
- 4.13 Relative Link . . . 39
- 4.14 Remove Title . . . 40
- 4.15 Save Title . . . 40
- 4.16 Scratch . . . 41
- 4.17 Scratch (Inline) . . . 45
- 4.18 Table of Contents . . . 46
- 4.19 Video . . . 48

5 Implicit Processors **51**

- 5.1 Jinja . . . 51
- 5.2 Ordered List (OListProcessor) . . . 52

5.3 Remove . . . 52
5.4 Scratch Compatibility . . . 53
5.5 Style . . . 54
5.6 Unordered List (UListProcessor) . . . 56

6 Extensions **57**
6.1 Math . . . 57
6.2 Fenced Code . . . 57
6.3 Code Highlighting . . . 58
6.4 Sane Lists . . . 58

7 Contributing to Verto **59**
7.1 Issue Reporting and Bug Fixes . . . 59
7.2 The Code Base . . . 60
7.3 The Test Suite . . . 65
7.4 Creating a release . . . 67
7.5 Other notes . . . 68

8 Changelog **69**
8.1 0.11.0 . . . 69
8.2 0.10.0 . . . 69
8.3 0.9.3 . . . 70
8.4 0.9.2 . . . 70
8.5 0.9.1 . . . 70
8.6 0.9.0 . . . 70
8.7 0.8.0 . . . 70
8.8 0.7.4 . . . 70
8.9 0.7.3 . . . 71
8.10 0.7.2 . . . 71
8.11 0.7.1 . . . 71
8.12 0.7.0 . . . 71
8.13 0.6.1 . . . 72
8.14 0.6.0 . . . 72
8.15 0.5.3 . . . 72
8.16 0.5.2 . . . 72
8.17 0.5.1 . . . 72
8.18 0.5.0 . . . 72
8.19 0.4.1 . . . 73
8.20 0.4.0 . . . 73
8.21 0.3.1 . . . 73
8.22 0.3.0 . . . 74
8.23 0.2.0 . . . 74
8.24 0.1.0 . . . 74

Verto is an extension of the Python Markdown package, which allows authors to include complex HTML elements with simple text tags in their Markdown files.

For example:

```
>>> import verto
>>> converter = verto.Verto()
>>> result = converter.convert('{video url="https://www.youtube.com/watch?
↪v=dQw4w9WgXcQ"}')
>>> result.html_string
"<iframe src='http://www.youtube.com/embed/dQw4w9WgXcQ?rel=0' frameborder='0'␣
↪allowfullscreen></iframe>"
```

CHAPTER 1

Installing Verto

Currently Verto is only able to be installed locally.

The following commands in Terminal will download and install Verto ready for use:

```
$ git clone https://github.com/uccser/verto.git
$ cd verto
$ pip3 install .
```

CHAPTER 2

Using Verto

Using Verto to convert Markdown is a process of:

1. Importing the installed Verto package.
2. Creating a Verto converter.
3. Passing Markdown text through the Verto `convert` method and saving the result object.
4. Accessing data from the result object.

2.1 Step 1: Importing Verto

If Verto has been installed correctly, importing the package can be completed by:

```
import verto
```

2.2 Step 2: Creating Verto converter

Once the module is imported, you can create a Verto converter creating an Verto object:

```
converter = verto.Verto()
```

`Verto()` has optional parameters to customise the converter. These are:

- `processors` - A set of processor names given as strings for the converter to use. If this parameter is not given, the default processors are used. If `processors` is provided, all processors not listed are skipped.
  - *For example:* Creating a Verto converter that only deletes comment tags would be done by the following command:

    ```
    converter = verto.Verto(processors={"comment"})
    ```

- `html_templates` - A dictionary of HTML templates to override existing HTML templates for processors. The dictionary contains processor names given as a string as keys mapping HTML strings as values.

  The documentation page for each processor specificies how to create custom HTML for that processor.

  - *For example:* Creating a Verto converter that uses `<img src={{ source }}>` as custom HTML for `image` tags would be done by the following command:

    ```
    converter = verto.Verto(html_templates={'image': '<img src={{ source }}>'})
    ```

- `extensions` - A list of extra Markdown extensions to run in the converter. Details on how to use this parameter can be found on the *Extensions* page.
- `settings` - A dictionary of settings to override default Verto settings. The following settings are available:
  - `add_default_interactive_thumbnails_to_required_files` - Boolean stating whether default interactive thumbnail filepaths should be added to the required files set of images. Default is `True`.
  - `add_custom_interactive_thumbnails_to_required_files` - Boolean stating whether custom interactive thumbnail filepaths provided as tag arguments should be added to the required files set of images. External images are never added. Default is `True`.
  - `processor_argument_overrides` - A dictionary to modify the default argument rules for each tag. The default rules can found by reading the documentation for each tag.
    - *For example:* By default, the `image-inline` tag requires alt text to be given, to change this, the following custom argument rules would be used:

      ```
      {
        "image-inline": {
          "alt": False
        }
      }
      ```

> **Warning:** Some tags have multiple processors behind them (for example, the `image-inline`, `image-container` and `image-tag` processors are all used for images). This means that if you would like to change the default rules of one or more of their arguments, this will need to be done for each of the processors individually. For example, to set the `alt` argument as `False` for all images, the custom argument rules would look as follows:
>
> ```
> {
>   "image-inline": {
>     "alt": False
>   },
>   "image-tag": {
>     "alt": False
>   },
>   "image-container": {
>     "alt": False
>   }
> }
> ```

2.3 Step 3: Convert Markdown with converter

To convert Markdown to HTML with a Verto converter, we call the `convert()` method. The method returns a `VertoResult` object.

```
text = """
This **is** a line of Markdown.

{comment This is a comment using a Verto tag}

This is a *different* line of Markdown.
"""
result = converter.convert(text)
```

2.4 Step 4: Accessing VertoResult data

The `VertoResult` object contains several attributes which can be accessed using the dot notation. Continuing from our previous example, the following command would print the converted HTML.

```
print(result.html_string)
```

The following attributes are available:

- `html_string` - A resulting string of HTML after conversion by Verto.
- `title` - The text of the first heading saved by the `save-title` processor.
- `required_files` - A dictionary of files encountered in a Verto conversion. The dictionary has a string for the file type as the key (for example: `image`) and a set of all file paths encountered as the value (for example: `{'image/face.png', 'image/logo.png`}`).
  - See *Accessing Scratch image data* for data from Scratch processor.
- `heading_tree` - A tuple of namedtuples which describes the tree of headings, as generated by our heading processor. Each namedtuple contains a `title` (string), `title_slug` (string), `level` (integer) and `children` (tuple of nodes).
  - For example the heading root after a conversion of a file:

```
(
  HeadingNode(title='This is an H1', title_slug='this-is-an-h1', level=1,
↪children=(
    HeadingNode(title='H2 Title', title_slug='h2-title', level=2, children=())
  )),
  HeadingNode(title='This is an H1', title_slug='this-is-an-h11', level=1,
↪children=()),
)
```

- `required-glossary-terms` - A dictionary of term slugs to a list of tuples containing reference text and link IDs.
  - Here is an example of the `required-glossary-terms` after a conversion of a file:

```
required-glossary-terms = {
  "algorithm":
    [("Binary Search", "glossary-algorithm"),
```

(continues on next page)

(continued from previous page)

```
        ("Quick Sort", "glossary-algorithm-2"),
        ("Merge Sort", "glossary-algorithm-3")],
    "alphabet":
      [("Formal Languages", "glossary-alphabet")],
    "brooks-law":
      []
}
```

2.5 (Optional) Step 5: Clearing Saved Data

Lastly there is some data that is saved between conversions such as `required_files` and unique ids used in the `glossary` and for `headings`. This can be cleared by using the following method:

```
converter.clear_saved_data()
```

2.6 Configuring Verto converter after creation

The following functions allow you to change the processors or HTML templates used in conversion by the Verto converter after its creation.

2.6.1 Changing processors

`Verto.`**`update_processors`**(*processors*)

Update the processors used for conversion with the given set. The updated set will be used for converting from this point onwards. If parameter is empty, default processors will be used.

Parameters **`processors`** – A set of processor names given as strings for which their processors are enabled. If given, all other processors are skipped.

`static` `Verto.`**`processor_defaults`**(*processors*)

Returns a copy of the default processor set.

Returns A set of default processor names as strings.

This function is useful if you want to make minor changes to the default used processors. For example: You wish to still use all default processors but skip video tags:

```
processors = Verto.processor_defaults()
processors.remove('video')
converter = Verto(processors=processors)
```

Or with an existing Verto instance `converter`:

```
processors = Verto.processor_defaults()
processors.remove('video')
converter.update_processors(processors)
```

2.6.2 Changing HTML templates

Verto.**update_templates**(*html_templates*)

Update the template dictionary with the given dictionary of templates, while leaving all other HTML templates (including any custom set templates) untouched. The updated dictionary will be used for converting from this point onwards.

Parameters **html_templates** – A dictionary of HTML templates to override existing HTML templates for processors. Dictionary contains processor names given as a string as keys mapping HTML strings as values. eg: {'image': '<img src={{ source }}>'}

Verto.**clear_templates**()

Set the template dictionary to it's original values.

2.7 Full list of package methods

class verto.**Verto**

A converter object for converting markdown with complex elements to HTML.

__init__(*processors=frozenset({'image-tag', 'button-link', 'interactive-tag', 'table-of-contents', 'image-inline', 'relative-link', 'conditional', 'image-container', 'heading', 'interactive-container', 'boxed-text', 'scratch', 'blockquote', 'iframe', 'comment', 'save-title', 'style', 'glossary-link', 'scratch-inline', 'video', 'panel'}), html_templates={}, extensions=[], custom_settings={}*)

Creates a Verto object.

Parameters

- **processors** – A set of processor names given as strings for which their processors are enabled. If given, all other processors are skipped.
- **html_templates** – A dictionary of HTML templates to override existing HTML templates for processors. Dictionary contains processor names given as a string as keys mapping HTML strings as values. eg: {'image': '<img src={{ source }}>'}
- **extensions** – A list of extra extensions to run on the markdown package.
- **custom_settings** – A dictionary of settings to override default settings.

clear_saved_data()

Clears data that is saved between documents. This should be called between conversions on unrelated documents.

clear_templates()

Set the template dictionary to it's original values.

convert(*text*)

Return a VertoResult object after converting the given markdown string.

Parameters **text** – A string of Markdown text to be converted.

Returns A VertoResult object.

static processor_defaults()

Returns a copy of the default processor set.

Returns A set of default processor names as strings.

update_processors(*processors=frozenset({'image-tag', 'button-link', 'interactive-tag', 'table-of-contents', 'image-inline', 'relative-link', 'conditional', 'image-container', 'heading', 'interactive-container', 'boxed-text', 'scratch', 'blockquote', 'iframe', 'comment', 'save-title', 'style', 'glossary-link', 'scratch-inline', 'video', 'panel'})*)

Update the processors used for conversion with the given set. The updated set will be used for converting from this point onwards. If parameter is empty, default processors will be used.

Parameters **processors** – A set of processor names given as strings for which their processors are enabled. If given, all other processors are skipped.

update_templates(*html_templates*)

Update the template dictionary with the given dictionary of templates, while leaving all other HTML templates (including any custom set templates) untouched. The updated dictionary will be used for converting from this point onwards.

Parameters **html_templates** – A dictionary of HTML templates to override existing HTML templates for processors. Dictionary contains processor names given as a string as keys mapping HTML strings as values. eg: {'image': '<img src={{ source }}>'}

class verto.Verto.**VertoResult**

Object created by Verto containing the result data after a conversion by run.

html_string

The converted HTML as a string.

title

The text of the first heading found by the *Save Title* processor.

required_files

A dictionary of files encountered in a Verto conversion. The dictionary has a string for the file type as the key (for example: `image`) and a set of all file paths encountered as the value (for example: `{'image/face.png', 'image/logo.png`}`).

heading_tree

A tuple of namedtuples which describes the tree of headings, as generated by our heading processor. Each namedtuple contains a title (string), title_slug (string), level (integer) and children (tuple of nodes).

required_glossary_terms

A dictionary of terms to a list of tuples containing reference text and link IDs.

CHAPTER 3

Using Processors

- Tags should always be separated with newlines before and after each tag.
- If a processor requires both a start and end tag (for example: panels) and one tag is missing then a `TagNotMatchedError` Exception will be thrown.
- You can escape a double quote in a parameter with a slash `\"` though this also means you can never end a parameter with a `\`.

