AsciiSpec Userguide

Table of Contents

List of Figures

This userguide covers the basics of common and practical AsciiDoc syntax along with basic tips for using AsciiSpec for documentation.

Chapter 1. Converting Documents with AsciiSpec

To convert this document with AsciiSpec, `cd` to the docs directory and run:

```
asciispec userguide.adoc
```

This will use the default backend and convert the `userguide.adoc` sample document to a HTML file called `userguide.html`.

Using CLI options

To convert to HTML using a different CSS stylesheet, we can pass document attributes via the command line using the `-a` flag:

```
asciispec -a stylesheet=mystyle.css mydoc.adoc
```

To convert to docbook, use the `-b docbook` flag:

```
asciispec -b docbook userguide.adoc
```

Most built-in CLI parameters are described in the help page by running `asciispec -h` and in further detail in the CLI Options section of the AsciiDoctor user manual.

1.1. PDF

Apache FOP (Formatting Objects Processor) is required for higher quality **PDF** generation. A fork with custom PDF styling and syntax highlighting can be found at the following location:

https://github.com/NumberFour/asciidoctor-fopub

1. Clone the Asciidoctor Fopub repository

   ```
   git clone https://github.com/NumberFour/asciidoctor-fopub.git
   ```

2. Add the `asciidoctor-fopub/bin` directory to your shell profile:

   ```
   export PATH=$PATH:~/path/to/asciidoctor-fopub/bin/
   ```

3. Confirm successful install using an empty `fopub` command.

   ```
   $ fopub
   ~/path/to/asciidoctor-fopub/bin/fopub: You must specify a DocBook v4.5 or DocBook v5 XML source file as the first command argument
   ```

4. Convert XML to PDF using the following:

   ```
   fopub myfile.xml
   ```

1.2. Configuration File

AsciiSpec processors with configurable target URLs have to be set up by means of a configuration file. For this case, `config.adoc` can be copied to the location of the source document and used as a template. The following line of code must be included at the top of your source document:

```
include::config.adoc[]
```

> To configure a specific processor, see the Chapter 3, *AsciiSpec Processors* section below.

Chapter 2. AsciiDoctor Syntax

In order that we understand the use of AsciiSpec processors, it's important to know the context in which they function. This section provides a brief overview of how an AsciiDoc document is structured. The following list is a simplified overview of the AsciiDoctor AST:

`Document`	The document contains `Sections` and `Blocks` that make up the document and holds the document attributes.
`Section`	Models sections in the document and dictates the structure of the `Document` tree.
`Blocks`	Content within a `Section`, differentiated by **context** such as 'paragraph' or 'image'.
`Lists`, `Tables`, `ListItems` …	Nested content within a `Block`. Can also themselves be `Blocks`.

2.1. Blocks

Usage

```
[quote]
Before I came here I was confused about this subject.
Having listened to your lecture I am still confused.
But on a higher level.
```

Blocks are content in a section with styles or `contexts` such as paragraphs, source listings, images, etc. Square brackets `[]` are used to indicate the style of the block that follows and an empty line will indicate that the block has finished. All plain text of one or more lines will be parsed as a `block` with the 'paragraph' style by default, therefore:

```
It was the best of times..

// Is the same as writing the following:

[paragraph]
It was the best of times..
```

To style a block with a source `listing` context, we use `[source]` as with this example:

```
[source]
export public class Fibonacci {
 public seq()  {
```

Output:

```
export public class Fibonacci {
 public seq()  {
```

2.1.1. Optional Attributes

```
.Fibonacci.n4js
[source,n4js]
----
export public class Fibonacci {
 public seq()  {

      var arr = [];
// etc...
----
```

In the first line we have demonstrated how to add a *title* to a block. This is done using a full stop followed by the title (`Fibonacci.n4js`). A title can be added in this way to many different block types by default.

In line 2 of the example above we declare some attributes for the block. The first attribute is to set the block context as `source` and the second attribute is the listing language - `n4js`.

Notice the use of four hyphens to delimit the block: `----` this indicates to the parser where the block begins and ends. The listing block can then also include the empty line:

Output:

Fibonacci.n4js

```
export public class Fibonacci {
 public seq()  {

      var arr = [];
// etc...
```

2.2. Block Macro

Usage:

`toc::[]`

Block macros are used to create a block member in a document. The above example creates a table of contents block at that position in the document (to enable this feature, see setting document attributes below).

