AsciidoctorJ

Java bindings for Asciidoctor

Alex Soto, Dan Allen

Table of Contents

AsciidoctorJ is the official library for running Asciidoctor on the JVM. Using AsciidoctorJ, you can convert AsciiDoc content or analyze the structure of a parsed AsciiDoc document from Java and other JVM languages.

Distribution

The version of AsciidoctorJ matches the version of Asciidoctor RubyGem it bundles. AsciidoctorJ is published to Maven Central and Bintray. The artifact information can be found in the tables below.

Table 1. Artifact information for AsciidoctorJ in jCenter (Bintray)

Group Id	Artifact Id	Version	Download
org.asciidoctor	asciidoctorj	1.5.6	pom jar javadoc (jar) sources (jar)
org.asciidoctor	asciidoctorj-epub3	1.5.0-alpha.6	
org.asciidoctor	asciidoctorj-pdf	1.5.0-alpha.11	

Table 2. Artifact information for AsciidoctorJ in Maven Central

Group Id	Artifact Id	Version	Download
org.asciidoctor	asciidoctorj	1.5.6	pom jar javadoc (jar) sources (jar)
org.asciidoctor	asciidoctorj-epub3	1.5.0-alpha.6	
org.asciidoctor	asciidoctorj-pdf	1.5.0-alpha.11	

! The artifactId changed from asciidoctor-java-integration to asciidoctorj starting with version 1.5.0.

Installation

AsciidoctorJ is a standard `.jar` file. To start using it, you need to add the library to your project's classpath. To start using it under WildFly AS, you can't just use it, you also have to *modify your WildFly installation* due to classpath loading issues; see Running AsciidoctorJ on WildFly AS.

Declaring the dependency in a Maven build file (i.e., pom.xml)

```
<dependencies>
Ê <dependency>
Ê   <groupId>org.asciidoctor</groupId>
Ê   <artifactId>asciidoctorj</artifactId>
Ê   <version>1.5.6</version> !
Ê </dependency>
</dependencies>
```

Declaring the dependency in a Gradle build file (e.g., build.gradle)

```
dependencies {
Ê compile 'org.asciidoctor:asciidoctorj:1.5.6'
}
```

Declaring the dependency in an SBT build file (e.g., build.sbt)

```
libraryDependencies += "org.asciidoctor" % "asciidoctorj" % "1.5.6" !
```

! Specifying the version of AsciidoctorJ implicitly selects the version of Asciidoctor

Declaring the dependency in a Leiningen build file (e.g., project.clj)

```
:dependencies [[org.asciidoctor/asciidoctorj "1.5.6"]]
```

"

In addition to using AsciidoctorJ directly, you can invoke it as part of your build using the Maven or Gradle plugin.

- How to Install and Use the Asciidoctor Maven Plugin
- How to Install and Use the Asciidoctor Gradle Plugin

Windows Installation

A Chocolatey package is available which installs the ascidoctorj-1.5.6-bin.zip Bintray artifact along with a binary shim in %ChocolateyInstall%\bin which lets you run AsciidoctorJ from the command line.

```
C:\> choco install asciidoctorj
C:\> where asciidoctorj
C:\ProgramData\chocolatey\bin\asciidoctorj.exe
C:\> asciidoctorj -b pdf README.adoc
```

Converting documents

The main entry point for AsciidoctorJ is the `Asciidoctor` Java interface. This interface provides four methods for converting AsciiDoc content.

- `convert`
- `convertFile`
- `convertFiles`
- `convertDirectory`

You'll learn about these methods in the converting documents section.

Prior to Asciidoctor 1.5.0, the term `render` was used in these method names instead of `convert` (i.e., `render`, `renderFile`, `renderFiles` and `renderDirectory`). AsciidoctorJ continues to support the old method names for backwards compatibility.

Table 3. Convert methods on the `Asciidoctor` interface

Method Name	Return Type	Description
`convert`	`String`	Parses AsciiDoc content read from a string or stream and converts it to the format specified by the `backend` option.
`convertFile`	`String`	Parses AsciiDoc content read from a file and converts it to the format specified by the `backend` option.
`convertFiles`	`String[]`	Parses a collection of AsciiDoc files and converts them to the format specified by the `backend` option.
`convertDirectory`	`String[]`	Parses all AsciiDoc files found in the specified directory (using the provided strategy) and converts them to the format specified by the `backend` option.

All the methods listed in Table 3 are overloaded to accommodate various input types and options.

You retrieve an instance of the `Asciidoctor` interface from the factory method provided.

Creating an Asciidoctor instance from Asciidoctor.Factory

```
import static org.asciidoctor.Asciidoctor.Factory.create;
import org.asciidoctor.Asciidoctor;

Asciidoctor asciidoctor = create();
```

Once you retrieve an instance of the `Asciidoctor` interface, you can use it to convert AsciiDoc content. Here's an example of using AsciidoctorJ to convert an AsciiDoc string.

The following `convertFile` or `convertFiles` methods will only return a converted `String` object or array if you disable writing to a file, which is enabled by default. To disable writing to a file, create a new `Options` object, disable the option to create a new file with `option.setToFile(false)`, and then pass the object as a parameter to `convertFile` or `convertFiles`.

Converting an AsciiDoc string

```
//...
import java.util.HashMap;
//...

String html = asciidoctor.convert(
Ê   "Writing AsciiDoc is _easy_!",
Ê   new HashMap<String, Object>());
System.out.println(html);
```

The `convertFile` method will convert the contents of an AsciiDoc file.

Converting an AsciiDoc file

```
//...
import java.util.HashMap;
//...

String html = asciidoctor.convertFile(
Ê   new File("sample.adoc"),
Ê   new HashMap<String, Object>());
System.out.println(html);
```

The `convertFiles` method will convert a collection of AsciiDoc files:

Converting a collection of AsciiDoc files

```
//...
import java.util.Arrays;
//...

String[] result = asciidoctor.convertFiles(
Ê   Arrays.asList(new File("sample.adoc")),
Ê   new HashMap<String, Object>());

for (String html : result) {
Ê   System.out.println(html);
}
```

% If the converted content is written to files, the `convertFiles` method will return a String Array (i.e., `String[]`) with the names of all the converted documents.

Another method provided by the `Asciidoctor` interface is `convertDirectory`. This method converts all of the files with AsciiDoc extensions (`.adoc` *(preferred)*, `.ad`, `.asciidoc`, `.asc`) that are present within a specified folder and following given strategy.

An instance of the `DirectoryWalker` interface, which provides a strategy for locating files to process, must be passed as the first parameter of the `convertDirectory` method. Currently `Asciidoctor`

provides two built-in implementations of the `DirectoryWalker` interface:

Table 4. Built-in `DirectoryWalker` implementations

Class	Description
`AsciiDocDirectoryWalker`	Converts all files of given folder and all its subfolders. Ignores files starting with underscore (_).
`GlobDirectoryWalker`	Converts all files of given folder following a glob expression.