CHAPTER 4

Available Processors

The following pages covers how to use the available processors within Markdown text:

4.1 Blockquote

Processor name: `blockquote`

You can include an blockquote using the following text tag:

```
{blockquote}

This text is the blockquote's contents.

This is the second line.

{blockquote end}
```

4.1.1 Optional Tag Parameters

- `footer` - Boolean flag to indicate whether the blockquote contains a footer.
  - If given as `true`, then the last line should start with "- " to show it's the footer.
- `source` - Sets the `cite` parameter of the `blockquote` element.
- `alignment` - Valid values are 'left', 'center', or 'right'. Providing one of these values Will add CSS classes to the image for alignment.

The default HTML for a panel is:

```
<blockquote class="blockquote{% if alignment == 'left' %} text-left{% elif alignment
↪=='center' %} text-center{% elif alignment =='right' %} text-right{% endif %}"
{%- if source %} cite="{{ source }}"{% endif %}>
```

(continues on next page)

(continued from previous page)

```
{% autoescape false -%}
{{ content }}
{%- endautoescape -%}
{% if footer %}
<footer class="blockquote-footer">
{{ footer }}
</footer>
{% endif %}
</blockquote>
```

Using the following example tag:

```
{blockquote}

This text is the blockquote's contents.

This is the second line.

{blockquote end}
```

The resulting HTML would be:

```
<blockquote class="blockquote">
<p>This text is the blockquote's contents.</p>
<p>This is the second line.</p>
</blockquote>
```

4.1.2 Overriding HTML for Blockquote

When overriding the HTML for blockquotes, the following Jinja2 placeholders are available:

- `{{ content }}` - The text enclosed by the blockquote tags.
- `{{ footer }}` - The provided footer text.
- `{{ alignment }}` - The location to add extra CSS classes for alignment.
- `{{ source }}` - The URL for the source.

Example

For example, providing the following HTML:

```
<blockquote class="blockquote grey-text{% if alignment == 'left' %} float-left{% elif␣
↪alignment =='center' %} float-center{% elif alignment =='right' %} float-right{%␣
↪endif %}"
{%- if source %} cite="{{ source }}"{% endif %}>
{% autoescape false -%}
{{ content }}
{%- endautoescape -%}
{% if footer %}
<footer class="blockquote-footer">
{{ footer }}
</footer>
{% endif %}
</blockquote>
```

with the following tag:

```
{blockquote alignment="right"}

Blockquote contents.

{blockquote end}
```

would result in:

```
<blockquote class="blockquote grey-text float-right">
<p>Blockquote contents.</p>
</blockquote>
```

4.2 Boxed Text

Processor name: `boxed-text`

You can enclose text inside of a box using the following text tag:

```
{boxed-text}

**Computer Science Report for 2.44**

Put your introduction to what bits are here.

{boxed-text end}
```

4.2.1 Optional Tag Parameters

- `indented` - If `yes`, the box will have indentation on the left to match indentation of the first level of a list.

The default HTML for button links is:

```
<div class='boxed-text{% if indented == "yes" %} boxed-text-indented{% endif %}{% if␣
↪type %} boxed-text-{{ type }}{% endif %}'>
{% autoescape false -%}
{{ text }}
{% endautoescape -%}
</div>
```

Using the example tag above, the resulting HTML would be:

```
<div class="boxed-text">
<p><strong>Computer Science Report for 2.44</strong></p>
<p>Put your introduction to what bits are here.</p>
</div>
```

4.2.2 Overriding HTML for Boxed Text

When overriding the HTML for boxed text, the following Jinja2 placeholders are available:

- `{{ text }}` - The text enclosed by the boxed text tags.
- `indented` - Set to `yes` if the indentation parameter was set to `True`.

Example

For example, providing the following HTML:

```
<div class="card">
<div class="card-block">
{{ text }}
</div>
</div>
```

with the following tag:

```
{boxed-text}

This text is displayed using a Bootstrap 4 card.

{boxed-text end}
```

would result in:

```
<div class="card">
<div class="card-block">
<p>This text is displayed using a Bootstrap 4 card.</p>
</div>
</div>
```

4.3 Button Link

Processor name: `button-link`

You can create a link on a button using the following text tag:

```
{button-link link="http://www.google.com" text="Visit Google"}
```

4.3.1 Required Tag Parameters

- `link` - The URL to link to.
  - If the given link is a relative, a placeholder for Django to prepend the root is outputted.
  - If the `file` parameter is set to `yes`, then the link will be rendered with a Django static command. See `file` parameter below.
- `text` - Text to display on the button.

4.3.2 Optional Tag Parameters

- `file` - If set to `yes` the link will be rendered with a Django static command. This is useful if you wish to create a button link to a file or image.
  - For example, the link `files/python-sort-example.py` would be rendered as `{% static 'files/python-sort-example.py' %}`. This can be overriden, see the override section below.:

The default HTML for button links is:

```
<a class='button' href='{{ link }}'>{{ text }}</a>
```

Example 1

Using the following tag:

```
{button-link link="http://www.google.com" text="Visit Google"}
```

The resulting HTML would be:

```
<a class="button" href="http://www.google.com">Visit Google</a>
```

Example 2

Using the following tag:

```
{button-link link="files/python-sort-example.py" text="Download Python Sorting Example
↪" file="yes"}
```

The resulting HTML would be:

```
<a class="button" href="{% static 'files/python-sort-example.py' %}">Download Python␣
↪Sorting Example</a>
```

4.3.3 Overriding HTML for Button Link

When overriding the HTML for button links, the following Jinja2 placeholders are available:

- `{{ link }}` - The URL.
- `{{ text }}` - Text to display on the button.

If the `file` parameter is set to `yes`, the link is passed through the `relative-image-link.html` template. The default HTML for relative images is:

```
{% autoescape false -%}
{{ "{% static '" }}{{ file_path }}{{ "' %}" }}
{%- endautoescape %}
```

Example

For example, providing the following HTML:

```
<a class="btn btn-primary" href="{{ link }}">{{ text }}</a>
```

with the following tag:

```
{button-link link="https://github.com/uccser/verto" text="Verto on GitHub"}
```

would result in:

```
<a class="btn btn-primary" href="https://github.com/uccser/verto">Verto on GitHub</a>
```

4.4 Comment

Processor name: `comment`

You can include comments in the source text that are deleted before conversion:

```
{comment This is a comment for other authors to read}
```

Comment tags have no parameters or HTML templates associated with them.

Example

The following text:

```
Finally add a teaspoon of salt and tumeric.

{comment Could mention useful tools for combining mixture}

Combine and mix ingredients in a large bowl.

{comment Should also add temperature in Fahrenheit}

Bake at 180C for 40 minutes.
```

would result in (after Verto has finished conversion):

```
<p>Finally add a teaspoon of salt and tumeric.</p>
<p>Combine and mix ingredients in a large bowl.</p>
<p>Bake at 180C for 40 minutes.</p>
```

4.5 Conditional

Processor name: `conditional`

Danger: Conditional blocks require an understanding of Python logical operators and expressions to function properly. The use of this tag requires co-ordination between authors and developers, as the variables used in the condition are expected when the result is rendered in a template engine.

You can include an conditional using the following text tag:

```
{conditional if condition="version == 'teacher'"}

This is text that only teachers should see.

{conditional end}
```

4.5.1 Tag Parameters

Conditional tags function slightly differently to other block text tags. The first conditional text tag must contain the `if` flag, followed by optional conditional text tags containing the `elif` flag, followed by the optional text tag containing the `else` flag.

Note: Any strings within your Python expression need to be wrapped with single quotes (`'`) or escaped double quotes (`\"`).

Examples:

```
{conditional if="version == 'teacher'"}
{conditional if="version == \"teacher\""}
```

To create a set of conditional text tags, follow the following steps:

1. First if block

- Contains `if` flag - Specifies that the conditonal is an opening if statement.
- Contains `condition` parameter - A Python expression to evaluate if true. If true, the enclosed content will be displayed.

2. Then optional else if blocks. Multiple of these blocks can be listed after one another.

- Contains `elif` flag - Specifies that the conditonal is an *else if* following an opening if statement.
- Contains `condition` parameter - A Python expression to evaluate if true. If true, the enclosed content will be displayed.

3. Then optional else block

- Contains `else` flag - Specifies that the conditional is an *else* following an opening if statement, and optional elif statements.

4. Lastly conditional end block

Here is a more complicated example:

```
{conditional if condition="version == 'teacher'"}

This is text that only teachers should see.

{conditional elif condition="version == 'instructor'"}

This is text that only instuctors should see.

{conditional elif condition="version == 'coordinator'"}

This is text that only coordinators should see.

{conditional else}

This is text that everyone else should see.

{conditional end}
```

Example

The default HTML for a conditional is:

```
<remove>
{% autoescape false -%}
{{ "{% if " }}{{ if_expression }}{{ " %}" }}
{{ if_content }}
{% for elif_expression, elif_content in elifs.items() -%}
{{ "{% elif " }}{{ elif_expression }}{{ " %}" }}
{{ elif_content }}
{% endfor -%}
{% if has_else -%}
{{ "{% else %}" }}
```

(continues on next page)

(continued from previous page)

```
{{ else_content }}
{% endif -%}
{{ "{% endif %}" }}
{% endautoescape -%}
</remove>
```

Using the following example tag:

```
{conditional if condition="version == 'teacher'"}

This is text that only teachers should see.

{conditional end}
```

The resulting HTML would be:

```
{% if version == 'teacher' %}
<p>This is text that only teachers should see.</p>
{% endif %}
```

4.5.2 Overriding HTML for Conditional

When overriding the HTML for conditionals, the following Jinja2 placeholders are available:

- `{{ if_expression }}` - The expression from the conditional argument.
- `{{ if_content }}` - The content contained within the if expression.
- `{{ elifs }}` - An ordered dicitionary of *elif* expressions to content.
- `{{ has_else }}` - Whether an else was used in the full if statement.
- `{{ else_content }}` - The expression from the conditional argument.

Example

For example, if you wanted to output a mako template you would providing the following HTML:

```
<remove>
{% autoescape false -%}
{{ "% if " }}{{ if_expression }}{{ ":" }}
{{ if_content }}
{% for elif_expression, elif_content in elifs.items() -%}
{{ "% elif " }}{{ elif_expression }}{{ ":" }}
{{ elif_content }}
{% endfor -%}
{% if has_else -%}
{{ "% else:" }}
{{ else_content }}
{% endif -%}
{{ "% endif" }}
{% endautoescape -%}
</remove>
```

with the following tag:

```
{conditional if condition="version == 'teacher'"}

This is text that only teachers should see.

{conditional elif condition="version == 'instructor'"}

This is text that only instuctors should see.

{conditional elif condition="version == 'coordinator'"}

This is text that only coordinators should see.

{conditional end}
```

would result in:

```
% if version == 'teacher':
<p>This is text that only teachers should see.</p>
% elif version == 'instructor':
<p>This is text that only instuctors should see.</p>
% elif version == 'coordinator':
<p>This is text that only coordinators should see.</p>
% endif
```

4.6 Embed iframe

Processor name: `iframe`

You can embed a link within an `iframe` using the following text tag:

```
{iframe link="http://www.google.com"}
```

4.6.1 Required Tag Parameters

- `link` - The URL to embed within the `iframe` element.

The default HTML for iframe is:

```
<iframe src="{{ link }}">
<p>Your browser does not support iframes.</p>
</iframe>
```

Example

Using the following tag:

```
{iframe link="http://www.google.com"}
```

The resulting HTML would be:

```
<iframe src="http://www.google.com">
<p>Your browser does not support iframes.</p>
</iframe>
```

4.6.2 Overriding HTML for Emedding iframes

When overriding the HTML for iframes, the following Jinja2 placeholders are available:

- `{{ link }}` - The URL to embed within the `iframe` element.

Example

For example, providing the following HTML:

```
<iframe src="{{ link }}" class="embed-iframe" width="400" height="300" frameborder="0
↪">
</iframe>
```

with the following tag:

```
{iframe link="https://github.com/"}
```

would result in:

```
<iframe class="embed-iframe" frameborder="0" height="300" src="https://github.com/"␣
↪width="400">
</iframe>
```

4.7 Glossary Link

Processor name: `glossary-link`

You can include a link to a glossary term using the following text tag:

```
It's worth considering which {glossary-link term="algorithm"}algorithms{glossary-link␣
↪end} should be used.
```

Note: The `glossary-link` tag is an inline tag.