`Block` vs. `Block Macro`

The difference here is that with a block macro, all parameters that dictate how the block is rendered are contained within the macro delcaration. The already-existing lines of AsciiDoc source that follow the macro are not formatted or changed by its use.

As with **`Blocks`**, we must also prepend and append the **`Block Macro`** with an empty line. The following is an example of a block macro to insert a new block with the **`image`** context followed by the resulting output:

Source:

```
The following image macro is not rendered, +
it is considered the last line of this pargraph block. +
image::images/logo.png[]

Leaving an empty line before and after the image block macro +
will create a block as expected:

image::images/logo.png[]

beginning of next block...
```

Output

The following image macro is not rendered,
it is considered the last line of this pargraph block.
image::images/logo.png[]

Leaving an empty line before and after the image block macro
will create a block as expected:

beginning of next block…

2.3. Inline Macro

Inline macros are similar to block macros except that the macro is replaced by inline content. The syntax is different in that we use a single colon `:` instead of two `::`

```
We can simply insert a logo image:images/logo.png[Logo] directly into our paragraph...
```

We can simply insert a logo AsciiSpec directly into our paragraph…

Optional Attributes

In the above example, we have included some optional attributes in the square brackets that close the inline macro. The first attribute is the 'Alt Text' of the image, followed by the width and height of the image. The same method of passing attributes can be applied to the block macro above e.g.

`image::images/logo.png[Logo,15,18]`.

2.4. Document Attributes & Variables

Usage:

```
:attribute: value

{attribute}
```

Setting document attributes is done by adding an attribute entry line as `:attribute: value` above. Variables are declared using `{}` curly brackets and can be used for substitutions. Attributes can be inserted anywhere in a document unless they are specific *header attributes* as described in the next section. Here are two examples of setting an attribute via an attribute entry line and using inline shorthand:

Source	Output

`By attribute entry line:`
 	
`:country: Spain`
`We should travel to {country}!`
 	
`Attributes can be set inline too:`
`We should {set:country:France} travel to {country}!`
	By attribute entry line:
	
	We should travel to {country}!
	
	Attributes can be set inline too: We should travel to France!

An example of a common document attribute is `imagesdir` which specifies the images directory. `imagesdir` is empty by default, therefore, `image:a.jpg[]` will look for `a.jpg` in the same directory as the source document.

```
:imagesdir: images
```

Setting `imagesdir` as above saves the time of typing out the full path every time we use the `image:[]` macro.
Usually this is done once per document but can be used multiple times:

Setting Document Attributes

```
:imagesdir: images/icons

image:github.png[]

image:jira.png[]

:imagesdir: images

image:logo.png[]
```

Output:

2.4.1. Header Attributes

A header starts with a document title followed by two optional lines defining author and revision information.
Finally, document-wide settings are defined by means of *header attributes* :

```
= AsciiSpec Documentation
Brian Thomas Smith
First Draft
:toc: right
```

An example header attribute is `:toc:` which sets the position of the Table of Contents in the destination document. The above example right-aligns the Table of Contents. Another option is to enable the use of the `toc::[]` block macro to insert a Table of Contents block in any section:

```
= AsciiSpec Documentation
Brian Thomas Smith
First Draft
:toc: macro

// A Table of Contents is rendered here by default

== Section two

toc::[] // But will be rendered here instead
```

A full table of the available built-in document attributes, see the Built-in Attributes section in the AsciiDoctor User Manual.

2.5. Tables

Table blocks are delimited by a character (usually a pipe `|`) and three equals symbols (`|===`);

```
|===
| Hello | world
|===
```

Hello	world

A different character can be used to delimit cells by substituting the pipe `|` with the separator you wish to use. A comma can be used exactly as above to separate cells in the following way:

```
,===
, Hello , World
,===
```

Using commas in this way can provide an easy solution to including CSV values (`include::mydata.csv[]`) into a table without having to reformat the included document.

Formatting tables:

```
[cols="h,d"]
|===
| Backend 3+h| Description

| html (or html5) 3+| HTML5, styled with CSS3 (default).
| pdf 3+| PDF, a portable document format. Requires the asciidoctor-pdf gem.
|===
```

In the above table, formatting attributes **3+** are used. The ^ caret symbol is used to centre-align the text and **3+** indicates that the cell spans three consecutive columns.