If the converted content is not written into files, `convertDirectory` will return an array listing all the documents converted.

Converting all AsciiDoc files in a directory

```
//...
import org.asciidoctor.AsciiDocDirectoryWalker;
//...

String[] result = asciidoctor.convertDirectory(
Ê   new AsciiDocDirectoryWalker("src/asciidoc"),
Ê   new HashMap<String, Object>());

for (String html : result) {
Ê   System.out.println(html);
}
```

Another way to convert AsciiDoc content is by calling the `convert` method and providing a standard Java `java.io.Reader` and `java.io.Writer`. The `Reader` interface is used as the source, and the converted content is written to the `Writer` interface.

Converting content read from a java.io.Reader to a java.io.Writer

```
//...
import java.io.FileReader;
import java.io.StringWriter;
//...

FileReader reader = new FileReader(new File("sample.adoc"));
StringWriter writer = new StringWriter();

asciidoctor.convert(reader, writer, options().asMap());

StringBuffer htmlBuffer = writer.getBuffer();
System.out.println(htmlBuffer.toString());
```

Safe mode and file system access

Asciidoctor provides security levels that control the read and write access of attributes, the include directive, macros, and scripts while a document is processing. Each level includes the restrictions

enabled in the prior security level.

When Asciidoctor (and AsciidoctorJ) is used as *API*, it uses `SECURE` safe mode by default. This mode is the most restrictive one and in summary it disallows the document from attempting to read files from the file system and including their contents into the document.

We recommend you to set `SAFE` safe mode when rendering AsciiDoc documents using AsciidoctorJ to have almost all Asciidoctor features such as *icons*, *include directive* or retrieving content from *URIs* enabled.

Safe mode is set as option when a document is rendered. For example:

```
import static org.asciidoctor.OptionsBuilder.options;

Map<String, Object> options = options().safe(SafeMode.SAFE)
Ê                                      .asMap();

String outfile = asciidoctor.convertFile(new File("sample.adoc"), options);
```

We are going to explain in more detail options in next section.

You can read more about safe modes in http://asciidoctor.org/docs/user-manual/#running-asciidoctor-securely

Conversion options

Asciidoctor supports numerous options, such as:

`in_place`
: Converts the output to a file adjacent to the input file.

`template_dirs`
: Specifies a directory of Tilt-compatible templates to be used instead of the default built-in templates

`attributes`
: A Hash (key-value pairs) of attributes to configure various aspects of the AsciiDoc processor

The second parameter of the `convert` method is `java.util.Map`. The options listed above can be set in `java.util.Map`.

Using the in_place option and the backend attribute

```
Map<String, Object> attributes = new HashMap<String, Object>();
attributes.put("backend", "docbook"); !

Map<String, Object> options = new HashMap<String, Object>();
options.put("attributes", attributes); "
options.put("in_place", true); #

String outfile = asciidoctor.convertFile(new File("sample.adoc"), options);
```

! Defines the backend attribute as docbook in the attributes map

" Registers the attributes map as the attributes option in the options map

Defines the in_place option in the options map

Another way for setting options is by using org.asciidoctor.Options class. Options is a simple Java class which contains methods for setting required options. Note that related with org.asciidoctor.Options class, there is org.asciidoctor.Attributes class, which can be used for setting attributes.

The convert method is overloaded so org.asciidoctor.Options can be passed instead of a java.util.Map.

Using the in_place option and the backend attribute

```
Attributes attributes = new Attributes();
attributes.setBackend("docbook"); !

Options options = new Options();
options.setAttributes(attributes); "
options.setInPlace(true); #

String outfile = asciidoctor.convertFile(new File("sample.adoc"), options);
```

! Defines the backend attribute as docbook in the attributes class

" Registers the attributes class as the attributes option in the options class

Defines the in_place option in the options class

AsciidoctorJ also provides two builder classes to create these maps and classes in a more readable form.

`AttributesBuilder`
: Used to define attributes with a fluent API

`OptionsBuilder`
: Used to define options with a fluent API

The code below results in the same output as the previous example but uses the builder classes.

Setting attributes and options with the builder classes

```
import static org.asciidoctor.AttributesBuilder.attributes;
import static org.asciidoctor.OptionsBuilder.options;

//...
Map<String, Object> attributes = attributes().backend("docbook") !
Ê                                          .asMap();

Map<String, Object> options = options().inPlace(true)
Ê                                  .attributes(attributes) "
Ê                                  .asMap(); #

String outfile = asciidoctor.convertFile(new File("sample.adoc"), options);
```

! Defines the backend attribute as docbook using fluent API.

" Registers the attributes map as attributes.

Converts options to java.util.Map instance.

Setting attributes and options with the builder classes

```
import static org.asciidoctor.AttributesBuilder.attributes;
import static org.asciidoctor.OptionsBuilder.options;

//...
Attributes attributes = attributes().backend("docbook").get(); !
Options options = options().inPlace(true).attributes(attributes).get(); "

String outfile = asciidoctor.convertFile(new File("sample.adoc"), options); #
```

! Defines and returns an Attributes class instead of java.util.Map by calling get() method instead of asMap().

" Defines and returns an Options class instead of java.util.Map by calling get() method instead of asMap().

Converts the document passing Options class.

All methods used to convert content are overloaded with OptionsBuilder parameter, so it is no longer required to call get nor asMap methods.

The icons attribute requires a String to set the value used to Òdraw Ó icons. At this time, you can use two constants org.asciidoctor.Attributes.IMAGE_ICONS for using the same approach as AsciiDoc, that is using img tags, or org.asciidoctor.Attributes.FONT_ICONS for using icons from Font Awesome.

Attributes can be specified as String or Array instead of pair key/value by using org.asciidoctor.Attributes.setAttributes(String) or org.asciidoctor.Attributes.setAttributes(String...) and AttributesBuilder methods.

Passing attributes as a string

```
//...
Attributes attributes = attributes().attributes("toc numbered").get();
Options options = options().attributes(attributes).get();
```

Passing attributes as a string is equivalent to passing individual attributes.

Passing individual attributes

```
//...
Attributes attributes = attributes().tableOfContents(true).sectionNumbers(true).get();
Options options = options().attributes(attributes).get();
```

You can also use an array.

Passing attributes as an array

```
//...
String[] attributesArray = new String[]{"toc", "source-highlighter=coderay"};
Attributes attributes = attributes().attributes(attributesArray).sectionNumbers(true)
.get();
Options options = options().attributes(attributes).get();
```

Passing attributes as an array is equivalent to passing individual attribute.