4.7.1 Required Tag Parameters

- `term` - The slug of the term to link to in the glossary (for example `binary-search`).
  - Each term encountered is added to the required glossary terms stored by Verto. The set of terms can be accessed after conversion, see *Step 4: Accessing VertoResult data*.
  - When using the default HTML template, the term is stored in a data attribute named `data-glossary-term`.

4.7.2 Optional Tag Parameters

- `reference-text` - If included, adds a back reference link using the given the text after the definition.
  - If back reference text is included, then an ID should be generated for the link. Currently the ID is generated as `glossary-` + the term slug. If successive back links for the same term are found then an incrementing number is appended to the generated ID.

The back reference text and ID are stored in the required glossary terms, and can be accessed after conversion, see *Step 4: Accessing VertoResult data*.

Warning: The IDs generated assume no other Verto generated HTML is included on the same page. If two (or more) separate conversions of Markdown by Verto are displayed on the same page, there may be ID conflicts.

Note: The text between the start and end tags is used as text to display as a link to the glossary definition. If no text is given, then the link is invisible and can be used to add a back reference to be particular point.

The default HTML for glossary links is:

```
<a class="glossary-term"
 data-glossary-term="{{ term }}"
 {% if id %} id="{{ id }}" {% endif %}>{{ text }}</a>
```

Using the following example tag:

```
It's worth considering which {glossary-link term="algorithm"}algorithms{glossary-link␣
↪end} should be used.
```

The resulting HTML would be:

```
<p>It's worth considering which <a class="glossary-term" data-glossary-term="algorithm
↪">algorithms</a> should be used.</p>
```

4.7.3 Possible Glossary Usage with Django

While Verto will generate links for glossary terms from the HTML template, these links will not work by themselves.

The expected usage is to display the links on a webpage and when a user clicks a link, a JavaScript handler will catch the click. The handler can view the term in the `data-glossary-term` attribute and send a request to a database to retrieve the term's definition and associated data. The handler can then display this information via popup/modal/etc once it recieves the data.

Verto also provides the back reference text and ID for terms in the required glossary terms which can be accessed after conversion (see *Step 4: Accessing VertoResult data*). This data can be added to the database before the web server is run for displaying to users.

4.7.4 Overriding HTML for Glossary Links

When overriding the HTML for glossary links, the following Jinja2 placeholders are available:

- `{{ term }}` - The slug of the term to link to.
- `{{ text }}` - The text inclosed by `{glossary-link}` tags that should be the link to the glossary definition.
- `{{ id }}` - The ID of the glossary term's back reference.

Example

For example, we wish to create glossary links that link to a static glossary page, where each definition has a header with an ID of the term's slug.

By providing the following HTML:

```
<a class="glossary-term"
 href="{{ "{{ base_path }}" }}glossary#{{ term }}"
 {% if id %} id="{{ id }}" {% endif %}>{{ text }}</a>
```

with the following tag:

```
It's worth considering which {glossary-link term="algorithm" reference-text="Software
↪Engineering"}algorithms{glossary-link end} should be used.
```

would result in:

```
<p>It's worth considering which <a class="glossary-term" href="{{ base_path }}glossary
↪#algorithm" id="glossary-algorithm">algorithms</a> should be used.</p>
```

4.8 Heading

Processor name: `heading`

This replaces the output of Markdown headings that begin with the # character (atx-style headings). This processor an ID to each heading which allows for linking to a heading, and adds the heading number before the heading text.

Note: This processor replaces the output of the standard markdown block processor for atx-style headings.

You may create a heading by using the following format:

```
# This is an H1

## This is an H2

###### This is an H6
```

The default HTML for headings is:

```
<{{ heading_type }} id="{{ title_slug }}">
<span class="section_number">
{% if heading_level >= 1 -%}
{{ "{{ chapter.number }}" }}
{%- endif -%}
{%- if heading_level >= 2 -%}
.{{ level_2 }}
{%- endif -%}
{%- if heading_level >= 3 -%}
.{{ level_3 }}
{%- endif -%}
{%- if heading_level >= 4 -%}
.{{ level_4 }}
{%- endif -%}
{%- if heading_level >= 5 -%}
.{{ level_5 }}
{%- endif -%}
{%- if heading_level >= 6 -%}
.{{ level_6 }}
```

(continues on next page)

(continued from previous page)

```
{%- endif -%}
.
</span>
{{ title }}
</{{ heading_type }}>
```

Example

Using the following tag:

```
# This is an H1

## This is an H2

###### This is an H6
```

The resulting HTML would be:

```
<h1 id="this-is-an-h1">
<span class="section_number">
{{ chapter.number }}.
</span>
This is an H1
</h1>
<h2 id="this-is-an-h2">
<span class="section_number">
{{ chapter.number }}.1.
</span>
This is an H2
</h2>
<h6 id="this-is-an-h6">
<span class="section_number">
{{ chapter.number }}.1.0.0.0.1.
</span>
This is an H6
</h6>
```

4.8.1 Overriding HTML for Heading

When overriding the HTML for heading, the following Jinja2 placeholders are available:

- `{{ headling_level }}` - A number representing the heading level.
- `{{ heading_type }}` - The string of the heading tag i.e. *h1* etc.
- `{{ title }}` - The title of the heading.
- `{{ title_slug }}` - A slug of the heading, useful for ids.
- `{{ level_1 }}` - The current first level heading number.
- `{{ level_2 }}` - The current second level heading number.
- `{{ level_3 }}` - The current third level heading number.
- `{{ level_4 }}` - The current fourth level heading number.
- `{{ level_5 }}` - The current fifth level heading number.
- `{{ level_6 }}` - The current sixth level heading number.

The `level` parameters are useful for generating levels trails so that users know where they are exactly within the document.

Example

For example, providing the following HTML:

```
<{{ heading_type }} id="{{ title_slug }}">
<span class="section_number">
{{ level_1 }}.{{ level_2 }}.{{ level_3 }}.{{ level_4 }}.{{ level_5 }}.{{ level_6 }}.
</span>
{{ title }}
</{{ heading_type }}>
```

with the following markdown:

```
# This is an H1

## This is an H2

#### This is an H4

# This is an H1

### This is an H3

###### This is an H6

# This is an H1
```

would result in:

```
<h1 id="this-is-an-h1">
<span class="section_number">
1.0.0.0.0.0.
</span>
This is an H1
</h1>
<h2 id="this-is-an-h2">
<span class="section_number">
1.1.0.0.0.0.
</span>
This is an H2
</h2>
<h4 id="this-is-an-h4">
<span class="section_number">
1.1.0.1.0.0.
</span>
This is an H4
</h4>
<h1 id="this-is-an-h1-2">
<span class="section_number">
2.0.0.0.0.0.
</span>
This is an H1
</h1>
<h3 id="this-is-an-h3">
<span class="section_number">
2.0.1.0.0.0.
```

(continues on next page)

(continued from previous page)

```
</span>
This is an H3
</h3>
<h6 id="this-is-an-h6">
<span class="section_number">
2.0.1.0.0.1.
</span>
This is an H6
</h6>
<h1 id="this-is-an-h1-3">
<span class="section_number">
3.0.0.0.0.0.
</span>
This is an H1
</h1>
```

4.9 Image

Processor name: `image-tag` and `image-container` (see also *Inline Image*)

You can include an image using the following text tag:

```
{image file-path="http://placehold.it/350x150" caption="true"}

This is the caption text.

{image end}
```

The caption is the first block of text within the image block. It is within the block rather than given as a parameter to allow translation systems to easily identify which text should be translated.

If a caption is not needed, an end tag is not required (see example below).

```
{image file-path="http://placehold.it/350x150" caption="false"}
```

4.9.1 Required Tag Parameters

- `file-path` - The path to the image.
  - Each file-path provided is added to the `images` set in required files stored by Verto. The set of filepaths can be accessed after conversion, see *Step 4: Accessing VertoResult data*.
  - **Note:** If the given link is a relative (a link that doesn't start with `http:`), the link will be rendered with a Django static command. For example, the link `images/example.png` would be rendered as `{% static 'images/example.png' %}` This can be overriden, see the override section below.

4.9.2 Optional Tag Parameters

- `alt` - Description text of the image used when an image is not displayed, or can be read when using a screen reader (for those with reading difficulties).
- `caption` - Boolean flag to indicate whether the image should display a caption.

  - If given as `true`, the the subtitle is the first block of text with the image block.

- `caption-link` (requires caption parameter) - Converts the caption text into a link to the given caption link URL.
- `source` - Adds the text 'Source' under the image with a link to the given source URL. Displays after the caption if a caption is given.
- `alignment` - Valid values are 'left', 'center', or 'right'. Providing one of these values Will add CSS classes to the image for alignment.
- `hover-text` - Additional text to be displayed when the user hovers their cursor over the image (note this won't appear on touch devices so use sparingly).

The default HTML for image is:

```
<div>
<img
{% if file_relative %}
{% autoescape false -%}
src="{{ "{% static '" }}{{ full_file_path }}{{ "' %}" }}"
{%- endautoescape %}
{% else %}
src="{{ full_file_path }}"
{% endif %}
{% if alt %} alt="{{ alt }}" {% endif -%}
{%- if hover_text %} title="{{ hover_text }}" {% endif -%}
 class="{% if alignment == 'left' %}left-align{% elif alignment =='center' %}center-
↪align{% elif alignment =='right' %}right-align{% endif %}">
{% if caption and caption_link -%}
<p><a href="{{ caption_link }}">{{ caption }}</a></p>
{%- elif caption -%}
<p>{{ caption }}</p>
{%- endif -%}
{%- if source_link -%}
<p><a href="{{ source_link }}">Source</a></p>
{%- endif -%}
</div>
```

Using the following example tag:

```
{image file-path="http://placehold.it/350x150" source="https://placehold.it/" alt=
↪"placeholder image" hover-text="This is hover text" alignment="left"}
```

The resulting HTML would be:

```
<div>
<img alt="placeholder image" class="left-align" src="http://placehold.it/350x150"␣
↪title="This is hover text" />
<p><a href="https://placehold.it/">Source</a></p>
</div>
```

4.9.3 Overriding HTML for Images

When overriding the HTML for images, the following Jinja2 placeholders are available:

- `{{ full_file_path }}` - The location for the path to the URL.
- `{{ alt }}` - The alternative text for the image.

- `{{ hover_text }}` - The text to display when the user hovers over the image (see image title attribute).
- `{{ alignment }}` - The location to add extra CSS classes for alignment.
- `{{ caption }}` - The text for the image caption.
- `{{ caption_link }}` - The URL for the caption link.
- `{{ source_link }}` - The URL for the source.
- `{{ file_relative }}` - If the `full_file_path` is a relative link, this is the boolean value `True`, otherwise `False`.

  If `{{ file_relative }}` is `True`, the following placeholders are also available to allow finer control of output of relative images (see *Example 2* below):

  - `{{ file_path }}` - The file path of the image, with file extension removed.
  - `{{ file_extension }}` - The file extension for the image.
  - `{{ file_width_value }}` - If the file name of the image ends in a width suffix (for example: `apple@200px.png`), this is the numerical width value as an integer (in the example before: `200`).
  - `{{ file_width_unit }}` - If the file name of the image ends in a width suffix (for example: `apple@200px.png`), this is the width unit (in the example before: `px`).

Example 1

For example, providing the following HTML:

```
<div class="text-center">
<img alt="placeholder image" src="{{ full_file_path }}" class="rounded img-thumbnail"␣
↪/>
</div>
```

with the following tag:

```
{image alt="dogs" file-path="http://placehold.it/350x150" alt="placeholder image"␣
↪source="https://placehold.it/"}
```

would result in:

```
<div class="text-center">
<img alt="placeholder image" class="rounded img-thumbnail" src="http://placehold.it/
↪350x150" />
</div>
```

Example 2

This is an example of using the `scrset` attribute for relative images.

The following HTML for `image.html`:

```
<img alt="placeholder image"
{% autoescape false -%}
{%- if file_relative %}
{%- if file_width_value -%}
srcset="{% if file_width_value > 200 %}{{ "{% static '" }}{{ file_path }}@200px{{␣
↪file_extension }}{{ "' %}" }} 200w{% endif -%}
{% if file_width_value > 400 %}, {{ "{% static '" }}{{ file_path }}@400px{{ file_
↪extension }}{{ "' %}" }} 400w{% endif -%}
{% if file_width_value > 600 %}, {{ "{% static '" }}{{ file_path }}@600px{{ file_
↪extension }}{{ "' %}" }} 600w{% endif -%}
```

(continues on next page)

(continued from previous page)

```
{% if file_width_value > 800 %}, {{ "{% static '" }}{{ file_path }}@800px{{ file_
↪extension }}{{ "' %}" }} 800w{% endif -%}"
{%- endif -%}
src="{{ "{% static '" }}{{ full_file_path }}{{ "' %}" }}"
{%- else -%}
src="{{ full_file_path }}"
{%- endif -%}
{%- endautoescape -%}>
```

with the following tag:

```
{image alt="placeholder image" file-path="path/to/image@500px.png"}
```

would result in:

```
<img alt="placeholder image" src="{% static 'path/to/image@500px.png' %}" srcset="{%␣
↪static 'path/to/image@200px.png' %} 200w, {% static 'path/to/image@400px.png' %}␣
↪400w" />
```

4.10 Inline Image

Processor name: `image-inline` (see also *Image*)

Note: The inline image tag allows for the use of images inside tables, *etc* without causing style errors. The tag functions almost exactly the same as the `image` tag except for the alignment argument.

