Asciidoc Documentation Skeleton

Preface

The **Asciidoc Document Skeleton** a helper for setting up a base file and folder structure for **multi-chapter** AsciiDoc projects based on *Jekyll* and *J1 Template*. You need **both** to use this skeleton to create AsciiDoc documents from it.

The Asciidoc Document Skeleton is fully **relocateable** and can be placed in any subfolder of your Jekyll site.

The skeleton can be used to create HTML output (backend **html5**) and PDF output (backend **pdf**) as well. *J1 Template* comes with the full support of *Asciidoctor PDF*, a Ruby-based implementation for *Asciidoctor* based on the PDF converter *Prawn*.

Happy Jekylling!

Introduction

The skeleton of type **documentation** (book) can be used to create HTML output (backend **html5**) for websites and **PDF** output (backend **pdf**) for offline reading as well. *J1 Template* comes with the full support of *Asciidoctor PDF*, a Ruby-based add-on for *Asciidoctor* using the Ruby PDF writer *Prawn*.

If you have plans to convert documents of type type documentation (book) into PDF, make sure you have enabled the Ruby **Gem** `asciidoctor-pdf` with your projects's Gemfile. To make the converter usable, the plugin has to be enabled with your site config file `_config.yml` in section **PLUGIN configuration** as well.

Prawn is a pure Ruby PDF generation library that provides a lot of great functionality while trying to remain simple by providing a reasonable performance.

Some of the important features of the **PDF** writer *Prawn* are:

- Vector drawing support, including lines, polygons, curves, ellipses, etc.
- Extensive text rendering support, including flowing text nd limited inline formatting options.
- Support for both PDF builtin fonts as well as embedded TrueType fonts
- A variety of low level tools for basic layout needs, including a simple grid system
- PNG and JPG image embedding, with flexible scaling options
- Security features including encryption and password protection
- Tools for rendering repeatable content (i.e headers, footers, and page numbers)
- Comprehensive internationalization features, including full support for UTF-8 based fonts, right-to-left text rendering, fallback font support, and extension points for customizable text wrapping.
- Support for PDF outlines for document navigation

Converting the Skeleton

The Asciidoc skeleton **documentation** (book) is fully **relocateable** and can be placed in any subfolder of your Jekyll site for **HTML** output. For **PDF** output, a single variable `BASE_PATH` has to be set for your environment.

See the **batch** files `a2p.bat` for *Windows* (`cmd.exe`) or `a2p.sh` used on *Unix* or *Linux* OS for the shell (bash).

Asciidoctor PDF is a native PDF converter for AsciiDoc that plugs into the PDF backend. It bypasses the requirement to generate an intermediary format such as DocBook, Apache FO, or LaTeX. Instead, you can use Asciidoctor PDF to convert your documents directly from AsciiDoc to PDF. The aim of this library is to take the pain out of creating PDF documents from AsciiDoc.

Asciidoctor PDF is made possible by an amazing Ruby gem named *Prawn*. And what a gem it is! Prawn is a nimble PDF writer for *Ruby*. More important, it's a hackable platform that offers both high level APIs for the most common needs and low level APIs for bending the document model to accommodate special circumstances.

Asciidoctor PDF is currently *alpha* software. While the converter handles most AsciiDoc content, there's still work needed to fill in gaps where conversion is incomplete, incorrect or not implemented. See the milestone v1.5.0 in the Issue tracker for details.

With *Prawn*, you can write text, draw lines and shapes and place images *anywhere* on the page and add as much color as you like. In addition, it brings a fluent API and aggressive code re-use to the printable document space.

Here's an example that demonstrates how to use *Prawn* to create a basic PDF document.

Create a basic PDF document using Prawn

```
require 'prawn'

Prawn::Document.generate 'output.pdf' do
  text 'Hello, PDF creation!'
end
```

It's that easy. And that's just the beginning.

Prawn is the *killer library* for PDF generation we've needed to close this critical gap in *Asciidoctor*. It absolutely takes the pain out of creating printable documents. Picking up from there, *Asciidoctor PDF* takes the pain out of creating PDF documents *from AsciiDoc*.