Passing individual attributes

```
//...
Attributes attributes = attributes().tableOfContents(true).sectionNumbers(true)
.sourceHighlighter("coderay").get();
Options options = options().attributes(attributes).get();
```

Locating files

A utility class `AsciiDocDirectoryWalker` is available for searching the AsciiDoc files present in a root folder and its subfolders. `AsciiDocDirectoryWalker` locates all files that end with `.adoc`, `.ad`, `.asciidoc` or `.asc`. Also it ignores all files starting with underscore (_).

Locating AsciiDoc files with `AsciiDocDirectoryWalker`

```
import java.util.List;
import org.asciidoctor.AsciiDocDirectoryWalker;

DirectoryWalker directoryWalker = new AsciiDocDirectoryWalker("docs"); !
List<File> asciidocFiles = directoryWalker.scan(); "
```

! Defines which parent directory is used for searching.

" Returns a list of all AsciiDoc files found in root folder and its subfolders.

A utility class `GlobDirectoryWalker` is available for searching the AsciiDoc files present in a root folder and scanning using a `Glob` expression. `GlobDirectoryWalker` locates all files that end with `.adoc`, `.ad`, `.asciidoc` or `.asc`.

Locating AsciiDoc files with `GlobDirectoryWalker`

```
import java.util.List;
import org.asciidoctor.GlobDirectoryWalker;

DirectoryWalker directoryWalker = new GlobDirectoryWalker("docs", "**/*.adoc");  !
List<File> asciidocFiles = directoryWalker.scan();  "
```

! Defines which parent directory is used for searching and the glob expression.

" Returns a list of all AsciiDoc files matching given glob expression.

Reading the document tree

Instead of converting an AsciiDoc document, you may want to parse the document to read information it contains or navigate the document's structure. AsciidoctorJ let's you do this!

There are two approaches you can take to read the structure of an AsciiDoc document with AsciidoctorJ.

Using wrapper classes not connected with Ruby internal model

The structure is copied in Java non-proxied classes so any change does not modify the original document.

Using JRuby Java wrapper classes

The Java classes are linked to Ruby internal classes. Any modifications done here are reflected to the original document.

Wrapping classes

Example AsciiDoc document with header information

```
= Sample Document
Doc Writer <doc.writer@asciidoc.org>; John Smith <john.smith@asciidoc.org>
v1.0, 2013-05-20: First draft
:title: Sample Document
:tags: [document, example]

Preamble...
```

The `readDocumentHeader` method on the `Asciidoctor` interface retrieves information from the header of an AsciiDoc document without parsing or converting the entire document. This method returns an instance of `org.asciidoctor.ast.DocumentHeader` with all information from the header filled.

Reading header information from the AsciiDoc document

```
//...
import org.asciidoctor.ast.DocumentHeader;

//...
DocumentHeader header = asciidoctor.readDocumentHeader(
Ê   new File("header-sample.adoc"));

System.out.println(header.getDocumentTitle().getMain()); !

Author author = header.getAuthor(); "
System.out.println(author.getEmail()); #
System.out.println(author.getFullName()); $

RevisionInfo revisionInfo = header.getRevisionInfo();

System.out.println(revisionInfo.getDate()); %
System.out.println(revisionInfo.getNumber()); &
System.out.println(revisionInfo.getRemark()); '
```

! prints `Sample Document`

" prints `Doc Writer`

prints `doc.writer@asciidoc.org`

$ prints `Doc Writer`

% prints `2013-05-20`

& prints `1.0`

' prints `First draft`

The `readDocumentHeader` method can be extremely useful for building an index of documents.

```
import java.io.File;
import java.util.HashSet;
import java.util.Set;
import org.asciidoctor.Asciidoctor;
import org.asciidoctor.AsciiDocDirectoryWalker;
import org.asciidoctor.DirectoryWalker;
import org.asciidoctor.DocumentHeader;

//...

Asciidoctor asciidoctor = Asciidoctor.Factory.create();
Set<DocumentHeader> documentIndex = new HashSet<DocumentHeader>();
DirectoryWalker directoryWalker = new AsciiDocDirectoryWalker("docs");  !

for (File file : directoryWalker.scan()) {
Ê documentIndex.add(asciidoctor.readDocumentHeader(file));
}
```

! Converts all files in the `docs` folder and its subfolders.

You can also load the document inside a `Document` object. This object represents the whole document, including its headings. You can use it to navigate through the internals of a parsed document. To load a document, use the `load` or `loadFile` methods.

The `readDocumentStructure` method provides a useful way of parsing an AsciiDoc file into the structured object. First, it gathers the same information as `readDocumentHeader` and puts it in the `header` field of the `StructuredDocument` object. The actual content of the file is split into separate ContentParts based on blocks of the content.

This feature provides several use cases.

AsciiDoc document with two blocks defined by section titles

```
= Sample Document

== Section one
This is content of section one

== Section two
And content of section two

...
```

Each section defines new content part. List of all parts can be get by `getParts` method on `StructuredDocument`. Each part will than contain of title (i.e. "Section one") and converted text content as html.

Print content of each part

```
for (ContentPart part : document.getParts()){
Ê System.out.println(part.getTitle());
Ê System.out.println("----");
Ê System.out.println(part.getContent);
Ê System.out.println("----");
}
```

AsciiDoc document with two blocks defined by styles

```
= Sample Document

[style one]
This is content of first content part

[[partId]]
[style two,role=partRole]
--
And content of second content part

This block can be as long as you want.
--
```

This way you can then use methods like `getPartByStyle` to retrieve particular content parts.

Retrieve content part by style

```
ContentPart style_two = document.getPartByStyle("style two");
// other possible way of retrieving parts:
ContentPart style_two = document.getPartById("partId")
ContentPart style_two = document.getPartByRole("partRole")

//and also for lists
List<ContentPart> parts = document.getPartsByStyle("style two");
List<ContentPart> parts = document.getPartsByRole("partRole");
List<ContentPart> parts = document.getPartsByContext("open");
```

Really nice thing about it is possibility to parse images to `Image` object that you can use later to embed in html page directly from your java code or manipulate in any other way.