You can include an inline image using the following text tag:

```
An inline image: {image-inline alt="alt text" file-path="img/example.png"}.
```

4.10.1 Required Tag Parameters

- `file-path` - The path to the image.
  - Each file-path provided is added to the `images` set in required files stored by Verto. The set of filepaths can be accessed after conversion, see *Step 4: Accessing VertoResult data*.
  - **Note:** If the given link is a relative (a link that doesn't start with `http:`), the link will be rendered with a Django static command. For example, the link `images/example.png` would be rendered as `{% static 'images/example.png' %}` This can be overriden, see the override section below.

4.10.2 Optional Tag Parameters

- `alt` - Description text of the image used when an image is not displayed, or can be read when using a screen reader (for those with reading difficulties).
- `caption` - Lists the given text as a caption under the image.
- `caption-link` (requires caption parameter) - Converts the caption text into a link to the given caption link URL.

- `source` (optional) - Adds the text 'Source' under the image with a link to the given source URL. Displays after the caption if a caption is given.
- `hover-text` - Additional text to be displayed when the user hovers their cursor over the image (note this won't appear on touch devices so use sparingly).

The default HTML for image is:

```
<div>
<img
{% if file_relative %}
{% autoescape false -%}
src="{{ "{% static '" }}{{ full_file_path }}{{ "' %}" }}"
{%- endautoescape %}
{% else %}
src="{{ full_file_path }}"
{% endif %}
{% if alt %} alt="{{ alt }}" {% endif -%}
{%- if hover_text %} title="{{ hover_text }}" {% endif -%}>
{% if caption and caption_link -%}
<p><a href="{{ caption_link }}">{{ caption }}</a></p>
{%- elif caption -%}
<p>{{ caption }}</p>
{%- endif -%}
{%- if source_link -%}
<p><a href="{{ source_link }}">Source</a></p>
{%- endif -%}
</div>
```

Using the following example tag:

```
An inline image: {image-inline alt="alt text" file-path="img/example.png"}.
```

The resulting HTML would be:

```
<p>An inline image: <div>
<img alt="alt text" src="{% static 'img/example.png' %}" />
</div>.</p>
```

4.10.3 Overriding HTML for Images

When overriding the HTML for images, the following Jinja2 placeholders are available:

- `{{ full_file_path }}` - The location for the path to the URL.
- `{{ alt }}` - The alternative text for the image.
- `{{ hover_text }}` - The text to display when the user hovers over the image (see image title attribute).
- `{{ caption }}` - The text for the image caption.
- `{{ caption_link }}` - The URL for the caption link .
- `{{ source_link }}` - The URL for the source .
- `{{ file_relative }}` - If the `full_file_path` is a relative link, this is the boolean value `True`, otherwise `False`.

  If `{{ file_relative }}` is `True`, the following placeholders are also available to allow finer control of output of relative images (see *Example 2* below):

- {{ file_path }} - The file path of the image, with file extension removed.
- {{ file_extension }} - The file extension for the image.
- {{ file_width_value }} - If the file name of the image ends in a width suffix (for example: apple@200px.png), this is the numerical width value as an integer (in the example before: 200).
- {{ file_width_unit }} - If the file name of the image ends in a width suffix (for example: apple@200px.png), this is the width unit (in the example before: px).

Example 1

For example, providing the following HTML:

```
<div class="text-center">
<img alt="{{ alt }}" src="{{ full_file_path }}" class="rounded img-thumbnail" />
</div>
```

with the following tag:

```
An inline image: {image-inline alt="alt text" file-path="http://placehold.it/350x150"␣
↪caption="Placeholder image" source="https://placehold.it/"}.
```

would result in:

```
<p>An inline image: <div class="text-center">
<img alt="alt text" class="rounded img-thumbnail" src="http://placehold.it/350x150" />
</div>.</p>
```

Example 2

This is an example of using the scrset attribute for relative images.

The following HTML for image.html:

```
<div>
<img
{% autoescape false -%}
{%- if file_relative %}
{%- if file_width_value -%}
srcset="{% if file_width_value > 200 %}{{ "{% static '" }}{{ file_path }}@200px{{␣
↪file_extension }}{{ "' %}" }} 200w{% endif -%}
{% if file_width_value > 400 %}, {{ "{% static '" }}{{ file_path }}@400px{{ file_
↪extension }}{{ "' %}" }} 400w{% endif -%}
{% if file_width_value > 600 %}, {{ "{% static '" }}{{ file_path }}@600px{{ file_
↪extension }}{{ "' %}" }} 600w{% endif -%}
{% if file_width_value > 800 %}, {{ "{% static '" }}{{ file_path }}@800px{{ file_
↪extension }}{{ "' %}" }} 800w{% endif -%}"
{%- endif -%}
src="{{ "{% static '" }}{{ full_file_path }}{{ "' %}" }}"
{%- else -%}
src="{{ full_file_path }}"
{%- endif -%}
{%- endautoescape -%}>
</div>
```

with the following tag:

```
An inline image: {image-inline alt="alt text" file-path="path/to/image@500px.png"}.
```

would result in:

```
<p>An inline image: <div>
<img src="{% static 'path/to/image@500px.png' %}" srcset="{% static 'path/to/
↪image@200px.png' %} 200w, {% static 'path/to/image@400px.png' %} 400w" />
</div>.</p>
```

4.11 Interactive

Processor name: `interactive`

The term *interactive* was defined in the Computer Science Field Guide to describe an interactive component of a page. This could be educational game or demostration that is created in HTML, CSS, and JS. Interactives can be small examples to display within the text (for example: animations comparing sorting algorithms) to larger interactives that require a whole page to view (for example: viewing pixels of an image.).

By using the interactive tag, Verto can include or link to an interactive within a page. Verto does not directly include the interactive, but creates Django commands for a Django system to render the interactive or link to interactive as requested.

You can include an interactive using the following text tag:

```
{interactive slug="binary-cards" type="in-page"}
```

4.11.1 Required Tag Parameters

- `slug` - The slug to the interactive to include/link to. This slug is added to the list of interactives in `required_files`.
- `type` - Sets the way the interactive is included in the page. Must be set to one of the following values:
  - `in-page` - The interactive is included in the page by including the HTML (this is the preferred method for including an interactive on a page).
  - `whole-page` - Creates a link to the interactive displayed on a separate page (this is the preferred method for including an interactive on a separate page). The link shows a thumbnail of the interactive with text (the text is set using the `text` parameter). By default, the thumbnail should be a `thumbnail.png` file found within the interactive's img folder.
  - `iframe` - The interactive is included in the page by embedding using an iframe. This is used if the interactive is included multiple times on the page to avoid conflicts in JavaScript/CSS.

4.11.2 Optional Tag Parameters

- `text` - Boolean flag to indicate whether the interactive has custom text to dispaly.
  - If given as `true`, the the subtitle is the first block of text with the interactive block.
  - This is only use with the `whole-page` value.
  - If no text is given, the link uses the text `Click to load {{ slug }}`.
- `parameters` (used with `whole-page` and `iframe` values) - Adds the parameters to interactive link. For example: `digits=5&start=BBBBB`. Do not include the `?` at the start, as this is already included in the output.

- `thumbnail` (optional - used with `whole-page` value) - Displays an alternative thumbnail for the interactive. When not provided, it defaults to `interactives/interactive-slug/img/thumbnail.png`, where `interactive-slug` is the value given for the `slug` tag parameter (see above).
  - If the `thumbnail` value provided is a relative link (a link that doesn't start with `http:`), the link will be rendered with a Django static command. For example, the link:

    ```
    thumbnail-2.png
    ```

    would be rendered as:

    ```
    {% static 'thumbnail-2.png' %}
    ```

  - Each `thumbnail` provided is added to the `images` set in required files stored by Verto. The set of filepaths can be accessed after conversion, see *Step 4: Accessing VertoResult data*.

The default HTML for an interactive is:

```
{%- if type == 'in-page' -%}
<remove>
{{ "{% include 'interactive/" }}{{ slug }}{{ "/index.html' %}" }}
</remove>
{% elif type == 'iframe' -%}
<iframe src="{{ "{% url 'interactive' slug='" }}{{ slug }}{{ "' %}" }}{% if␣
↪parameters %}?{{ parameters }}{% endif %}">
<p>Your browser does not support iframes.</p>
</iframe>
{% elif type == 'whole-page' -%}
<a
   href="{{ "{% url 'interactive' slug='" }}{{ slug }}{{ "' %}" }}{% if parameters %}?
↪{{ parameters }}{% endif %}"
   class="btn btn-interactive-link">
<img class="btn-interactive-link-image"
{% if thumbnail_file_relative %}
{% autoescape false -%}
src="{{ "{% static '" }}{{ thumbnail_file_path }}{{ "' %}" }}"
{%- endautoescape %}
{% else %}
src="{{ thumbnail_file_path }}"
{% endif %}/>
<div class="btn-interactive-whole-page-text">
{% if text -%}
{{ text }}
{% else -%}
Click to load {{ slug }}
{% endif -%}
</div>
</a>
{%- endif -%}
```

4.11.3 Examples

in-page example

Using the following example tag:

```
{interactive slug="binary-cards" type="in-page"}
```

The resulting HTML would be:

```
{% include 'interactive/binary-cards/index.html' %}
```

whole-page example

Using the following example tag:

```
{interactive slug="binary-cards" type="whole-page" parameters="digits=5&start=BBBBB"␣
↪text="true"}

Binary Cards Interactive

{interactive end}
```

The resulting HTML would be:

```
<a class="btn btn-interactive-link" href="{% url 'interactive' slug='binary-cards' %}?
↪digits=5&amp;start=BBBBB">
<img class="btn-interactive-link-image" src="{% static 'interactives/binary-cards/img/
↪thumbnail.png' %}" />
<div class="btn-interactive-whole-page-text">
Binary Cards Interactive
</div>
</a>
```

iframe example

Using the following example tag:

```
{interactive slug="binary-cards" type="iframe" parameters="digits=5&start=BBBBB"}
```

The resulting HTML would be:

```
<iframe src="{% url 'interactive' slug='binary-cards' %}?digits=5&amp;start=BBBBB">
<p>Your browser does not support iframes.</p>
</iframe>
```

4.11.4 Overriding HTML for Interactives

When overriding the HTML for interactives, the following Jinja2 placeholders are available:

- `{{ type }}` - The type of the interactive.
- `{{ slug }}` - The slug of the interactive to include/link to.
- `{{ text }}` - The text to to display to a link to a `whole-page` interactive.
- `{{ parameters }}` - GET parameters to append to the interactive link.
- `{{ thumbnail_file_path }}` - The location for the path to the thumbnail image.
- `{{ thumbnail_file_relative }}` - If the `thumbnail_file_path` is a relative link, this is the boolean value `True`, otherwise `False`.

Example

This example creates a link to `whole-page` interactives without a thumbnail.

For example, providing the following HTML:

```
{% if type == 'in-page' -%}
<div>
{{ "{{% include 'interactive/" }}{{ slug }}{{ "/index.html' %}}" }}
</div>
{% elif type == 'iframe' -%}
<iframe src="{{ "{% url 'interactive' slug='" }}{{ slug }}{{ "' %}" }}{% if␣
↪parameters %}?{{ parameters }}{% endif %}">
<p>Your browser does not support iframes.</p>
</iframe>
{% elif type == 'whole-page' -%}
<a
  href="{{ "{% url 'interactive' slug='" }}{{ slug }}{{ "' %}" }}{% if parameters %}?
↪{{ parameters }}{% endif %}"
  class="btn btn-interactive-link">
<div class="btn-interactive-whole-page-text">
{% if text -%}
{{ text }}
{% else -%}
Click to load {{ slug }}
{% endif -%}
</div>
</a>
{%- endif -%}
```

with the following tag:

```
{interactive slug="binary-cards" type="whole-page" parameters="digits=5&start=BBBBB"␣
↪thumbnail="binarycards.png" text="true"}

Binary Cards Interactive

{interactive end}
```

would result in:

```
<a class="btn btn-interactive-link" href="{% url 'interactive' slug='binary-cards' %}?
↪digits=5&amp;start=BBBBB">
<div class="btn-interactive-whole-page-text">
Binary Cards Interactive
</div>
</a>
```

4.12 Panel

Processor name: `panel`

You can include an panel using the following text tag:

```
{panel type="teacher-note" subtitle="true"}

# Teacher Note

## Curriculum guides for Algorithms

This text is the panel's contents.
```

(continues on next page)

(continued from previous page)

```
{panel end}
```

The title and subtitle are the first Level 1 and Level 2 Headings within the panel. They are within the panel rather than given as parameters to allow translation systems to easily identify which text should be translated.