Highlights

- Direct AsciiDoc to PDF conversion
- SVG support
- PDF document outline (i.e., bookmarks)
- Table of contents page(s)
- Document metadata (title, authors, subject, keywords, etc)
- Internal cross reference links
- Syntax highlighting with Rouge, Pygments, or CodeRay
- Page numbering
- Customizable running content (header and footer)
- “Keep together” blocks (i.e., page breaks avoided in certain block content)
- Orphaned section titles avoided
- Autofit verbatim blocks (as permitted by base_font_size_min setting)
- Table border settings honored
- Font-based icons
- Custom (TTF) fonts
- Double-sided printing mode (margins alternate on recto and verso pages)

Prerequisites

All that's needed is *Ruby* (1.9.3 or above; 2.3.x recommended) and a few *Ruby* gems, which we explain how to install in the next section.

To check if you have *Ruby* available, use the `ruby` command to query the version installed:

```
ruby --version
```

By default, *Asciidoctor PDF* automatically installs the latest version of *Prawn* if you don't already have *Prawn* installed. This will fail on older versions of *Ruby*. To workaround, you need to install certain dependencies first.

Starting with *Prawn* 2.0.0, *Prawn* requires Ruby >= 2.0.0 during installation. Therefore, to use *Asciidoctor PDF* with Ruby 1.9.3, you must first explicitly install the following dependencies to lock the versions:

```
gem install prawn --version 1.3.0
gem install addressable --version 2.4.0
gem install prawn-svg --version 0.21.0
gem install prawn-templates --version 0.0.3
```

You can then proceed with installation of *Asciidoctor PDF* on Ruby 1.9.3.

Starting with *Prawn* 2.2.0, *Prawn* requires Ruby >= 2.1.0 during installation. Therefore, to use *Asciidoctor PDF* with Ruby 2.0.0, you must first explicitly install the following dependencies to lock the versions:

```
gem install prawn --version 2.1.0
gem install prawn-svg --version 0.26.0
gem install prawn-templates --version 0.0.4
```

For all other versions of *Ruby*, you can simply install the *Asciidoctor PDF* gem. It will transitively install the required dependencies.

System Encoding

Asciidoctor assumes you're using UTF-8 encoding. To minimize encoding problems, make sure the default encoding of your system is set to UTF-8.

If you're using a non-English Windows environment, the default encoding of your system may not be UTF-8. As a result, you may get an `Encoding::UndefinedConversionError` or other encoding issues when invoking *Asciidoctor*. To solve these problems, we recommend at least changing the active code page in your console to UTF-8.

```
chcp 65001
```

Once you make this change, all your Unicode headaches will be behind you.

Getting Started

You can get *Asciidoctor PDF* by installing the published gem or running the code from source.

Install the Published Gem

Asciidoctor PDF is published as a pre-release on RubyGems.org. First, make sure you have satisfied the prerequisites. Then, you can install the published gem using the following command:

```
gem install asciidoctor-pdf --pre
```

If you want to syntax highlight source listings, you'll also want to install Rouge, Pygments, or CodeRay. Choose one (or more) of the following:

Rouge (preferred)

```
gem install rouge
```

Pygments

```
gem install pygments.rb
```

CodeRay

```
gem install coderay
```

You then activate syntax highlighting for a given document by adding the `source-highlighter` attribute to the document header (Rouge shown):

```
:source-highlighter: rouge
```

Assuming all the required gems install properly, verify you can run the `asciidoctor-pdf` script:

```
asciidoctor-pdf -v
```

If you see the version of *Asciidoctor PDF* printed, you're ready to use *Asciidoctor PDF*.

Let's grab an AsciiDoc document to distill and start putting *Asciidoctor PDF* to use!

An example AsciiDoc document

If you don't already have an AsciiDoc document, you can use the **basic_example.adoc** file found in the examples directory of this project.

It's time to convert the AsciiDoc document directly to PDF.

Convert AsciiDoc to PDF

Converting to PDF is a simple as running the `asciidoctor-pdf` script using *Ruby* and passing our AsciiDoc document as the first argument.

You'll need the `rouge` gem installed to run this example since it uses the `source-highlighter` attribute with the value of `rouge`.

```
asciidoctor-pdf 000_basic_example.adoc
```

This command is just a **shorthand** way of running:

```
asciidoctor -r asciidoctor-pdf -b pdf 000_basic_example.adoc
```

The `asciidoctor-pdf` command just saves you from having to remember all those flags. That's why we created it.