Define images

```
[Images]
image::src/some{sp}image{sp}1.JPG[TODO title1,link="link1.html"]
image::src/some{sp}image{sp}2.JPG[TODO title2,link="link2.html"]
```

to get a list of images defined in the document and then to process images:

Retrieve image information

```
List<ContentPart> images = document.getPartsByContext("image");
for (ContentPart image : images){
Ê String src = (String) image.getAttributes().get("target");
Ê String alt = (String) image.getAttributes().get("alt");
Ê String link = (String) image.getAttributes().get("link");
}
```

As of final example consider following complete use case:

AsciiDoc document with product definition

```
= Sample product
v1.0, 2013-10-12
:hardbreaks:

:price: 70 pln
:smallImage: photos/small/small_image.jpg

[Description]
short product description

[Images]
image::photos/image1.jpg[title]
image::photos/image2.jpg[title]


[Detail]
--
Detail information about product. Note that you can use all asciidoc features here
like:
.simple list
* lists
* images
* titles
* further blocks

[role=text-center]
also you can also add css style by assigning role to the text.
--
```

and the way it can be than transformed to java object:

Java method for getting product

```
Product product = new Product();
product.setTitle(document.getHeader().getDocumentTitle());
product.setPrice(new Price((String) document.getHeader().getAttributes().get("price")
));
product.setSmallImage(new Image((String)document.getHeader().getAttributes().get(
"smallImage"),product.getTitle()));

product.setDescription(document.getPartByStyle("description").getContent());

List<ContentPart> images = document.getPartsByContext("image");
for (ContentPart image : images) {
Ê Image image = new Image();
Ê image.setSrc((String) image.getAttributes().get("target"));
Ê image.setAlt((String) image.getAttributes().get("alt"));
Ê product.getImages().add(image);
}

product.setDetail(document.getPartByStyle("detail").getContent());
```

Last feature of structure document is possibility to configure how deeply should blocks be processed. Default is one level only so if you want to have more nested structure add `STRUCTURE_MAX_LEVEL` parameter to processing options.

Configuration of the structure document processing

```
Map<String,Object> parameters = new HashMap<String, Object>();
parameters.put(Asciidoctor.STRUCTURE_MAX_LEVEL, 2);
StructuredDocument document = asciidoctor.readDocumentStructure(
Ê   new File("target/test-classes/documentblocks.asciidoc"),
Ê   parameters);
```

JRuby wrapping classes

```
import org.asciidoctor.ast.Document;

//...

Document document = asciidoctor.load(DOCUMENT, new HashMap<String, Object>()); !
assertThat(document.doctitle(), is("Document Title")); "
```

! Document from an String is loaded into `Document` object.

" Title of the document is retrieved.

But also all blocks that conforms the document can be retrieved. Currently there are support for three kinds of blocks. `Blocks` itself, `Section` for sections of the document and `AbstractBlock` which is the base type where all kind of blocks (including those not mapped as Java class) are mapped.

```
import org.asciidoctor.ast.Document;
import org.asciidoctor.ast.Section;

//...

Document document = asciidoctor.load(DOCUMENT, new HashMap<String, Object>()); !
Section section = (Section) document.blocks().get(1); "

assertThat(section.index(), is(0)); #
assertThat(section.sectname(), is("sect1"));
assertThat(section.special(), is(false));
```

! Document from an String is loaded into `Document` object.

" All blocks are get and because in this example the first block is a Section block, we cast it directly.

Concrete methods for sections can be called.

Blocks can also be retrieved from query using `findBy` method.

```
Document document = asciidoctor.load(DOCUMENT, new HashMap<String, Object>());
Map<Object, Object> selector = new HashMap<Object, Object>(); !
selector.put("context", ":image"); "

List<AbstractBlock> findBy = document.findBy(selector);
assertThat(findBy, hasSize(2)); #
```

! To make queries you need to use a `map` approach. Currently this is because of the Asciidoctor API but it will change in near future.

" In this example all blocks with context as image is returned. Notice that the colon (`:`) must be added in the value part.

Document used as example contains two images.

Extension API

If you plan to write extensions for Asciidoctor you should seriously consider using the latest version of the 1.6.0 family. While the current version is an alpha version it contains the same solid version of Asciidoctor. The Extension API has changed a bit, in an incompatible way though, but is much more solid when it comes to extensions. To not break compatibility with existing users of the Extension API it is not possible to bring these stabilization efforts to the current 1.5 family.

One of the major improvements to Asciidoctor recently is the extensions API. AsciidoctorJ brings this extension API to the JVM environment. AsciidoctorJ allows us to write extensions in Java instead of Ruby.

Asciidoctor provides seven types of extension points. Each extension point has an abstract class in Java that maps to the extension API in Ruby.

Table 5. AsciidoctorJ extension APIs

Name	Class
`Preprocessor`	org.asciidoctor.extension.Preprocessor
`Treeprocessor`	org.asciidoctor.extension.Treeprocessor
`Postprocessor`	org.asciidoctor.extension.Postprocessor
`Block processor`	org.asciidoctor.extension.BlockProcessor
`Block macro processor`	org.asciidoctor.extension.BlockMacroProcessor
`Inline macro processor`	org.asciidoctor.extension.InlineMacroProcessor
`Include processor`	org.asciidoctor.extension.IncludeProcessor
`Docinfo processor`	org.asciidoctor.extension.DocinfoProcessor

To create an extension two things are required:

1. Create a class implementing an extension class (this will depend on the kind of interface being developed)
2. Register your class using the `JavaExtensionRegistry` class

An extension can be registered by:

- Passing as String a fully qualified class.
- Passing a Class object of the extension.
- Passing an instance of the extension object.

In the first two cases, the lifecycle of the instance is managed by JRuby. In the last case, the caller is the owner of the lifecycle of the object.

Using an already created instance as an extension is useful when using CDI. For example, you can inject any extension inside the code and then register that instance as an extension.

Preprocessor

This extension updates an attribute value defined in a document.

Preprocessor example

```
public class ChangeAttributeValuePreprocessor extends Preprocessor { !

Ê   public ChangeAttributeValuePreprocessor(Map<String, Object> config) { "
Ê       super(config);
Ê   }

Ê   @Override
Ê   public PreprocessorReader process(Document document, PreprocessorReader reader) {
 #
Ê       document.getAttributes().put("content", "Alex");
Ê       return reader;
Ê   }

}
```

! Class must extend from `Preprocessor`.

" A constructor receiving a `Map` must be provided in case you want to send options to the preprocessor.

The `process` method receives a `Document` and `PreprocessorReader`.

Register a Preprocessor

```
JavaExtensionRegistry extensionRegistry = this.asciidoctor.javaExtensionRegistry(); !

extensionRegistry.preprocessor(ChangeAttributeValuePreprocessor.class); "

String content = asciidoctor.convertFile(new File(
Ê               "target/test-classes/changeattribute.adoc"),
Ê               new Options()); #
```

! `JavaExtensionRegistry` class is created.

" `Preprocessor` extension is registered by using `Class` approach.

We can call any `convert` method as usual; no extra parameters are required.

Treeprocessor

This extension detects literal blocks that contain terminal commands. It strips the prompt character and styles the command.