4.12.1 Required Tag Parameters

- `type` - The type of panel to create.
  - The type is saved as a CCS class (with `panel-` prefix) in the panel (this allows colouring of all the same types of panels).

4.12.2 Optional Tag Parameters

- `subtitle` - Boolean flag to indicate whether the panel should display a subtitle.
  - If given as `true`, then the subtitle is the first Level 2 Heading within the panel.
- `expanded` - A value to state the panel's state:
  - If given as 'true', the panel contains the CSS class `panel-expanded` to state it should be expanded on load.
  - If set to 'always', the panel contains the CSS class `panel-expanded-always` to state it should be expanded at load and cannot be closed.
  - When `expanded` is not given or not a value above, the panel contains no extra CSS classes and be closed on load.

The default HTML for a panel is:

```
<div class='panel panel-{{ type }}{% if expanded == 'always'%} panel-expanded-always{
↪% elif expanded == 'true' %} panel-expanded{% endif %}'>
<div class='panel-header'>
<strong>{{ title }}{% if subtitle %}:</strong> {{ subtitle }}{% else %}</strong>{%␣
↪endif %}
</div>
<div class='panel-body'>
{% autoescape false -%}
{{ content }}
{% endautoescape -%}
</div>
</div>
```

Using the following example tag:

```
{panel type="teacher-note" subtitle="true"}

# Teacher Note

## Curriculum guides for Algorithms

This text is the panel's contents.

{panel end}
```

The resulting HTML would be:

```
<div class="panel panel-teacher-note">
<div class="panel-header">
<strong>Teacher Note:</strong> Curriculum guides for Algorithms
</div>
<div class="panel-body">
<p>This text is the panel's contents.</p>
</div>
</div>
```

4.12.3 Overriding HTML for Panels

When overriding the HTML for panels, the following Jinja2 placeholders are available:

- `{{ type }}` - The type of panel to be created.
- `{{ expanded }}` - Text either set to 'true' or 'always' to state if the panel should be expanded. See parameter description above.
- `{{ title }}` - The provided title text.
- `{{ subtitle }}` - The provided subtitle text.
- `{{ content }}` - The text enclosed by the panel tags.

Example

For example, providing the following HTML:

```
<div class="card">
<div class="card-header">
<h5 class="mb-0">
<strong>{{ title }}{% if subtitle %}:</strong> {{ subtitle }}{% else %}</strong>{%␣
↪endif %}
</h5>
</div>
<div class="collapse{% if expanded == 'always'%} show-always{% elif expanded == 'true
↪' %} show{% endif %}">
<div class="card-block">
{{ content }}
</div>
</div>
</div>
```

with the following tag:

```
{panel type="note" expanded="true"}

# Note

This panel uses Bootstrap 4 card structure.

{panel end}
```

would result in:

```
<div class="card">
<div class="card-header">
<h5 class="mb-0">
<strong>Note</strong>
</h5>
</div>
<div class="collapse show">
<div class="card-block">
<p>This panel uses Bootstrap 4 card structure.</p>
</div>
</div>
</div>
```

4.13 Relative Link

Processor name: relative-link

This processor will find any relative links (a link that doesn't start with http:), and prepend the link with a Django template placeholder. When the resulting HTML is rendered with Django, the Django system can insert the root URL into the template placeholder for correct rendering.

The default HTML for relative links is:

```
<a href='{{ "{{ base_path }}" }}{{ link_path }}{{ link_query }}'>{{ text }}</a>
```

Using the following example tag:

```
Check out this [resource](resource/134).
```

The resulting HTML would be:

```
<p>Check out this <a href="{{ base_path }}resource/134">resource</a>.</p>
```

4.13.1 Overriding HTML for Relative Links

When overriding the HTML for relative links, the following Jinja2 placeholders are available:

- {{ link_path }} - The given link URL.
- {{ link_query }} - The given link query parameters.

Example

For this example, we wish to create HTML to be used in a static site system (not Django). The relative link processor should append the website's URL to each link.

For example, providing the following HTML template:

```
http://www.example.com/{{ link_path }}{{ link_query }}
```

with the following Markdown:

```
Check out this [resource](resource/556).
```

would result in:

```
<p>Check out this <a href="http://www.example.com/resource/556">resource</a>.</p>
```

4.14 Remove Title

Processor name: `remove-title`

This preprocessor runs before any conversion of Markdown and searches for a heading in the first line of provided Markdown text. If a heading is found on the first line, it deletes that line of text. This preprocessor runs after the `save-title` preprocessor if present.

This preprocessor is **not** turned on by default. To use `remove-title`, it needs to be explicity provided in the `processors` parameter when creating the Verto converter, or given in the `update_processors` method (see example below).

Example

With the following text saved in `example_string`:

```
# Example Title

This is a sentence.
```

```
import verto
converter = verto.Verto()
tags = verto.tag_defaults()
tags.add('remove-title')
converter.update_tags(tags)
result = converter.convert(example_string)
```

The `html_string` value in `result` would be:

```
<p>This is a sentence.</p>
```

4.15 Save Title

Tag name: `save-title`

This preprocessor runs before any conversion of Markdown and searches for a heading in the first line of provided Markdown text. If a heading is found on the first line, it saves the text for that heading in the `title` attribute of the `VertoResult` object.

Example

With the following text saved in `example_string`:

```
# Example Title

This is a sentence.
```

```
import verto
converter = verto.Verto()
result = converter.convert(example_string)
print(result.title)
```

would result in:

```
Example Title
```

4.16 Scratch

Processor name: `scratch`

Danger: Scratch blocks require an understanding of the Scratch programming language and how Verto is integrated with other systems. The use of this processor requires co-ordination between authors and developers to achieve the desired functionality.

Note: The following examples assume usage of the fenced code extension, by having `markdown.extensions.fenced_code` in the list of extensions given to Verto.

You can include an image of Scratch blocks using Scratch Block Plugin notation using the following notation:

````
```scratch
when flag clicked
clear
forever
pen down
if <<mouse down?> and <touching [mouse-pointer v]?>> then
switch costume to [button v]
else
add (x position) to [list v]
end
move (foo) steps
turn ccw (9) degrees
```
````

to produce the following image:

The syntax is the same for default Markdown code blocks. The only difference is that Verto handles the content differently due to the `scratch` language set at the start.

Note: This processor also works with syntax introduced by the *fenced_blocks* and/or *codehilite* extensions.

You can test the output of your Scratch block text at scratchblocks.github.io. You can also generate Scratch block text from a published Scratch project at scratchblocks.github.io/generator/.

> **Warning:** Verto doesn't create the Scratch images, but saves data for another system (for example: Django) to create the images. See *Accessing Scratch image data* section below.

The default HTML for scratch blocks is:

```
<div class="scratch-inline-images">
{% for hash in images -%}
<object type="image/svg+xml" data="{% autoescape false -%}{{ "{% static '" }}scratch-
↪blocks-{{ hash }}.svg{{ "' %}" }}{%- endautoescape %}">
</object>
{% endfor -%}
</div>
```

Using the following example tag:

```
```scratch
when flag clicked
clear
forever
pen down
```

(continues on next page)
```

(continued from previous page)

````
if <<mouse down?> and <touching [mouse-pointer v]?>> then
switch costume to [button v]
else
add (x position) to [list v]
end
move (foo) steps
turn ccw (9) degrees
```
````

The resulting HTML would be:

```
<div class="scratch-inline-images">
<object data="{% static 'scratch-blocks-
↪a0f8fcad796864abfacac8bda6e0719813833fd1fca348700abbd040557c1576.svg' %}" type=
↪"image/svg+xml">
</object>
</div>
```

4.16.1 Options

Options that change the output behaviour can be specified by appending them after the language separated by `:`. The options avaliable are:

- `split` - This option turns separate Scratch blocks (which are dictated by an empty line) into separate images.
- `random` - This parameter randomises the order of separate Scratch blocks (which are dictated by an empty line).

For example options can be used like:

````
Scratch is great for kids you can great simple code like:

```scratch:split
when flag clicked
say [Hi]

when flag clicked
say [Hi]
move (foo) steps
turn ccw (9) degrees

when flag clicked
clear
forever
pen down
if <<mouse down?> and <touching [mouse-pointer v]?>> then
switch costume to [button v]
else
add (x position) to [list v]
end
move (foo) steps
turn ccw (9) degrees
```
````

Or for more than one option:
````

````
Scratch is great for kids you can great simple code like:

```scratch:split:random
when flag clicked
say [Hi]

when flag clicked
say [Hi]
move (foo) steps
turn ccw (9) degrees

when flag clicked
clear
forever
pen down
if <<mouse down?> and <touching [mouse-pointer v]?>> then
switch costume to [button v]
else
add (x position) to [list v]
end
move (foo) steps
turn ccw (9) degrees
```
````

4.16.2 Accessing Scratch image data

When Verto encounters a code block with the Scratch language (see example above), it doesn't not generate the image but saves the enclosed text and hash of the text in the `required_files` attribute under the key `scratch_images`.

The following is an example of the result of required files after the parsing two Scratch blocks, where the `scratch_images` key points to a set of `namedtuple` objects containing a `hash` (string) and `text` (string):

```
required_files = {
    "scratch_images": [
        ScratchImageMetaData(
          hash="a3b77ed3c3fa57e43c830e338dc39d292c7def676e0e8f7545972b7da20275da",
          text="when flag clicked\nsay [Hi]\n"
        ),
        ScratchImageMetaData(
          hash="a0f8fcad796864abfacac8bda6e0719813833fd1fca348700abbd040557c1576",
          text="when flag clicked\nclear\nforever\npen down\nif <<mouse down?> and
↪<touching [mouse-pointer v]?>> then\nswitch costume to [button v]\nelse\nadd (x␣
↪position) to [list v]\nend\nmove (foo) steps\nturn ccw (9) degrees\n"
        )
    ]
}
```

The processor replaces the text with an image linked to the expected location of the image.

After Verto has completed a conversion, you will need to retrieve this data from `required_files` and render it to an image in the expected location. The scratchblocks renderer on GitHub allows of rendering to an SVG or PNG.

4.16.3 Overriding HTML for Scratch

When overriding the HTML for Scratch code, the following Jinja2 placeholders are available:

- `{{ images }}` - A list of hashes of the Scratch code-blocks used in the expected filename(s).

Example

For example, providing the following HTML:

```
<div class="scratch-inline-images">
{% for hash in images -%}
<img class="scratch-blocks" src="scratch-blocks-{{ hash }}.svg" onerror="this.src=
↪'scratch-blocks-{{ hash }}.png'"/>
{% endfor -%}
</div>
```

with the following tag:

```
```scratch
when flag clicked
say [Hi]
```
```

would result in:

```
<div class="scratch-inline-images">
<img class="scratch-blocks" onerror="this.src='scratch-blocks-
↪a3b77ed3c3fa57e43c830e338dc39d292c7def676e0e8f7545972b7da20275da.png'" src="scratch-
↪blocks-a3b77ed3c3fa57e43c830e338dc39d292c7def676e0e8f7545972b7da20275da.svg">
</div>
```

4.17 Scratch (Inline)

Processor name: `scratch-inline`

Note: The inline scratch processor works similarly to the default *Scratch* processor except that it matches inline codeblocks instead and requires a colon after the scratch tag.

Danger: Scratch blocks require an understanding of the Scratch programming language and how Verto is integrated with other systems. The use of this processor requires co-ordination between authors and developers to achieve the desired functionality.

Note: The following examples assume usage of the fenced code extension, by having `markdown.extensions.fenced_code` in the list of extensions given to Verto.