When the script completes, you should see the file **000_basic_example.pdf** in the same directory. Open the **000_basic_example.pdf** file with a PDF viewer to see the result.

[PDF document rendered in a PDF viewer] | */assets/images/pages/asciidoc_skeletons/example-pdf-screenshot.jpg*

Figure 1. Example PDF document rendered in a PDF viewer

You're also encouraged to try converting the documents in the examples directory to see more of what *Asciidoctor PDF* can do. The pain of the DocBook toolchain should be melting away from now.

Themes

The layout and styling of the PDF is driven by a YAML configuration file. To learn how the theming system works and how to create and apply custom themes, refer to the *Asciidoctor PDF* Theme Guide. You can use the built-in theme files, which you can find in the **data/themes** directory, as examples.

Support for Non-Latin Languages

Asciidoctor can process the full range of characters in the UTF-8 character set. That means you can write your document in any language, save the file with UTF-8 encoding, and expect *Asciidoctor* to convert the text properly. However, you may notice that certain characters for certain languages, such as Chinese, are missing in the PDF. Read on to find out why and how to address it.

If you're writing in a non-Latin language, you need to use a dedicated theme that provides the necessary fonts. For example, to produce a PDF from a document written in a CJK language such as Chinese, you need to use a CJK theme. You can get such a theme by installing the `asciidoctor-pdf-cjk-kai_gen_gothic` gem. See the asciidoctor-pdf-cjk-kai_gen_gothic project for detailed instructions.

Using a dedicated theme is necessary because PDF is a "bring your own font" kind of system. In other words, the font you provide must provide glyphs for all the characters. There's no one font that supports all the worlds languages (though some, like Noto Serif, certainly come close). Even if there were such a font, bundling that font with the main gem would make the package enormous. It would also severely limit the style choices in the default theme, which targets Latin-based languages.

Therefore, we're taking the strategy of creating separate dedicated theme gems that target each language family, such as CJK. The base theme for CJK languages is provided by the asciidoctro-pdf-cjk project and a concrete implementation provided by the asciidoctor-pdf-cjk-kai_gen_gothic project that's based on the kai_gen_gothic font. Of course, you're free to follow this model and create your own theme gem that uses fonts of your choice.

Language Overview

The theme language in **Asciidoctor PDF** is based on the YAML data format and incorporates many concepts from CSS and SASS. Therefore, if you have a background in web design, the theme language should be immediately familiar to you.

Like CSS, themes have both selectors and properties. Selectors are the component you want to style. The properties are the style elements of that component that can be styled. All selector names are implicit (e.g., `heading`), so you customize the theme primarily by manipulating pre-defined property values (e.g., `font_size`).

The theme language in **Asciidoctor PDF** supports a limited subset of the properties from CSS. Some of these properties have different names from those found in CSS.

- Underscores (_) can be used in place of hyphens (-) for all property names in the theme language.
- Instead of separate properties for font weight and font style, the theme language combines these settings in the `font_style` property (allowed values: `normal`, `bold`, `italic` and `bold_italic`).
- The `text_align` property from CSS is the `align` property in the theme language.
- The `color` property from CSS is the `font_color` property in the theme language.

A theme (or style) is described in a YAML-based data format and stored in a dedicated theme file. YAML is a human-friendly data format that resembles CSS and helps to describe the theme. The theme language adds some extra features to YAML, such as variables, basic math, measurements and color values. These enhancements will be explained in detail in later sections.

The theme file must be named *<name>-theme.yml*, where `<name>` is the name of the theme. Here's an example of a basic theme file:

basic-theme.yml

```
page:
  layout:             portrait
  margin:             [0.75in, 1in, 0.75in, 1in]
  size:               Letter
base:
  font_color:         '#333333'
  font_family:        Times-Roman
  font_size:          12
  line_height_length: 17
  line_height:        $base_line_height_length / $base_font_size
vertical_spacing:     $base_line_height_length
heading:
  font_color:         '#262626'
  font_size:          17
  font_style:         bold
  line_height:        1.2
  margin_bottom:      $vertical_spacing
link:
  font_color:         '#002FA7'
outline_list:
  indent:             $base_font_size * 1.5
```

When creating a new theme, you only have to define the keys you want to override from the base theme, which is loaded prior to loading your custom theme. All the available keys are documented in section **Keys**. The converter uses the information from the theme map to help construct the PDF.