Treeprocessor example

```
public class TerminalCommandTreeprocessor extends Treeprocessor { !

Ê private Document document;

Ê public TerminalCommandTreeprocessor(Map<String, Object> config) {
```

```
Ê    super(config);
Ê }

Ê @Override
Ê public Document process(Document document) {

Ê   this.document = document;

Ê   final List<AbstractBlock> blocks = this.document.getBlocks(); "

Ê   for (int i = 0; i < blocks.size(); i++) {
Ê       final AbstractBlock currentBlock = blocks.get(i);
Ê       if(currentBlock instanceof Block) {
Ê           Block block = (Block)currentBlock;
Ê           List<String> lines = block.lines(); #
Ê           if (lines.size() > 0 && lines.get(0).startsWith("$")) {
Ê               blocks.set(i, convertToTerminalListing(block));
Ê           }
Ê       }
Ê   }

Ê   return this.document;
Ê }

Ê public Block convertToTerminalListing(Block block) {

Ê   Map<String, Object> attributes = block.getAttributes();
Ê   attributes.put("role", "terminal");
Ê   StringBuilder resultLines = new StringBuilder();

Ê   List<String> lines = block.lines();

Ê   for (String line : lines) {
Ê       if (line.startsWith("$")) {
Ê           resultLines.append("<span class=\"command\">")
Ê                      .append(line.substring(2, line.length()))
Ê                      .append("</span>");
Ê       }
Ê       else {
Ê           resultLines.append(line);
Ê       }
Ê   }

Ê   return createBlock(this.document, "listing", Arrays.asList(resultLines.toString()
), attributes,
Ê             new HashMap<Object, Object>()); $
Ê }

}
```

! Class must extend from Treeprocessor.

" The `document` instance can be used to retrieve all blocks of the current document.

All of the select block's lines are retrieved.

$ To create a new block, you must use the `createBlock` method. Set the parent document, the context (*listing*), the text content, attributes and options.

Register a Treeprocessor

```
JavaExtensionRegistry extensionRegistry = this.asciidoctor.javaExtensionRegistry(); !

extensionRegistry.treeprocessor("org.asciidoctor.extension.TerminalCommandTreeprocesso
r"); "

String content = asciidoctor.convertFile(new File(
Ê              "target/test-classes/sample-with-terminal-command.adoc"),
Ê              new Options()); #
```

! `JavaExtensionRegistry` class is created.

" `Treeprocessor` extension is registered using fully qualified class name as `String`.

We can call any `convert` method as usually, no extra parameters are required.

Postprocessor

This extension inserts custom footer text.

Postprocessor example

```
public class CustomFooterPostProcessor extends Postprocessor { !

Ê public CustomFooterPostProcessor(Map<String, Object> config) {
Ê     super(config);
Ê }

Ê @Override
Ê public String process(Document document, String output) { "

Ê     String copyright  = "Copyright Acme, Inc.";

Ê     if(document.basebackend("html")) {
Ê         org.jsoup.nodes.Document doc = Jsoup.parse(output, "UTF-8");

Ê         Element contentElement = doc.getElementById("footer-text");
Ê         contentElement.append(copyright);

Ê         output = doc.html();

Ê     }
Ê     return output; #
Ê }

}
```

! Class must extend from `Postprocessor`.

" `process` method receives the `Document` instance and the document converted as `String`.

The content that will be written in document is returned.

Register a Postprocessor

```
JavaExtensionRegistry extensionRegistry = this.asciidoctor.javaExtensionRegistry(); !

extensionRegistry.postprocessor(new CustomFooterPostProcessor(
Ê                                     new HashMap<String, Object>())); "

String content = asciidoctor.convertFile(new File(
Ê            "sample.adoc"),
Ê            options); #
```

! `JavaExtensionRegistry` class is created.

" `Postprocessor` extension is registered using an already created instance approach.

We can call any `convert` method as usually, no extra parameters are required.

DocinfoProcessor

This extension inserts custom content on header or footer of the document. For example can be used for adding `meta` tags on `<head>` tags.

DocinfoProcessor example

```
public class MetaRobotsDocinfoProcessor extends DocinfoProcessor {  !

Ê   public MetaRobotsDocinfoProcessor() {
Ê       super();
Ê   }

Ê   public MetaRobotsDocinfoProcessor(Map<String, Object> config) {
Ê       super(config);
Ê   }

Ê   @Override
Ê   public String process(Document document) {
Ê       return "<meta name=\"robots\" content=\"index,follow\">";  "
Ê   }
}
```

! Class must extend from `DocinfoProcessor`.

" In this case a meta tag is returned. By default *docinfo* is placed on header.

To register the DocinfoProcessor extension just use the `JavaExtensionRegistry`.

Register a DocinfoProcessor

```
JavaExtensionRegistry javaExtensionRegistry = this.asciidoctor.javaExtensionRegistry(
);  !

javaExtensionRegistry.docinfoProcessor(MetaRobotsDocinfoProcessor.class);  "

String content = asciidoctor.renderFile(
Ê               classpath.getResource("simple.adoc"),
Ê               options().headerFooter(true).safe(SafeMode.SERVER).toFile(false).get(
));
```

! `JavaExtensionRegistry` class is created.

" `DocinfoProcessor` extension is registered using an already created instance approach.

If you want to place *docinfo* on footer you need to set `location` option to `footer`.

Register a DocinfoProcessor

```
JavaExtensionRegistry javaExtensionRegistry = this.asciidoctor.javaExtensionRegistry(
);

Map<String, Object> options = new HashMap<String, Object>();
options.put("location", ":footer"); !
MetaRobotsDocinfoProcessor metaRobotsDocinfoProcessor = new
MetaRobotsDocinfoProcessor(options);

javaExtensionRegistry.docinfoProcessor(metaRobotsDocinfoProcessor);

String content = asciidoctor.renderFile(
Ê               classpath.getResource("simple.adoc"),
Ê               options().headerFooter(true).safe(SafeMode.SERVER).toFile(false).get(
));
```

! Sets the location option to footer. Note that a colon (:) is placed before *footer*.

Block processor

This extension registers a custom block style named *yell* that uppercases all the words.

Block processor example

```
public class YellBlock extends BlockProcessor { !

Ê public YellBlock(String name, Map<String, Object> config) { "
Ê     super(name, config);
Ê }

Ê @Override
Ê public Object process(AbstractBlock parent, Reader reader, Map<String, Object>
attributes) { #
Ê     List<String> lines = reader.readLines();
Ê     String upperLines = null;
Ê     for (String line : lines) {
Ê         if (upperLines == null) {
Ê             upperLines = line.toUpperCase();
Ê         }
Ê         else {
Ê             upperLines = upperLines + "\n" + line.toUpperCase();
Ê         }
Ê     }

Ê         return createBlock(parent, "paragraph", Arrays.asList(upperLines),
attributes, new HashMap<Object, Object>()); $
Ê }

}
```

! Class must extend from `BlockProcessor`.

" Constructor must receive the name of the block and a `Map` for sending options to block.

`process` method receives the parent block, a reader, and attributes defined in block.

$ To create a new block we must use `createBlock` method. We must set the parent document, the context (listing), the text content, attributes and options.