You can include an image of Scratch blocks using Scratch Block Plugin notation using the following notation:

```
Lipsum Lorem `scratch:say [Hello] for (2) secs` ipsem.
```

to produce the following image with the `scratch:` stripped:

which is inserted between the paragraph text.

You can test the output of your Scratch block text at scratchblocks.github.io. You can also generate Scratch block text from a published Scratch project at scratchblocks.github.io/generator/.

> **Warning:** Verto doesn't create the Scratch images, but saves data for another system (for example: Django) to create the images. See *Accessing Scratch image data* section in the scratch documentation.

The default HTML for scratch blocks is:

```
<object type="image/svg+xml" data="{% autoescape false -%}{{ "{% static '" }}scratch-
↪blocks-{{ hash }}.svg{{ "' %}" }}{%- endautoescape %}"></object>
```

Using the following example tag:

```
Lipsum Lorem `scratch:say [Hello] for (2) secs` ipsem.
```

The resulting HTML would be:

```
<p>Lipsum Lorem <object data="{% static 'scratch-blocks-
↪3dfa73663a21d295e1e5c1e5583d8d01edd68ec53ad3050597de126076c44ea5.svg' %}" type=
↪"image/svg+xml"></object> ipsem.</p>
```

4.17.1 Overriding HTML for Scratch

When overriding the HTML for Scratch code, the following Jinja2 placeholders are available:

- `{{ hash }}` - The hash of the Scratch code-blocks used in the expected filename.

Example

For example, providing the following HTML:

```
<img class="scratch-blocks" src="scratch-blocks-{{ hash }}.svg" onerror="this.src=
↪'scratch-blocks-{{ hash }}.png'"/>
```

with the following tag:

```
How about some scratch: `scratch:when flag clicked`
```

would result in:

```
<p>How about some scratch: <img class="scratch-blocks" onerror="this.src='scratch-
↪blocks-2f3ea223b778227287b8935bc5d209e25d3e8a25ef46ff85f6c44818159601d7.png'" src=
↪"scratch-blocks-2f3ea223b778227287b8935bc5d209e25d3e8a25ef46ff85f6c44818159601d7.svg
↪" /></p>
```

4.18 Table of Contents

Processor name: `table-of-contents`

Danger: The table-of-contents tag currently requires integration with other systems, use of this tag should accompany supervision of a developer.

You can create a placeholder for a web framework (for example: Django) to insert a table of contents by using the following tag:

```
{table-of-contents}
```

4.18.1 Tag Parameters

There are no required or optional tag parameters for table of contents.

Example

Using the following tag:

```
{table-of-contents}
```

The resulting HTML would be:

```
{{ table_of_contents }}
```

4.18.2 Overriding HTML for Table of Contents

There are no Jinja2 placeholders available when overriding the HTML for table of contents.

The default HTML for table of contents is:

```
<remove>
{{ "{{ table_of_contents }}" }}
</remove>
```

Example

For example, providing the following HTML:

```
<div class="toc">
<strong>Table of Contents</strong>
<ul>
{% raw -%}
{% for heading in headings %}
<li>
<a href="#{{ heading.id }}">{{ heading.name }}</a>
</li>
{% endfor %}
{% endraw -%}
</ul>
</div>
```

with the following tag:

```
{table-of-contents}
```

would result in:

```
<div class="toc">
<strong>Table of Contents</strong>
<ul>
{% for heading in headings %}
<li>
<a href="#{{ heading.id }}">{{ heading.name }}</a>
</li>
{% endfor %}
</ul>
</div>
```

4.19 Video

Processor name: `video`

You can include an video using the following text tag:

```
{video url="https://www.youtube.com/watch?v=dQw4w9WgXcQ"}
```

4.19.1 Required Tag Parameters

- `url` - The video embed URL for the video. Currently only YouTube and Vimeo videos are supported.
  - **For YouTube videos:** Provide the URL in any of the following formats and Verto will automatically create the embed link. YouTube videos have related videos hidden at video end by default.
    - `https://www.youtube.com/watch?v=dQw4w9WgXcQ` - Standard URL
    - `https://youtu.be/dQw4w9WgXcQ` - Shortened URL
    - `https://www.youtube.com/embed/dQw4w9WgXcQ` - Embed URL
  - **For Vimeo videos:** Provide the URL in any of the following formats and Verto will automatically create the embed link.
    - `https://vimeo.com/94502406` - Standard URL
    - `https://player.vimeo.com/video/94502406`- Embed URL

4.19.2 Optional Tag Parameters

- `title` - The title text for the video.

The default HTML for a video is:

```
<div>
{% if title %}
<div class="video-title">{{ title }}</div>
{% endif %}
<iframe allowfullscreen='allowfullscreen' frameborder='0' src='{{ video_url }}'></
↪iframe>
</div>
```

If the URL provided is YouTube video, the video identifier is extracted and is passed through the `video-youtube.html` template:

```
https://www.youtube.com/embed/{{ identifier }}
```

If the URL provided is YouTube video, the video identifier is extracted and is passed through the `video-vimeo.html` template:

```
https://player.vimeo.com/video/{{ identifier }}
```

Example

Using the following example tag:

```
{video url="https://www.youtube.com/watch?v=dQw4w9WgXcQ"}
```

The resulting HTML would be:

```
<div>
<iframe allowfullscreen="allowfullscreen" frameborder="0" src="https://www.youtube.
↪com/embed/dQw4w9WgXcQ"></iframe>
</div>
```

4.19.3 Overriding HTML for Videos

When overriding the HTML for videos, the following Jinja2 placeholders are available:

- `{{ video_url }}` - The URL of the video to be embedded.
- `{{ title }}` - The title of the video.

In the `video-youtube.html`, the following Jinja2 placeholders are available:

- `{{ youtube_identifier }}` - The identifer of the YouTube video to be embedded.

In the `video-vimeo.html`, the following Jinja2 placeholders are available:

- `{{ vimeo_identifier }}` - The identifer of the Vimeo video to be embedded.

Example

For example, providing the following HTML for `video-youtube.html`:

```
https://www.youtube.com/embed/{{ identifier }}?autoplay=1&loop=1
```

with the following tag:

```
{video url="https://youtu.be/Uw0PISu2pog"}
```

would result in:

```
<div>
<iframe allowfullscreen="allowfullscreen" frameborder="0" src="https://www.youtube.
↪com/embed/Uw0PISu2pog"></iframe>
</div>
```

CHAPTER 5

Implicit Processors

The following pages cover processors that do not require explicit use when authoring Markdown:

5.1 Jinja

Processor name: `jinja`

The `jinja` processor is a post-processor that is used to undo HTML escaping on Jinja/Django statements (i.e. `{% ... %}`) that may be present in the document for further processing of the document after conversion. This processor does not do any sanitizing of the Jinja/Django statements and therefore should not be used on untrusted input without sanitation before or after the Verto conversion. This processor should be used with the *Conditional* as the default HTML-template produces Jinja statements.

For example the following document with an if statement:

```
<div>
{% if thing &lt; object %}
<p>
A paragraph explaining the &lt; operation.
</p>
{% endif %}
</div>
```

Verto will unescape the Jinja/Django statement and produce the following output:

```
<div>
{% if thing < object %}
<p>
A paragraph explaining the &lt; operation.
</p>
{% endif %}
</div>
```

5.2 Ordered List (OListProcessor)

Processor name: `olist`

This processor overwrites functionality provided by the Python Markdown package allowing for the use of container tags such as *Panel* and *Boxed Text* tags within an ordered list while also providing the same features of the sane lists extension.

Indentation is important when creating ordered lists and is expected to be by default `4` spaces for inner content and `2` spaces after the number. Authors should follow the following example where indentation spaces are replaced with the `•` character.

```
1.••Text here.
••••More text here.

2.••Text here.

••••1.••List within a list.
••••••••More text here.
```

5.2.1 Sane Lists

The Sane Lists extension alters the markdown lists such that types are not allowed to mix. This extension is implemented in Verto by default and therefore does not need to be added as an extension. This means the following markdown:

```
1.  Ordered list item.
*   Not a separate item.
```

produces the output:

```
<ol>
  <li>
    Ordered list item.
*   Not a separate item.
  </li>
</ol>
```

5.3 Remove

Processor name: `remove`

The `remove` processor is a post-processor that searches the document for remove HTML-elements (i.e. `<remove>...</remove>`) and removes them from the document leaving the content unchanged. This is useful when creating HTML-templates as they can be used to add multiple siblings to a parent element that are not valid HTML, allowing the document to be parsed as a valid HTML-document up until their removal.

Note: The `remove` processor does not remove the content between the remove element tags, but instead only removes the tag itself.

For example the *Conditional* processors default HTML template, as follows, does not produce valid HTML and so is placed within a remove element so that Verto can add it to the element tree.

```
<remove>
{% autoescape false -%}
{{ "{% if " }}{{ if_expression }}{{ " %}" }}
{{ if_content }}
{% for elif_expression, elif_content in elifs.items() -%}
{{ "{% elif " }}{{ elif_expression }}{{ " %}" }}
{{ elif_content }}
{% endfor -%}
{% if has_else -%}
{{ "{% else %}" }}
{{ else_content }}
{% endif -%}
{{ "{% endif %}" }}
{% endautoescape -%}
</remove>
```

Therefore a Markdown document like:

```
<div>
<remove>
<p>Content in here.</p>
{{ Django Variable }}
<p>Something about the variable.</p>
</remove>
</div>
```

When parsed with Verto will produce the output:

```
<div>
<p>Content in here.</p>
{{ Django Variable }}
<p>Something about the variable.</p>
</div>
```

5.4 Scratch Compatibility

Processor name: `scratch-compatibility`

The `scratch-compatibility` processor is a pre-processor that is enabled by the *Scratch* processor when the `codehilite` and `fenced_code` extensions are enabled.

When both `codehilite` and `fenced_code` extensions are enabled the `fenced_code` extension modifies the fenced code-blocks by using methods from the `codehilite` extension before stashing them to be place in later in the document. The `scratch-compatibility` processor is therefore needed to stash the fenced code-blocks before `fenced_code` so that they can be processed properly by the *Scratch* processor later.

Note: We consider the `codehilite` and `fenced_code` extensions a bad way of writing extensions as the output of one dramatically changes depending on if the other is active.

We believe that an extension like these should produce predictable output and handle compatibility through inputs.

For example if the following Markdown document is processed using both the `codehilite` and `fenced_code` extensions

````
Scratch is great for kids you can create simple code like:

```scratch
when flag clicked
say [Hi]
```
```scratch
when flag clicked
say [Hi]
move (foo) steps
turn ccw (9) degrees
```

```scratch
when flag clicked
clear
forever
pen down

if <<mouse down?> and <touching [mouse-pointer v]?>> then
switch costume to [button v]
else
add (x position) to [list v]
end

move (foo) steps
turn ccw (9) degrees
```
````

Verto will produce the following output (which is the same as the `scratch` processor would expect):

```
<p>Scratch is great for kids you can create simple code like:</p>
<div class="scratch-inline-images">
<object data="{% static 'scratch-blocks-
↪a3b77ed3c3fa57e43c830e338dc39d292c7def676e0e8f7545972b7da20275da.svg' %}" type=
↪"image/svg+xml">
</object>
</div>
<div class="scratch-inline-images">
<object data="{% static 'scratch-blocks-
↪cd6d9a0d464bb8f5eec1e6fc9a4e33378a64ebfce7c6198794ead614962f38e5.svg' %}" type=
↪"image/svg+xml">
</object>
</div>
<div class="scratch-inline-images">
<object data="{% static 'scratch-blocks-
↪8e8a2129c3cecf32101248439961735fc1d45793fadc56e2575673f63d42b9fb.svg' %}" type=
↪"image/svg+xml">
</object>
</div>
```

5.5 Style

Processor name: `style`

The `style` processor is a pre-processor that checks that the input Markdown to enforce style rules. These rules

include:

- Processor tags have empty lines before and after.
- Processor tags do not share a line with other text.

An example of a valid document follows:

```
This is valid.

{panel}

This is valid.

{panel end}

This is valid.
```

5.5.1 Error Example(s)

Note: The examples covered in this section are invalid and will raise errors.

The following examples raise errors because the processor tags do not have empty lines before and after.

```
This is not valid
{panel}
This is not valid
{panel end}
This is not valid
```

```
{panel}
This is not valid
{panel end}
```

```
This is not valid
{panel}

{panel end}
```

```
{panel}

{panel end}
This is not valid
```

The following examples raise errors because the processor tags share a line with other text.

```
{panel} This is not valid

{comment this is not valid} {panel end}
```

```
{panel}

{glossary-link this is not valid} {panel end}
```

5.6 Unordered List (UListProcessor)

Processor name: `ulist`

This processor overwrites functionality provided by the Python Markdown package allowing for the use of container tags such as *Panel* and *Boxed Text* tags within an unordered list while also providing the same features of the sane lists extension.