Instead of creating a theme from scratch, another option is to download the default-theme.yml file from the source repository. Save the file using a unique name (e.g. *custom-theme.yml*) and start hacking on it.

Alternatively, you can snag the file from your local installation using the following command:

```
ASCIIDOCTOR_PDF_DIR=`gem contents asciidoctor-pdf --show-install-dir`; \
  cp "$ASCIIDOCTOR_PDF_DIR/data/themes/default-theme.yml" custom-theme.yml
```

Keys may be nested to an arbitrary depth to eliminate redundant prefixes (an approach inspired by SASS). Once the theme is loaded, all keys are flattened into a single map of qualified keys. Nesting is simply a shorthand way of organizing the keys. In the end, a theme is just a map of key/value pairs.

Nested keys are adjoined to their parent key with an underscore (_). This means the selector part (e.g., `link`) is combined with the property name (e.g., `font_color`) into a single, qualified key (e.g., `link_font_color`).

For example, let's assume we want to set the base (i.e., global) font size and color. These keys may be written longhand:

```
base_font_color:      '#333333'
base_font_family:     Times-Roman
base_font_size:       12
```

Or, to avoid having to type the prefix `base_` multiple times, the keys may be written hierarchically:

```
base:
  font_color:         '#333333'
  font_family:        Times-Roman
  font_size:          12
```

Or even:

```
base:
  font:
    color:            '#333333'
    family:           Times-Roman
    size:             12
```

Each level of nesting must be indented by two more spaces of indentation than the parent level. Also note the presence of the colon after each key name.

Values

The value of a key may be one of the following types:

- String
  - Font family name (e.g., Roboto)
  - Font style (normal, bold, italic, bold_italic)
  - Alignment (left, center, right, justify)
  - Color as hex string (e.g., #ffffff)
  - Image path
  - Enumerated type (where specified)
  - Text content (where specified)
- Null (clears any previously assigned value)
  - *empty* (i.e., no value specified)
  - null
  - ~
- Number (integer or float) with optional units (default unit is points)
- Array
  - Color as RGB array (e.g., [51, 51, 51])
  - Color CMYK array (e.g., [50, 100, 0, 0])
  - Margin (e.g., [1in, 1in, 1in, 1in])
  - Padding (e.g., [1in, 1in, 1in, 1in])
- Variable reference (e.g. `$base_font_color`)
- Math expression

Note that keys almost always require a value of a specific type, as documented in section **Keys**.

Inheritance

Like CSS, inheritance is a principle feature in the **Asciidoctor PDF** theme language. For many of the properties, if a key is not specified, the key inherits the value applied to the parent content in the content hierarchy. This behavior saves you from having to specify properties unless you want to override the inherited value.

The following keys are inherited:

- font_family
- font_color
- font_size
- font_style
- text_transform
- line_height (currently some exceptions)
- margin_bottom (if not specified, defaults to `$vertical_spacing`)

Heading Inheritance

Headings inherit starting from a specific heading level (e.g., `heading_h2_font_size`), then to the heading category (e.g., `heading_font_size`), then directly to the base value (e.g., `base_font_size`). Any setting from an enclosing context, such as a sidebar, is skipped.

Variables

To save you from having to type the same value in your theme over and over, or to allow you to base one value on another, the theme language supports variables. Variables consist of the key name preceded by a dollar sign (`$`) (e.g., `$base_font_size`). Any qualified key that has already been defined can be referenced in the value of another key. (In order words, as soon as the key is assigned, it's available to be used as a variable).

Variables are defined from top to bottom (i.e., in document order). Therefore, a variable must be defined before it is referenced. In other words, the path the variable refers to must be **above** the usage of that variable.

For example, once the following line is processed,

```
base:
  font_color:           '#333333'
```

the variable `$base_font_color` will be available for use in subsequent lines and will resolve to `#333333`. Let's say you want to make the font color of the sidebar title the same as the heading font color. Just assign the value `$heading_font_color` to the `$sidebar_title_font_color`.

```
heading:
  font_color:           '#191919'
sidebar:
  title:
    font_color:         $heading_font_color
```

You can also use variables in math expressions to use one value to build another. This is commonly done to set font sizes proportionally.It also makes it easy to test different values very quickly.

```
base:
  font_size:            12
  font_size_large:      $base_font_size * 1.25
  font_size_small:      $base_font_size * 0.85
```

We'll cover more about math expressions later.