Register a Block processor

```
JavaExtensionRegistry extensionRegistry = this.asciidoctor.javaExtensionRegistry(); !

extensionRegistry.block("yell", YellBlock.class); "

String content = asciidoctor.convertFile(new File(
Ê             "target/test-classes/sample-with-yell-block.adoc"),
Ê             new Options()); #
```

! `JavaExtensionRegistry` class is created.

" `BlockProcessor` extension is registered with the context of block.

We can call any `convert` method as usually, no extra parameters are required.

Example of Block processor

```
[yell] !
The time is now. Get a move on.
```

! Note that *yell* is the context where block lives and is the same as the first parameter of `block` method of `ExtensionRegistry` class.

Block macro processor

This extension creates a block macro named gist for embedding a gist.

Block macro processor example

```
public class GistMacro extends BlockMacroProcessor { !

Ê public GistMacro(String macroName, Map<String, Object> config) { "
Ê     super(macroName, config);
Ê }

Ê @Override
Ê public Block process(AbstractBlock parent, String target,
Ê                      Map<String, Object> attributes) { #

Ê   String content = "<div class=\"content\">\n" +
Ê           "<script src=\"https://gist.github.com/"+target+".js\"></script>\n" +
Ê           "</div>";

Ê   return createBlock(parent, "pass", Arrays.asList(content), attributes,
Ê                      this.getConfig()); $
Ê }

}
```

! Class must extend from `BlockMacroProcessor`.

" Constructor must receive the macro name, and a `Map` for sending options to block..

`process` method receives the parent document, the content of the macro, and attributes defined in macro.

$ To create a new block we must use `createBlock` method. We must set the parent document, the context (listing), the text content, attributes and options.

Register a Block macro processor

```
JavaExtensionRegistry extensionRegistry = this.asciidoctor.javaExtensionRegistry(); !

extensionRegistry.blockMacro("gist", GistMacro.class); "

String content = asciidoctor.convertFile(new File(
Ê              "target/test-classes/sample-with-gist-macro.adoc"),
Ê              new Options()); #
```

! `JavaExtensionRegistry` class is created.

" `BlockMacroProcessor` extension is registered with the name of the macro.

We can call any `convert` method as usually, no extra parameters are required.

Example of Block macro processor

```
.My Gist
gist::123456[] !
```

! Note that *gist* is the name of the macro and is the same as the first parameter of `blockMacro` method of `ExtensionRegistry` class.

Inline macro processor

This extension creates an inline macro named man that links to a manpage.

Inline macro processor example

```
public class ManpageMacro extends InlineMacroProcessor { !

Ê public ManpageMacro(String macroName, Map<String, Object> config) { "
Ê     super(macroName, config);
Ê }

Ê @Override
Ê protected String process(AbstractBlock parent, String target, Map<String, Object>
attributes) { #

Ê     Map<String, Object> options = new HashMap<String, Object>();
Ê     options.put("type", ":link");
Ê     options.put("target", target + ".html");
Ê     return createInline(parent, "anchor", target, attributes, options).convert(); $
Ê }

}
```

! Class must extend from `InlineMacroProcessor`.

" Constructor must receive the macro name, and a `Map` for passing options to extension.

`process` method receives the parent document, the content of the macro, and attributes defined in macro.

$ Because it is an inline macro, only a replacement string must be returned.

Register an Inline macro processor

```
JavaExtensionRegistry extensionRegistry = this.asciidoctor.javaExtensionRegistry(); !

extensionRegistry.inlineMacro("man", ManpageMacro.class); "

String content = asciidoctor.convertFile(new File(
Ê                "target/test-classes/sample-with-man-link.adoc"),
Ê                new Options()); #
```

! `ExtensionRegistry` class is created.

" `InlineMacroProcessor` extension is registered with the name of the macro.

We can call any `convert` method as usually, no extra parameters are required.

Example of Inline macro processor

```
See man:gittutorial[7] to get started. !
```

! Note that *man* is the name of the macro and is the same as the first parameter of `inlineMacro` method of `ExtensionRegistry` class.

Include processor

Include a file from a URI.

Include processor example

```
public class UriIncludeProcessor extends IncludeProcessor { !

Ê public UriIncludeProcessor(Map<String, Object> config) {
Ê   super(config);
Ê }

Ê @Override
Ê public boolean handles(String target) {
Ê   return target.startsWith("http://") || target.startsWith("https://"); "
Ê }

Ê @Override
Ê public void process(DocumentRuby document, PreprocessorReader reader, String target,
Ê           Map<String, Object> attributes) {

Ê   StringBuilder content = readContent(target);
Ê   reader.push_include(content.toString(), target, target, 1, attributes); #

Ê }

Ê private StringBuilder readContent(String target) {

Ê   StringBuilder content = new StringBuilder();

Ê   try {

Ê     URL url = new URL(target);
Ê     InputStream openStream = url.openStream();

Ê     BufferedReader bufferedReader = new BufferedReader(
Ê                   new InputStreamReader(openStream));

Ê     String line = null;
Ê     while ((line = bufferedReader.readLine()) != null) {
Ê       content.append(line);
Ê     }

Ê     bufferedReader.close();

Ê     } catch (MalformedURLException e) {
Ê         throw new IllegalArgumentException(e);
Ê     } catch (IOException e) {
Ê         throw new IllegalArgumentException(e);
Ê     }
Ê     return content;
Ê }

}
```

! Class must extend from `IncludeProcessor`.

" `handles` method is used by processor to decide if included element should be converted by this processor or not. `target` attribute is the value of `include` macro.

`push_include` method inserts new content (retrieved from the url) in current position of document.

Register an include processor

```
JavaExtensionRegistry extensionRegistry = this.asciidoctor.javaExtensionRegistry(); !

extensionRegistry.includeProcessor(UriIncludeProcessor.class); "

String content = asciidoctor.convertFile(new File(
Ê               "target/test-classes/sample-with-uri-include.adoc"),
Ê               new Options()); #
```

! `ExtensionRegistry` class is created.

" `IncludeProcessor` extension is registered.

We can call any `convert` method as usually; no extra parameters are required.

Example of include processor

```
= Example of URI

.Gemfile
[source,ruby]
----
include::https://raw.githubusercontent.com/asciidoctor/asciidoctor/master/Gemfile[]
----
```

Unregistering extensions

You can unregister all extensions by calling the `org.asciidoctor.Asciidoctor.unregisterAllExtensions()` method.

Additionally, since AsciidoctorJ 1.5.6.1 it is possible to unregister specific extensions when using the ExtensionGroup API. This allows to register and unregister a group of extensions as a whole:

```
BlockProcessor extension1 = ...
Treeprocessor extension2 = ...
ExtensionGroup extensionGroup =
Ê asciidoctor.createGroup()
Ê   .block(extension1)
Ê   .treeprocessor(extension2);

asciidoctor.convert(...) !

extensionGroup.register();
asciidoctor.convert(); "

extensionGroup.register();
asciidoctor.convert(); #
```

! Convert a document without the extensions active because they were not registered yet.