Backend	Description
html (or html5)	HTML5, styled with CSS3 (default).
pdf	PDF, a portable document format. Requires the asciidoctor-pdf gem.

A full overview of the possibilities to create complex tables can be found in the tables section of the User Manual.

2.6. Blocks

Content can be formatted in blocks as with the following:

```
[quote,Enrico Fermi,Notes on Quantum Mechanics (1954)]
Before I came here I was confused about this subject.
Having listened to your lecture I am still confused.
But on a higher level.
```

> Before I came here I was confused about this subject. Having listened to your lecture I am still confused. But on a higher level.
>
> — Enrico Fermi *Notes on Quantum Mechanics (1954)*

For a full list of block types see the AsciiDoctor User Manual: built-in blocks summary.

2.6.1. Titles & attributes

Adding a title to a block of content is done by adding a fullstop followed by the title text in the line previous to the block. We can add further attributes which are relevant to the type of block. In the case of a `[verse]` block, we can attribute the author and the source of the content separated with commas like so: `[verse, Carl Sagan, Cosmos]`.

```
.Deep Thought of the Day
[verse, Carl Sagan, Cosmos: A Personal Voyage]
If you want to make an apple pie from scratch, you must first create the universe.
```

The above is rendered as follows:

> If you want to make an apple pie from scratch, you must first create the universe.
>
> — Carl Sagan *Cosmos: A Personal Voyage*

2.7. Source Code

There are a few easy ways of including source code in our documents. Listing blocks are defined using `[source]` and are delimited with `----`. In our case, we have added a custom N4JS language theme in the `scripts` folder:

```
.Custom N4JS highlighting theme
[source,n4js]
 export public class Fibonacci {
 public seq()  {
      var arr = [];
          var a = 0;
          var b = 1;
...
```

Custom N4JS highlighting theme

```
 export public class Fibonacci {
 public seq()  {
      var arr = [];
          var a = 0;
          var b = 1;
...
```

We can specify the language we want to highlight as the first attribute in the source block. In the following, we have written `[source,html]`:

```
<!DOCTYPE html>
<title>Title</title>
<style>body {width: 500px;}</style>
<script type="application/javascript">
  function $init() {return true;}
</script
```

2.8. Admonition Blocks

A useful feature built-in to AsciiDoctor is the inclusion of admonition blocks. By default, the following admonition blocks are available; `TIP`, `NOTE`, `IMPORTANT`, `CAUTION`, `WARNING`. They render as with the `WARNING` block below, except with different icons.

Source:

```
`WARNING:` Don't divide by zero. In ordinary arithmetic, the expression has no meaning, as there is no number which, multiplied by 0..
```

Output:

Don’t divide by zero. In ordinary arithmetic, the expression has no meaning, as there is no number which, multiplied by 0..

2.9. Media

The above video is embedded with the following syntax: `video::3NjQ9b3pgIg[youtube,800,600]`

```
.A lovely screenshot
image::images/logo.png[]
```

Figure 2.1. A lovely screenshot

Chapter 3. AsciiSpec Processors

For custom AsciiSpec features, see the AsciiSpec Processors index.

Chapter 4. General Tips

4.1. Colons

When learning AsciiDoc syntax, it can be confusing whether to use one or two colons for certain macros. The rule is as follows:

Type	Syntax	Example
Inline	`:`	We can include this `image:test.png[]` inline
Block	`::`	The following Table of Contents `toc::[]` cannot be declared inline.

Chapter 5. Resources

AsciiSpec Docs - NumberFour AsciiSpec Documentation
AsciiDoc Syntax Quick Reference - Covers most standard formatting needs.
AsciiDoctor User Manual - Reference Manual detailing document attributes, conversion settings, extended features etc.

5.1. Sublime Text packages

OmniMarkup Preview - Serves a live preview to a browser for realtime editing.
OmniMarkup Custom Fork - A custom fork that provides styles and syntax highlighting aligned with AsciiSpec.
Sublime Text AsciiDoc Package - Syntax highlighting, snippets, keymaps and more.

5.2. Document Converters

Kramdown - Easily convert GitHub Flavoured Markdown (`.md`) to AsciiDoc (`.adoc`).
Pandoc - A universal document converter.