Custom Variables

You can define arbitrary key names to make custom variables. This is one way to group reusable values at the top of your theme file. If you are going to do this, it's recommended that you organize the keys under a custom namespace, such as `brand`.

For instance, here's how you can define your brand colors:

```
brand:
  primary:            '#E0162B' ①
  secondary:          '#FFFFFF' ②
  alert:              '0052A5'  ③
```

① To align with CSS, you may add a # in front of the hex color value. A YAML preprocessor is used to ensure the value is not treated as a comment as it would normally be the case in YAML.

② You may put quotes around the CSS-style hex value to make it friendly to a YAML editor or validation tool.

③ The leading # on a hex value is entirely optional. However, we recommend that you always use either a leading # or surrounding quotes (or both) to prevent YAML from mangling the value.

You can now use these custom variables later in the theme file:

```
base:
  font_color:         $brand_primary
```

Math Expressions

The theme language supports basic math operations to support calculated values. Like programming languages, multiple and divide take precedence over add and subtract.

The following table lists the supported operations and the corresponding operator for each.

Table 1. Math Expressions

Operation	Operator
multiply	*
divide	/
add	+
subtract	-

Operators must always be surrounded by a space on either side (e.g., 2 + 2, not 2+2).

Here's an example of a math expression with fixed values.

```
conum:
  line_height:        4 / 3
```

Variables may be used in place of numbers anywhere in the expression:

```
base:
  font_size:          12
  font_size_large:    $base_font_size * 1.25
```

Values used in a math expression are automatically coerced to a float value before the operation. If the result of the expression is an integer, the value is coerced to an integer afterwards.

Numeric values less than 1 must have a 0 before the decimal point (e.g., 0.85).

The theme language also supports several functions for rounding the result of a math expression. The following functions may be used if they surround the whole value or expression for a key.

round(…)
: Rounds the number to the nearest half integer.

floor(…)
: Rounds the number up to the next integer.

ceil(…)
: Rounds the number down the previous integer.

You might use these functions in font size calculations so that you get more exact values.

```
base:
  font_size:          12.5
  font_size_large:    ceil($base_font_size * 1.25)
```

Measurement Units

Several of the keys require a value in points (pt), the unit of measure for the PDF canvas. A point is defined as 1/72 of an inch. If you specify a number without any units, the units defaults to pt.

However, us humans like to think in real world units like inches (in), centimeters (cm), or millimeters (mm). You can let the theme do this conversion for you automatically by adding a unit notation next to any number.

The following units are supported:

Table 2. Measurement Units

Unit	Suffix
Centimeter	cm
Inches	in
Millimeter	mm
Percentage[1]	%, vw, or vh
Points	pt (default)

A percentage with the % unit is calculated relative to the width or height of the content area. Viewport-relative percentages (vw or vh units) are calculated as a percentage of the page width or height, respectively. Currently, percentage units can only be used for placing elements on the title page or for setting the width of a block image.

Numbers with more than two digits should be written as a float (e.g., 100.0), a math expression (e.g, 1 * 100), or with a unit (e.g., 100pt). Otherwise, the value may be misinterpreted as a hex color (e.g., '100') and could cause the converter to crash.

Here's an example of how you can use inches to define the page margins:

```
page:
  margin:             [0.75in, 1in, 0.75in, 1in]
```

The order of elements in a measurement array is the same as it is in CSS:

1. top
2. right
3. bottom
4. left

Alignments

The align subkey is used to align text and images within the parent container.

Text Alignments

Text can be aligned as follows:

- left
- center
- right
- justify (stretched to each edge)

Image Alignments

Images can be aligned as follows:

- left
- center
- right

Font Styles

In most cases, whereever you can specify a custom font family, you can also specify a font style. These two settings are combined to locate the font to # use.