" Convert with extensions active after registration.

Convert without extensions active after the extensions were unregistered.

Ruby extensions

You can even register extensions written in Ruby using AsciidoctorJ. To register a *Ruby* extension you must get a `RubyExtensionRegistry` class instead of `JavaExtensionRegistry`.

Register a Ruby extension in Java

```
RubyExtensionRegistry rubyExtensionRegistry = this.asciidoctor.rubyExtensionRegistry(
); !
rubyExtensionRegistry.loadClass(Class.class.getResourceAsStream("/YellRubyBlock.rb")).
block("rubyyell", "YellRubyBlock"); "

String content = asciidoctor.convertFile(new File(
Ê              "target/test-classes/sample-with-ruby-yell-block.ad"),
Ê              options().toFile(false).get());
```

! `rubyExtensionRegistry` method is called to get a rubyExtensionRegistry instance.

" Ruby file containing a class implementing a Block extension is loaded inside the Ruby runtime. Then the block is registered with a name (rubyyell), and we pass the name of the class to be instantiated.

YellBlock.rb

```
require 'asciidoctor'
require 'asciidoctor/extensions'

class YellRubyBlock < Asciidoctor::Extensions::BlockProcessor
Ê option :contexts, [:paragraph]
Ê option :content_model, :simple

Ê def process parent, reader, attributes
Ê   lines = reader.lines.map {|line| line.upcase.gsub(/\.( |$)/, '!\\1') }
Ê   Asciidoctor::Block.new parent, :paragraph, :source => lines, :attributes =>
attributes
Ê end
end
```

Extension SPI

In previous examples, the extensions were registered manually. However, AsciidoctorJ provides another way to register extensions. If any implementation of the SPI interface is present on the classpath, it will be executed.

To create an autoloadable extension you should do next steps.

Create a class that implements `org.asciidoctor.extension.spi.ExtensionRegistry`.

org.asciidoctor.extension.ArrowsAndBoxesExtension.java

```
public class ArrowsAndBoxesExtension implements ExtensionRegistry { !

Ê @Override
Ê public void register(Asciidoctor asciidoctor) { "

Ê   JavaExtensionRegistry javaExtensionRegistry = asciidoctor.javaExtensionRegistry();
Ê   javaExtensionRegistry.postprocessor(ArrowsAndBoxesIncludesPostProcessor.class); #
Ê   javaExtensionRegistry.block("arrowsAndBoxes", ArrowsAndBoxesBlock.class);

Ê }

}
```

! To autoload extensions you need to implement `ExtensionRegistry`.

" AsciidoctorJ will automatically run the `register` method. The method is responsible for registering all extensions.

All required Java extensions are registered.

Next, you need to create a file called `org.asciidoctor.extension.spi.ExtensionRegistry` inside `META-INF/services` with the implementation's full qualified name.

META-INF/services/org.asciidoctor.extension.spi.ExtensionRegistry

```
org.asciidoctor.extension.ArrowsAndBoxesExtension
```

And that's all. Now when a `.jar` file containing the previous structure is dropped inside classpath of AsciidoctorJ, the `register` method will be executed automatically and the extensions will be registered.

Logs handling API

$ This API is inspired by Java Logging API (JUL). If you are familiar with `java.util.logging.*` you will see familiar analogies with some of its components.

AciidoctorJ (v1.5.7+) offers the possibility to capture messages generated during document rendering. These messages correspond to logging information and are organized in 6 severity levels:

1. DEBUG
2. INFO
3. WARN
4. ERROR
5. FATAL
6. UNKNOWN

The easiest way to capture messages is registering a `LogHandler` through the `Asciidoctor` instance.

Registering a LogHandler

```
Asciidoctor asciidoctor = Asciidoctor.Factory.create();

asciidoctor.registerLogHandler(new LogHandler() { !
Ê   @Override
Ê   public void log(LogRecord logRecord) {
Ê       System.out.println(logRecord.getMessage());
Ê   }
});
```

! Use `registerLogHandler` to register one or more handlers.

The `log` method in the `org.asciidoctor.log.LogHandler` interface provides a `org.asciidoctor.log.LogRecord` that exposes the following information:

Severity severity

Severity level of the current record.

Cursor cursor

Information about the location of the event, contains:

- LineNumber: relative to the file where the message occurred.
- Path: source file simple name, or `<stdin>` value when rending from a String.
- Dir: absolute path to the source file parent directory, or the execution path when rending from a String.
- File: absolute path to the source file, or `null` when rending from a String.
  These will point to the correct source file, even when this is included from another.

String message

Descriptive message about the event.

String sourceFileName

Contains the value `<script>`.
For the source filename see `Cursor` above.

String sourceMethodName

The Asciidoctor Ruby engine method used to render the file; `convertFile` or `convert` whether you are rending a File or a String.

Logs Handling SPI

Similarly to AsciidoctorJ extensions, the Log Handling API provides an alternate method to register Handlers without accessing `Asciidoctor` instance.

Start creating a normal LogHandler implementation.

```
package my.asciidoctor.log.MemoryLogHandler;

import java.util.ArrayList;
import java.util.List;
import org.asciidoctor.log.LogHandler;
import org.asciidoctor.log.LogRecord;

/**
Ê* Stores LogRecords in memory for later analysis.
Ê*/
public class MemoryLogHandler extends LogHandler {

Ê private List<LogRecord> logRecords = new ArrayList<>();

Ê @Override
Ê public void log(LogRecord logRecord) {
Ê   logRecords.add(record);
Ê }

Ê public List<LogRecord> getLogRecords() {
Ê   return logRecords;
Ê }
}
```

Next, create a file called `org.asciidoctor.log.LogHandler` inside `META-INF/services` with the implementation's full qualified name.

META-INF/services/org.asciidoctor.log.LogHandler

```
my.asciidoctor.log.MemoryLogHandler
```

And that's all. Now when a .jar file containing the previous structure is dropped inside classpath of AsciidoctorJ, the handler will be registered automatically.

Converting to EPUB3

The Asciidoctor EPUB3 gem (asciidoctor-epub3) is bundled inside the AsciidoctorJ EPUB3 jar (asciidoctorj-epub3). To use it, simply add the asciidoctorj-epub3 jar to your dependencies. The version of the AsciidoctorJ EPUB3 jar aligns with the version of the Asciidoctor EPUB3 gem.

Here's how you can add the AsciidoctorJ EPUB3 jar to your Maven dependencies:

```
<dependencies>
Ê <dependency>
Ê   <groupId>org.asciidoctor</groupId>
Ê   <artifactId>asciidoctorj-epub3</artifactId>
Ê   <version>1.5.0-alpha.4</version>
Ê   <scope>runtime</scope>
Ê </dependency>
</dependencies>
```

Once you've added the AsciidoctorJ EPUB3 jar to your classpath, you can set the backend attribute to epub3. The document will be converted to the EPUB3 format.