Indentation is important when creating unordered lists and is expected to be `4` spaces by default for inner content and `3` spaces after the bullet. Authors should follow the following example where indentation spaces are replaced with the • character.

```
*•••Text here.
••••More text here.

*•••Text here.

••••*•••List within a list.
••••••••More text here.
```

See *Ordered List (OListProcessor)* for details on Sane Lists.

CHAPTER 6

Extensions

As Verto is an extension of the Python Markdown package, you should be able to include any extension for the original package. This page details using extensions with Verto, plus listing a few useful extensions that we recommend.

To include a package, pass a list of extensions to the `extensions` keyword when creating the Verto object. For example:

```
extra_extensions = [
    'markdown.extensions.fenced_code',
    'markdown.extensions.codehilite',
    'markdown.extensions.sane_lists',
    mdx_math.MathExtension(enable_dollar_delimiter=True)
]
converter = Verto(extensions=extra_extensions)
```

A list of extensions for the Markdown package can be found in their official documentation.

6.1 Math

Math can be rendered by including the Python Markdown Math package, and passing it through to Verto as an extension to use. A guide on how to install and use the extension can be found in the package's README file.

6.2 Fenced Code

The Fenced Code Blocks extension adds a secondary way to define code blocks, which overcomes a few limitations of the indented code blocks. More details on the Fenced Code extension can be found in the Fenced Code documentation.

6.3 Code Highlighting

The CodeHilite extension adds code/syntax highlighting to standard Python-Markdown code blocks using Pygments. More details on the CodeHilite extension can be found in the CodeHilite documentation.

6.4 Sane Lists

The Sane Lists extension alters the behavior of the Markdown List syntax to be less surprising. More details on the Sane Lists extension can be found in the Sane Lists documentation.

CHAPTER 7

Contributing to Verto

Welcome to the Verto developer community! We have spent many months developing this project, and we would love for you to get involved! The following documentation has been written to help you get a grasp on how Verto is pieced together to make contributing as simple and straight forward as possible. Please feel free to fix bugs and/or suggest new features and improvements to the system (or the docs) by making a pull request.

Verto was created to be used by two much larger projects (the CS Unplugged and CS Field Guide websites) as the Markdown to HTML converter. The tags we chose are designed to allow authors of these two projects to easily write material without technical elements getting in the way. It is therefore important to us that Verto remains as simple and robust as possible, please keep this in mind if you decide to work on Verto with us.

The git repository for Verto can be found here, jump in and take a look around!

Note: The two projects that Verto was developed for are Django projects, so you may come across HTML (in templates, test cases etc) that contains Django syntax.

For example, below is the expected output for a for a image tag test:

```
<div>
 <img alt="Lipsum" class="" src="{% static 'computer-studying-turing-test.png' %}"/>
</div>
```

This does not mean that Verto is only suitable for Django projects, as it's just a matter of customising the relevant HTML templates.

7.1 Issue Reporting and Bug Fixes

If you come across a bug in Verto, please report it on the repo issue tracker.

If you choose to fix the bug for us, consider adding the relevant tests to the Test Suite (detailed further down this page) to help us catch any future bugs.

7.2 The Code Base

If you would like to contribute to Verto, create a fork of the repository.

7.2.1 Overview

Before reading this section, make sure you have read *how to use* (or even better, have already used Verto!).

Terminology

There are a couple of terms we use when describing Verto to become familiar with:

- **Tag**

  This refers to the custom markdown syntax that Verto processes.

  For example:

  ```
  {comment this will be removed by the converter}

  {image file-path="img/totally-real-image.png" alt="process me"}
  ```

  are examples of the `comment` and `image` tags in Verto.

- **Processor**

  This refers to the class that is responsible for converting a specific tag. For example, `RelativeLinkPattern` is the processor for internal links.

Project Structure

Below is a basic overview of the project structure:

```
├── docs/
├── verto/
|   ├── errors/
│   ├── html-templates/
│   ├── VertoExtension.py
│   ├── Verto.py
│   ├── processor-info.json
│   ├── processors/
│   ├── tests/
│   └── utils/
├── requirements.txt
├── requirements-dev.txt
└── setup.py
```

The items of interest are:

- `Verto()` - The convertor object itself. This is what a user will use to create a Verto converter, and what is used to define a custom processor list, custom html templates and custom Markdown Extensions to use.
- `VertoResult()` (found in `Verto.py`) - The object returned by `Verto()` containing:
  - Converted html string
  - Title

  - Required files (images, interactives, scratch images, page scripts)
  - Heading tree
  - Required glossary terms
- `VertoExtension()` - This is the main class of the project, and inherits the `Extension` class from Markdown. It loads all of the processor information, loads the template files and clears and populates the attributes to be returned by the `VertoResult` object.
- `processor-info.json` - Every processor is listed in this file, and will at least contain a class determining whether it is custom or generic, where custom processors will have a pattern to match it's corresponding tag. Most will also define required and optional parameters, these correspond to arguments in the tag's html template.
- `processors/` - There is a different processor for each tag. A processor uses it's corresponding description loaded from `processor-info.json` to find matches in the text, and uses the given arguments in the matched tag to populate and output it's html template.
- `html-templates/` - The html templates (using the Jinja2 template engine) with variable arguments to be populated by processors.
- `errors/` - Contains all the errors exposed by the Verto module. Where an Error is an exception that is caused by user input. New errors should be created in here inheriting from the base `Error` class.
- `utils/` - Contains classes and methods not necessarily unique to Verto that are useful in any sub-module. This includes slugify handlers, html parsers and serialisers, and other utilities. The utilities should be used over external libraries as they are purposely built because of: compatibility reasons, licensing restrictions, and/or unavailability of require features.
- `tests/` - explained in the Test Suite section further down the page.

It is important to note that Verto is not just a Markdown Extension, it is a wrapper for Python Markdown. `VertoExtension` **is** an extension for Python Markdown. We have created a wrapper because we wanted to not only convert text, but also extract information from the text as it was being converted (recall `VertoResult()` listed above).

7.2.2 Creating a New Processor

There are two ways distinctly different ways to create a new processor. The simplist way is to make use of the provided generic processors and define the new processor in the `processor-info.json` file, while more complex processors reqiure additional source code. Complex processors should be considered when custom functionality is required that cannot be achieved with generic processors.

In all cases new processors should:

- Be thoroughly tested (see the section on *testing*)
- Have clear and accurate documentation. See the docs on other processors for the preferred format. Your docs should include:
  - An example of the tag in markdown
  - Required parameters
  - Optional parameters
  - Examples
  - Examples of overriding the html

We recommend writing documentation and test cases before you even write the processor itself as this will give you a clear idea of how a processor in Verto should behave.

Generic Processors

There are two types of generic processors:

- tags (`generic_tag`): which match `{<processor_name> <args>}` in the markdown text replacing with the given html-template.
- containers (`generic_container`): which are a pair of tags which capture the content between the tags for the html-template. A generic container's opening tag specifies the arguments, while the closing tag only has the `end` argument allowing for the content to contain generic containers.

To create a new processor that uses the generic processors the processor must be added to the `processor-info.json` file and an associated html-template must be created. The template must only have one root level node after rendering.

How to make a JSON Definition

The json description of a generic processor must contain the attributes:

- `class`: Either `generic_tag` or `generic_container` for a generic processor.
- `arguments`: An object describing arguments passed to the tag.
- `template_parameters`: An object describing template parameters.
- (Optional) `template_name`: A custom name for the html-template to use. Defaults to the processor name otherwise.
- (Optional) `tag_argument`: The text given at the beginning of a tag (e.g. the tag argument for `{image file-path="example.png"}` is `image`). This is only necessary for processors with different names sharing the same resources (e.g. both `image-container` and `image-tag` share the `tag_argument`).

The `argument` parameter is a dictionary (or object) containing argument name, argument-info pairs. Where the argument-info contains the attributes:

- `required`: `true` if the argument must be set or `false` otherwise.
- (Optional) `dependencies`: A list of argument-names that must also be set if this argument is used.
- (Optional) `values`: A list of values of which the argument may take. If the argument does not have the value in this list the ArgumentValueError exception is raised.

These arguments are transformed for use in the html-template by the `template_parameters` attribute. This attribute is similar to the `argument` attribute by containing parameter name, parameter-info pairs. Where the parameter-info contains the attributes:

- `argument`: The name of the argument to retrieve the value of to use/transform into the parameter value.
- (Optional) `default`: The value the parameter defaults to if the argument is not given otherwise defaults to `None`.
- (Optional) `transform`: The name of the transform to modify the argument value by or defaults to null for no transformation. The avaliable transforms are detailed below.
- (Optional) `transform_condition`: A function that takes the context after parameters are set but before transformation (The transformations are done in order they appear in the json document). If the function returns `True` then the transformation is applied.

For a generic container type processor the `argument` of the parameter may be `content` which is the captured content between the start and end tags.

The set of currently avaliable transformations for the `transform` attribute are:

- `str.lower`: Converts the string into a lowercase version.
- `str.upper`: Converts the string into an UPPERCASE version.
- `relative_file_link`: Applies the relative-file-link html-template to the argument.

Examples

A generic tag processor, is a simple single line tag that uses the given arguments as parameters to an html template. An example of a processor that uses the generic tag processor is the *button-link* processor which is described in the json as:

```
"button-link": {
  "class": "generic_tag",
  "arguments": {
    "link": {
      "required": true,
      "dependencies": []
    },
    "text": {
      "required": true,
      "dependencies": []
    },
    "file": {
      "required": false,
      "dependencies": []
    }
  },
  "template_parameters": {
    "file": {
      "argument": "file",
      "transform": "str.lower",
      "default": "no"
    },
    "link": {
      "argument": "link",
      "transform": "relative_file_link",
      "transform_condition": "lambda context: context['file'] == 'yes'"
    },
    "text": {
      "argument": "text",
      "transform": null
    }
  }
}
```

And has the following html-template:

```
<a class='button' href='{{ link }}'>{{ text }}</a>
```

This enables the following markdown:

```
{button-link link="http://www.google.com" text="Visit Google"}
```

To generate the output:

```
<a class="button" href="http://www.google.com">Visit Google</a>
```

A generic container processor, a pair of matching tags where one opens the container and one closes the container. The start tag gives the arguments as parameters to an html template. The end tag is used to capture the content between the tags to be used as an additional parameter to the html template. An example of a processor that uses the generic container processor is the *boxed-text* processor which is described in the json as:

```
"boxed-text": {
    "class": "generic_container",
    "arguments": {
      "indented": {
        "required": false,
        "dependencies": []
      }
    },
    "template_name": "boxed-text",
    "template_parameters": {
      "indented": {
        "argument": "indented",
        "transform": "str.lower"
      },
      "text": {
        "argument": "content",
        "transform": null
      }
    }
}
```

And has the following html-template:

```
<div class='boxed-text{% if indented == "yes" %} boxed-text-indented{% endif %}{% if
↪type %} boxed-text-{{ type }}{% endif %}'>
{% autoescape false -%}
{{ text }}
{% endautoescape -%}
</div>
```

This enables the following markdown:

```
{boxed-text}

**Computer Science Report for 2.44**

Put your introduction to what bits are here.

{boxed-text end}
```

To generate the output:

```
<div class="boxed-text">
<p><strong>Computer Science Report for 2.44</strong></p>
<p>Put your introduction to what bits are here.</p>
</div>
```

Custom Processors

To create a custom processor, the `class` attribute of the processor in the `processor-info.json` file must be `"custom"`. A good place to start when programming a new processor is the Extension API page of the Python Markdown docs, and you can also read the source code itself.

There are several different kinds of processors in Python Markdown, each serving a slightly different purpose. We recommend reading the API docs to determine which processor best suits your purpose. Verto currently makes use of `preprocessor`, `blockprocessor`, `inlinepattern`, `treeprocessor` and `postprocessor`, but you are welcome to use another type of processor if it better suits the task.

The order of the processors matters and is defined when each processor is added to the `OrderedDict` in `VertoExtension.py`.

Each processor should try to be as independent of every other processor as possible. Sometimes this is not possible, and in this case compatibility should occur in the processor that happens last (i.e. the downstream processor). That is output should be consistent based on input, not the other way round (e.g. `codehilite` and `fenced_code`).