The following font styles are recognized:

- normal (no style)
- italic
- bold
- bold_italic

Text Transforms

Many places where font properties can be specified, a case transformation can be applied to the text. The following transforms are recognized:

- uppercase
- lowercase
- none (clears an inherited value)

Since Ruby 2.4, Ruby has built-in support for transforming the case of any letter defined by Unicode. If you're using Ruby < 2.4, and the text you want to transform contains

characters beyond the Basic Latin character set (e.g., an accented character), you must install either the `activesupport` or the `unicode` gem in order for those characters to be transformed.

```
gem install activesupport
```

or

```
gem install unicode
```

Colors

The theme language supports color values in three formats:

Hex
: A string of 3 or 6 characters with an optional leading #, optional surrounding quotes or both.

RGB
: An array of numeric values ranging from 0 to 255.

CMYK
: An array of numeric values ranging from 0 to 1 or from 0% to 100%.

Transparent
: The special value `transparent` indicates that a color should not be used.

Hex

The hex color value is likely most familiar to web developers. The value must be either 3 or 6 characters (case insensitive) with an optional leading hash (#), optional surrounding quotes or both.

To align with CSS, you may add a # in front of the hex color value. A YAML preprocessor is used to ensure the value is not treated as a comment as it would normally be the case in YAML.

You also may put quotes around the CSS-style hex value to make it friendly to a YAML editor or validation tool. In this case, the leading # on a hex value is entirely optional.

Regardless, we recommend that you always use either a leading # or surrounding quotes (or both) to prevent YAML from mangling the value.

The following are all **equivalent** values for the color **red**:

#ff0000	#FF0000	ff0000	FF0000	#f00	#F00	f00	F00

Here's how a hex color value appears in the theme file:

```
base:
  font_color:         '#ff0000'
```

RGB

An RGB array value must be three numbers ranging from 0 to 255. The values must be separated by commas and be surrounded by square brackets.

An RGB array is automatically converted to a hex string internally, so there's no difference between ff0000 and [255, 0, 0].

Here's how to specify the color red in RGB:

- [255, 0, 0]

Here's how a RGB color value appears in the theme file:

```
base:
  font_color:         [255, 0, 0]
```

CMYK

A CMYK array value must be four numbers ranging from 0 and 1 or from 0% to 100%. The values must be separated by commas and be surrounded by square brackets.

Unlike the RGB array, the CMYK array *is not* converted to a hex string internally. PDF has native support for CMYK colors, so you can preserve the original color values in the final PDF.

Here's how to specify the color red in CMYK:

- [0, 0.99, 1, 0]
- [0, 99%, 100%, 0]

Here's how a CMYK color value appears in the theme file:

```
base:
  font_color:         [0, 0.99, 1, 0]
```

Transparent

It's possible to specify no color by assigning the special value `transparent`, as shown here:

```
base:
  background_color:   transparent
```

Images

An image is specified either as a bare image path or as an inline image macro as found in the AsciiDoc syntax. Images are currently resolved relative to the value of the `pdf-stylesdir` attribute.

The following image types (and corresponding file extensions) are supported:

- PNG (.jpg)
- JPEG (.jpg)

- SVG (.svg)

The GIF format (.gif) is not supported.

Here's how an image is specified in the theme file as a bare image path:

```
title_page:
  background_image:   title-cover.jpg
```

Here's how the image is specified using the inline image macro:

```
title_page:
  background_image:   'image:title-cover.jpg[]'
```

Like in the AsciiDoc syntax, the inline image macro allows you to supply set the width of the image and the alignment:

```
title_page:
  logo_image:         'image:logo.jpg[width=250,align=center]'
```

Quoted String

Some of the keys accept a quoted string as text content. The final segment of these keys is always named `content`.

A content key accepts a string value. It's usually best to quote the string or use the YAML multi-line string syntax.

Text content may be formatted using a subset of inline HTML. You can use the well-known elements such as `<strong>`, `<em>`, `<code>`, `<a>`, `<sub>`, `<sup>`, `<del>`, and `<span>`. The `<span>` element supports the `style` attribute, which you can use to specify the `color`, `font-weight`, and `font-style` CSS properties. You can also use the `rgb` attribute on the `<color>` element to change the color or the `name` and `size` attributes on the `<font>` element to change the font properties. If you need to add an underline or strikethrough decoration to the text, you can assign the `underline` or `line-through` to the `class` attribute on any aforementioned element.

Here's an example of using formatting in the content of the menu caret:

```
menu_caret_content:   '<font size=\"1.15em\"><color
rgb=\"#b12146\">\u203a</color></font>'
```

The string must be double quoted in order to use a Unicode escape code like `\u203a`.

Additionally, normal substitutions are applied to the value of content keys for running content, so you can use most AsciiDoc inline formatting (e.g., `*strong*` or `{attribute-name}`) in the values of those keys.

Fonts

You can select from built-in PDF fonts, bundled-fonts, fonts bundled with Asciidoctor PDF or custom fonts loaded from TrueType font (TTF) files. If you want to use custom fonts, you must first declare them in your theme file.

Asciidoctor has no challenge working with Unicode. In fact, it prefers Unicode and considers the entire range. However, once you convert to PDF, you have to meet the font requirements of PDF in order to preserve Unicode characters. There's nothing *Asciidoctor* can do to convince PDF to work with extended characters without the right fonts in play.

Built-In (AFM) Fonts

The names of the built-in fonts (for general-purpose text) are as follows:

Table 3. Built-In Fonts

Font Name	Font Family
Helvetica	sans-serif
Times-Roman	serif
Courier	monospace

Using a built-in font requires no additional files. You can use the key anywhere a `font_family` property is accepted in the theme file. For example:

```
base:
  font_family:        Times-Roman
```

However, when you use a built-in font, the characters you can use in your document are limited to the characters in the WINANSI code set (Windows-1252). WINANSI includes most of the characters needed for writing in Western languages (English, French, Spanish, etc). For anything outside of that, PDF is BYOF (Bring Your Own Font).

Even though the built-in fonts require the content to be encoded in WINANSI, *you still type your AsciiDoc document in UTF-8*. **Asciidoctor PDF** encodes the content into WINANSI when building the PDF.

WINANSI Encoding Behavior

When using the built-in PDF (AFM) fonts on a block of content in your AsciiDoc document, any character that cannot be encoded to WINANSI is replaced with a logic "not" glyph (¬) and you'll see the following warning in your console:

```
The following text could not be fully converted to the Windows-1252 character set:
| <string with unknown glyph>
```

This behavior differs from the default behavior in Prawn, which simply crashes. For more information about how Prawn handles character encodings for built-in fonts, see this note in the Prawn CHANGELOG.

Bundled Fonts

Asciidoctor PDF bundles several fonts that are used by the default theme. You can also use these fonts in your custom theme by simply declaring them. These fonts provide more characters than the built-in PDF

fonts, but still only a subset of UTF-8 (to reduce the size of the gem).

The family name of the fonts bundled with **Asciidoctor PDF** are as follows:

Noto Serif
: A serif font that can be styled as normal, italic, bold or bold_italic.

M+ 1mn
: A monospaced font that maps different thicknesses to the styles normal, italic, bold and bold_italic. Also provides the circuled numbers used in callouts.

M+ 1p Fallback
: A sans-serif font that provides a very complete set of Unicode glyphs. Cannot be styled as italic, bold or bold_italic. Used as the fallback font.

At the time of this writing, you cannot use the bundled fonts if you change the value of the `pdf-fontsdir` attribute (and thus define your own custom fonts). This limitation may be lifted in the future.

Custom Fonts

The limited character set of WINANSI, or the bland look of the built-in fonts, may motivate you to load your own font. Custom fonts can enhance the look of your PDF theme substantially.

To start, you need to find a collection of TTF file of the font you want to use. A collection typically consists of all four styles of a font:

- normal
- italic
- bold
- bold_italic

You'll need all four styles to support AsciiDoc content properly. *Asciidoctor PDF cannot italicize a font dynamically like a browser can, so you need the italic style.*

Once you've obtained the TTF files, put them into a directory in your project where you want to store the fonts. It's recommended that you name them consistently so it's easier to type the names in the theme file.