The logic for each processor belongs in the `processors/` directory, and there are several other places where processors details need to be listed. These are:

- The processor's relevant information (regex pattern, required parameters etc) should be included in `processor-info.json`.
- If it should be a default processor, it should be added to the frozenset of `DEFAULT_PROCESSORS` in `Verto.py`.
- The relevant list in `extendMarkdown()` in `VertoExtension.py` (see OrderedDict in the Markdown API docs for manipulating processor order).
- The processor's template should be added to `html-templates` using the Jinja2 template engine syntax for variable parameters. A valid template will only have one root level node after rendering, if more root nodes are necessary the remove tag can be used as the root node which will be removed later.
- Any errors should have appropriate classes in the `errors\` directory, they should be well described by their class name such that for an expert knows immediately what to do to resolve the issue, otherwise a message should be used to describe the exact causation of the error for a novice.

7.3 The Test Suite

To start the test suite:

```
$ python3 -m verto.tests.start_tests
```

This will execute the Smoke, System and then Unit tests.

There are several arguments that can be used with this command to skip particular tests (`--no_smoke`, `--no_system` and `--no_unit`).

7.3.1 Test Suite Structure

We are now focusing on our project structure diagram from earlier:

```
└── verto/
    └── tests/
        ├── assets/
        ├── BaseTest.py
        ├── ConfigurationTest.py
        ├── ProcessorTest.py
        ├── SmokeTests.py
        └── start_tests.py
```

The items of interest are:

- `BaseTest())` - This class is inherited by nearly every other test file, and contains a method to read a given test asset file.
- `ConfigurationTest()` - This is the test class for testing different configurations of `Verto()` (e.g. using a custom list of processors and/or custom html templates). This class inherits the `BaseTest` class.
- `ProcessorTest.py` - This is the class inherited by all processor test classes. It contains several useful methods for testing processors, including those for loading templates and processor info.
- `SmokeDocsTest()` and `SmokeFileTest()` - These are the two classes for smoke testing.
- `start_tests.py` - This is the file that is executed in order to run each of the three types of tests (Smoke, System and Unit). Every new test class must be added to the relevant section of this file.
- `assets/` - This directory contains a sub directory for every test class that loads external assets (e.g. test input files).

7.3.2 Adding Tests

When writing a new test function, it is important that the method name is as descriptive as possible. The method name should also be prefixed with `test_` as the test suite will only execute methods with this prefix.

If you have added a new processor to `Verto`, then a corresponding test suite also needs to be added. This test suite should be added to the `unit_suite()` function in `start_tests.py`. The section below has details on how to write a processor test.

7.3.3 Processor Tests

All processor tests inherit from the `ProcessorTest` class. Processors should create a `Mock()` object, which will contain the bare minimum for the processor to be run (it's HTML template and properties loaded from `processor-info.json`), i.e. there is no reason for it to know about properties of the other processors.

A test method will typically follow the same sequence of steps:

1. Retrieve the test string (there is a `read_test_file()` method provided by the `ProcessorTest` class)
2. Confirm there are (not) matches to the regex in the test string
3. Convert the test string using the `verto_extension` (provided by the `SetUp()` method in `ProcessorTest`)
4. Load the expected converted result
5. Check the converted result is the same as the expected result

Testing Assets

Most tests will load an asset file. This file contains example Markdown text (and therefore has a `.md` extension). For comparing the converted result of this Markdown file with it's expected output, a corresponding "expected" file should be created. The expected file should have the same name as the corresponding test file, with `expected` appended to the file name (and has a `.html` extension).

These asset files should be placed in `verto/tests/assets/<processor-name>/`.

For example:

```
verto/tests/assets/boxed-text/no_boxed_text.md
verto/tests/assets/boxed-text/no_boxed_text_expected.html
```

Note:

- Asset files should have discriptive names, and in many cases will have the same name as the method they are used in.

7.4 Creating a release

This is our current process for creating and publishing a Verto release. This can only be performed by repository administrators.

1. Create a release branch. Checkout to this branch.

```
$ git checkout develop
$ git pull
$ git checkout -b release/0.7.0
```

2. Update the version number[1] within `verto/__init__.py`.
3. Check test suite for errors, and fix any issues that arise, or log an issue.
4. Detail the changes in `docs/source/changelog.rst`.
5. Complete the release branch. Be sure to tag the release with the version number for creating the release on GitHub.

```
$ git checkout master
$ git pull
$ git merge --no-ff release/0.7.0
$ git tag -a 0.7.0
$ git push
$ git push --tags
```

6. Upload the release to PyPI

```
$ rm -r dist/
$ python3 -m pip install --user --upgrade setuptools wheel
$ python3 setup.py sdist bdist_wheel
$ python3 -m pip install --user --upgrade twine
$ twine upload dist/*
```

7. Merge the release into develop

```
$ git checkout develop
$ git pull
$ git merge --no-ff release/0.7.0
```

8. Create the release on GitHub on the tagged commit.
9. Delete the release branch.

[1] We follow Semantic Versioning for our numbering system. The number is used by `setup.py` to tell PyPI which version is being uploaded or `pip` which version is installed, and also used during the documentation build to number the version of Verto it was built from.

7.5 Other notes

7.5.1 Why are requirements split across two files?

The `requirements.txt` file is used to specify required dependencies for Verto (and are automatically installed as dependencies when installed via `pip`). The `requirements-dev.txt` file is used to specify dependencies for developing Verto (for example, documentation generators).

7.5.2 Why are all dependency versions pinned?

Yes, this is not considered the best practice by the Python Software Foundation in their packaging guide. However pinning dependencies ensure we control over each Verto release, following the logic of Vincent Driessen.

CHAPTER 8

Changelog

8.1 0.11.0

- Replaces `custom_argument_rules` configuration parameter with `settings` parameter. The `custom_argument_rules` parameter is now set within `settings` under a new name. The settings also allow configuring if thumbnail images for interactives are added to the required images set. More information on these settings can be found in the *Using Verto* documentation.
- Improve documentation on how to create a release.
- Dependency updates:
  - Update `setuptools` from 40.4.3 to 41.0.1
  - Update `sphinx_rtd_theme` from 0.4.1 to 0.4.3.
  - Update `coverage` from 4.5.1 to 4.5.3.
  - Update `Jinja2` from 2.10 to 2.10.1.
  - Update `python-slugify` from 1.2.6 to 3.0.2.

8.2 0.10.0

- Add title parameter to *Video* processor for translations.
- Dependency updates:
  - Update `setuptools` to 40.4.3
  - Update `sphinx` to 1.8.1

8.3 0.9.3

- Resolve issues of broken package due to unpinned dependencies.
- Remove automated deployment to PyPI.

8.4 0.9.2

- Broken release, removed from PyPI.

8.5 0.9.1

- Broken release, removed from PyPI.

8.6 0.9.0

- Add *Blockquote* processor for customising block quote style.
- Added CodeCov to repo
- Dependency updates:
  - Update `python-slugify` to 1.2.6
  - Update `sphinx` to 1.8.0

8.7 0.8.0

- Modify *Interactive* processor for translating text, by required text between start and end tags for whole page interactives.
- Modify Verto parameters available on creation to allow modification of default required parameters for each processor.
- Dependency updates:
  - Update `setuptools` to 40.2.0.
  - Update `sphinx` to 1.7.7.
  - Update `sphinx_rtx_theme` to 0.4.1.

8.8 0.7.4

- Modify *Interactive* processor to use `slug` rather than `name` to identify interactives
- Modify *Video* processor template for youtube videos
- Modify *Boxed Text* processor to have optional type parameter
- Update style error message to inclue line numbers

- Dependency updates:
  - Update `python-slugify` to 1.2.5.
  - Update `setuptools` to 39.1.0.
  - Update `sphinx` to 1.6.6.
  - Update `sphinx_rtx_theme` to 0.3.0.

8.9 0.7.3

- Modified *Interactive* processor to change interactive template depending on the type of file path given for the thumbnail image of whole page interactives (external or internal) as well as changed the default path for the thumbnail.

8.10 0.7.2

- Fix bug where *Panel* processor does not handle punctuation characters in titles and subtitles.
- Dependency updates:
  - Update `markdown` to 2.6.11.
  - Update `setuptools` to 38.4.0.
  - Update `sphinx` to 1.6.6.

8.11 0.7.1

- *Save Title* and *Remove Title* processors now only search first line.

8.12 0.7.0

- *Relative Link* processor will now handle query parameters.
- Modify *Panel* processor for translating subtitles, by requiring subtitle text as second level heading.
- Modify *Image* processor for translating captions, by requiring caption text between start and end tags.
- Modify *Image* processor to allow finer control of output, in particular when dealing with image with width values.
- Add new tag configuration value `tag_argument` to override tag name.
- Dependency updates:
  - Update `markdown` to 2.6.10.
  - Update `Jinja2` to 2.10.
  - Update `setuptools` to 38.2.5.
  - Update `sphinx` to 1.6.5.

8.13 0.6.1

Fixes:

- Adds all interactives to required files.
- Typo in interactive tag documentation.

8.14 0.6.0

Features:

- Added *Inline Image* processor, intended for use in tables.
- Added *Scratch (Inline)* processor for inline scratch support.

Fixes:

- Removed `beautifulsoup4` dependency.
- Typo in VertoResult documentation (*heading_root* -> *heading_tree*).

8.15 0.5.3

In this hotfix Verto result data for unique identifiers and required files is now only cleared when explicitly told. Result data that is per document such as title and heading tree are cleared per conversion.

Fixes:

- Remove implicit Beautify processor, fixing white-spacing issues.
- All terms are added to glossary correctly now.

8.16 0.5.2

Fixes:

- Verto container tags, are now supported in markdown lists.

8.17 0.5.1

Fixes:

- Verto tags and custom tags, are now support embedding into markdown lists.

8.18 0.5.0

Fixes:

- A new more descriptive error when an argument is given and not readable.

- Custom HTML string parsing has been implemented, allowing for correct parsing of HTML and XHTML in templates.

Documentation:

- Basic example in README.
- New contributing documentation.
- Fixed reference to incorrect file in the image processor documentation.
- Added new documentation for implicit processors.

8.19 0.4.1

Fixes:

- pypi configuration fixes.
- pyup configuration to use develop branch.
- Improved asset file loading for deployed package.

8.20 0.4.0

Fourth prerelease of the Verto converter. (The project was renamed to Verto from Kordac in release.)

Adds support for the following processors:

- *Embed iframe*
- *Interactive*
- *Heading*
- *Scratch*
- *Table of Contents*

Features:

- The *Scratch* processor supports `split` and `random` options.

Fixes:

- Scratch blocks work with other extensions.
- Glossary slugs are now added to the output of Verto.
- Processors are now ordered correctly.

8.21 0.3.1

Fixes:

- Updated documentation and changelog.

8.22 0.3.0

Third prerelease of the Verto converter.

Adds support for the following processors:

- *Heading*
- *Embed iframe*
- *Interactive*
- *Scratch*
- *Table of Contents*

Fixes:

- Verto now orders tags correctly in the markdown pipeline.
- System tests for multiple calls to Verto and for multi-line templates.
- Glossary tags now correctly store slugs for the Verto result as per documentation.

8.23 0.2.0

Second prerelease of the Verto converter.

Adds support for the following processors:

- *Button Link*
- *Conditional*
- *Glossary Link*
- *Video*

Adds basic support for Code Climate.

Fixes:

- Verto default processors can be accessed via a static method.
- Required and optional arguments are now explicitly matched against input.
- Made tag parameters consistently use dashes as separators.
- Tests for previous processors now explicitly test matches.
- Tests fail on docs build failures and warnings.

8.24 0.1.0

Initial prerelease of Verto converter.

Includes the following processors:

- *Boxed Text*
- *Comment*
- *Image*

- *Panel*
- *Relative Link*
- *Remove Title*
- *Save Title*

Index

Symbols

__init__() (*verto.Verto method*), 9

C

clear_saved_data() (*verto.Verto method*), 9
clear_templates() (*verto.Verto method*), 9
convert() (*verto.Verto method*), 9

H

heading_tree (*verto.Verto.VertoResult attribute*), 10
html_string (*verto.Verto.VertoResult attribute*), 10

P

processor_defaults() (*verto.Verto static method*), 8, 9

R

required_files (*verto.Verto.VertoResult attribute*), 10
required_glossary_terms (*verto.Verto.VertoResult attribute*), 10

T

title (*verto.Verto.VertoResult attribute*), 10

U

update_processors() (*verto.Verto method*), 8, 9
update_templates() (*verto.Verto method*), 9, 10

V

Verto (*class in verto*), 9
VertoResult (*class in verto.Verto*), 10