Let's assume the name of the font is Roboto. Name the files as follows:

- roboto-normal.ttf (*originally Roboto-Regular.ttf*)
- roboto-italic.ttf (*originally Roboto-Italic.ttf*)
- roboto-bold.ttf (*originally Roboto-Bold.ttf*)
- roboto-bold_italic.ttf (*originally Roboto-BoldItalic.ttf*)

Next, declare the font under the `font_catalog` key at the top of your theme file, giving it a unique key (e.g., `Roboto`).

```
font:
  catalog:
    Roboto:
      normal:         roboto-normal.ttf
      italic:         roboto-italic.ttf
      bold:           roboto-bold.ttf
```

```
      bold_italic:    roboto-bold_italic.ttf
```

You can use the key that you assign to the font in the font catalog anywhere the `font_family` property is accepted in the theme file. For instance, to use the Roboto font for all headings, you'd use:

```
heading:
  font_family:        Roboto
```

When you execute **Asciidoctor PDF**, you need to specify the directory where the fonts reside using the `pdf-fontsdir` attribute:

```
asciidoctor-pdf -a pdf-style=basic-theme.yml -a pdf-fontsdir=path/to/fonts
document.adoc
```

Currently, all fonts referenced by the theme need to be present in the directory specified by the `pdf-fontsdir` attribute.

When **Asciidoctor PDF** creates the PDF, it only embeds the glyphs from the font that are needed to render the characters present in the document. In other words, **Asciidoctor PDF** automatically subsets the font. However, if you're storing the fonts in a repository, you may want to subset the font (for instance, by using FontForge) to reduce the space the font occupies in that storage. This is simply a personal preference.

You can add any number of fonts to the catalog. Each font must be assigned a unique key, as shown here:

```
font:
  catalog:
    Roboto:
      normal:         roboto-normal.ttf
      italic:         roboto-italic.ttf
      bold:           roboto-bold.ttf
      bold_italic:    roboto-bold_italic.ttf
    Roboto Light:
      normal:         roboto-light-normal.ttf
      italic:         roboto-light-italic.ttf
      bold:           roboto-light-bold.ttf
      bold_italic:    roboto-light-bold_italic.ttf
```

Text in SVGs will use the font catalog from your theme. We recommend that you match the font key to the name of the font seen by the operating system. This will allow you to use the same font names (aka families) in both your graphics program and **Asciidoctor PDF**.

Fallback Fonts

If a TrueType font is missing a character needed to render the document, such as a special symbol, you can have **Asciidoctor PDF** look for the character in a fallback font. You only need to specify a single fallback font, typically one that provides a full set of symbols.

The fallback font is only used when the primary font is a TrueType font (i.e., TTF, DFont, TTC). Any glyph missing from an AFM font is simply replaced with the "not" glyph (¬).

Like with other custom fonts, you first need to declare the fallback font. Let's choose Droid Sans Fallback. You can map all the styles to a single font file (since bold and italic don't usually make sense for symbols).

Using the fallback font slows down PDF generation slightly because it has to analyze every single character. It's use is not recommended for large documents. Instead, it's best to select primary fonts that have all the characters you need. Keep in mind that the default theme currently uses a fallback font, though this may change in the future.

```
font:
  catalog:
    Roboto:
      normal:          roboto-normal.ttf
      italic:          roboto-italic.ttf
      bold:            roboto-bold.ttf
      bold_italic:     roboto-bold_italic.ttf
    DroidSansFallback:
      normal:          droid-sans-fallback.ttf
      italic:          droid-sans-fallback.ttf
      bold:            droid-sans-fallback.ttf
      bold_italic:     droid-sans-fallback.ttf
```

Next, add the key name to the `fallbacks` key under the `font_catalog` key. The `fallbacks` key accepts an array of values, meaning you can specify more than one fallback font. However, we recommend using a single fallback font, if possible, as shown here:

```
font:
  catalog:
    Roboto:
      normal:          roboto-normal.ttf
      italic:          roboto-italic.ttf
      bold:            roboto-bold.ttf
      bold_italic:     roboto-bold_italic.ttf
    DroidSansFallback:
      normal:          droid-sans-fallback.ttf
      italic:          droid-sans-fallback.ttf
      bold:            droid-sans-fallback.ttf
      bold_italic:     droid-sans-fallback.ttf
  fallbacks:
    - DroidSansFallback
```

If you are using more than one fallback font, add additional lines to the `fallbacks` key.

Of course, make sure you've configured your theme to use your custom font:

```
base:
  font_family:        Roboto
```

That's it! Now you're covered. If your custom font is missing a glyph, **Asciidoctor PDF** will look in your fallback font. You don't need to reference the fallback font anywhere else in your theme file.