> **!** The *asciidoctor-epub3* gem is alpha. While it can be used successfully, there may be bugs and its functionality may change in incompatible ways before the first stable release. In other words, by using it, you are also testing it ;)

Let's see an example of how to use AsciidoctorJ with the EPUB3 converter.

spine.adoc

```
= Book Title
Author Name
:imagesdir: images  !

include::content-document.adoc[]  "
```

! The EPUB3 converter requires the value of the `imagesdir` attribute to be `images`.

" The EPUB3 converter must be run on a *spine* document that has at least one include directive (and no other body content) in order to function properly.

content-document.adoc

```
= Content Title
Author Name

[abstract]
This is the actual content.

== First Section

And off we go.
```

And finally we can convert the document to EPUB3 using AsciidoctorJ.

```
asciidoctor.convertFile(new File("spine.adoc"),
Ê                options().safe(SafeMode.SAFE).backend("epub3").get()); ! "

assertThat(new File("target/test-classes/index.epub").exists(), is(true));
```

! Currently, the EPUB3 converter must be run in `SAFE` or `UNSAFE` mode due to a bug

" `epub3` is the name of the backend that must be set to convert to EPUB3.

Loading Ruby libraries

Simple extensions may be fully implemented in Java, but if you want to create complex extensions you can mix Ruby and Java code. This means that you may need to execute a Ruby file or a RubyGem (i.e., gem) inside your extension.

To load a Ruby file inside the Ruby runtime, you can use `org.asciidoctor.internal.RubyUtils.loadRubyClass(Ruby, InputStream)`. You can also load a gem using an API that wraps Ruby's `require` command. The gem must be available inside the classpath. Next run `org.asciidoctor.internal.RubyUtils.requireLibrary(Ruby, String)`, passing the name of the gem as the second argument.

JRuby instance

Sometimes you may need the Ruby runtime used inside AsciidoctorJ. One reason is because you are using JRuby outside AsciidoctorJ and you want to reuse the same instance. Another reason is that you need to instantiate by yourself an Asciidoctor Ruby object.

To get this instance you can use `org.asciidoctor.internal.JRubyRuntimeContext.get()` to get it.

GEM_PATH

By default, AsciidoctorJ comes with all required gems bundled within the jar. In certain circumstances, you may want to load gems from an external folder. To accomplish this scenario, `create` method provides a parameter to set folder where gems are present. Internally, AsciidoctorJ will set `GEM_PATH` environment variable to given path.

Example of setting GEM_PATH

```
import static org.asciidoctor.Asciidoctor.Factory.create;
import org.asciidoctor.Asciidoctor;

Asciidoctor asciidoctor = create("/my/gem/path"); !
```

! Creates an `Asciidoctor` instance with given `GEM_PATH` location.

Using AsciidoctorJ in an OSGi environment

In a non OSGi context, the following snippet will successfully create an Asciidoctor object:

```
import static org.asciidoctor.Asciidoctor.Factory.create;
import org.asciidoctor.Asciidoctor;

Asciidoctor asciidoctor = create();
```

In an OSGi context it will not work because JRuby needs some paths to find the gems (the Asciidoctor ones and the Ruby themselves ones). In order to make it work, you will need two more classes (RubyInstanceConfig and JavaEmbedUtils) and a small modification of the previous snippet of code. The modifications take care of the class loaders because in OSGi, which are a key point in OSGi:

```
import static org.asciidoctor.Asciidoctor.Factory.create;
import org.asciidoctor.Asciidoctor;

RubyInstanceConfig config = new RubyInstanceConfig();
config.setLoader(this.getClass().getClassLoader()); !

JavaEmbedUtils.initialize(Arrays.asList("META-INF/jruby.home/lib/ruby/2.0",
"gems/asciidoctor-1.5.6/lib"), config); "  #  $

Asciidoctor asciidoctor = create(this.getClass().getClassLoader()); %
```

! The RubyInstanceConfig will use the class loader of the OSGi bundle ;

" The JavaEmbedUtils will specify the load paths of the required gems. If they are not specified, you will get JRuby exceptions ;

`META-INF/jruby.home/lib/ruby/2.0` specifies where the Ruby gems are located. Actually this path is located inside the `jruby-complete-<version>.jar` file. Without having this path specified you may get an `org.jruby.exceptions.RaiseException: (LoadError) no such file to load¦Ñ¦set` error ;

$ `gems/asciidoctor-<version>/lib` specifies where the gems for Asciidoctor are located. Actually this path is located inside the `asciidoctorj-<version>.jar` file ;

% The factory for the Asciidoctor object also specify the class loader to use.

We consider this code to be placed inside an OSGi bundle

This solution has pros and cons:

¥ *Pros:* you don't need to extract the gems located in the asciidoctorj binary ;

¥ *Cons:*

! the version of asciidoctor is hard coded ;

! the version of ruby is hard coded.

Optimization

JRuby may start slower than expected versus the C-based Ruby implementation (MRI). Fortunately, JRuby offers flags that can improve the start time and tune applications. Several Java flags can also be used in conjunction with or apart from the JRuby flags in order to improve the start time even more.

For small tasks such as converting an AsciiDoc document, two JRuby flags can drastically improve the start time:

Table 6. JRuby flags

Name	Value
`jruby.compat.version`	RUBY1_9
`jruby.compile.mode`	OFF

Both flags are set by default inside AsciidoctorJ.

The Java flags available for improving start time depend on whether your working on a 32- or 64-bit processor and your JDK version. These flags are set by using the `JRUBY_OPTS` environment variable. Let's see a summary of these flags and in which environments they can be used.

Table 7. Java flags

Name	JDK
`-client`	32 bit Java
`-Xverify:none`	32/64 bit Java
`-XX:+TieredCompilation`	32/64 bit Java SE 7
`-XX:TieredStopAtLevel=1`	32/64 bit Java SE 7

Setting flags for Java SE 6

```
export JRUBY_OPTS="-J-Xverify:none -J-client" !
```

! Note that you should add `-J` before the flag.

You can find a full explanation on how to improve the start time of JRuby applications in Optimization.

Running AsciidoctorJ on WildFly AS

If you want to use AsciidoctorJ in an application deployed on *WildFly AS*, you have to install AsciidoctorJ as a JBoss Module.

Follow the steps below:

1. Create an Asciidoctor module for *WildFly AS*.
2. Create the following folder tree: *$JBOSS_HOME/modules/org/asciidoctor/main*.
3. Create the module descriptor file *module.xml*.

   Asciidoctor module descriptor for WildFly AS

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.0" name="org.asciidoctor">
Ê   <resources>
Ê       <resource-root path="asciidoctorj-1.5.6.jar"/>
Ê       <resource-root path="jcommander-1.35.jar"/>
Ê       <resource-root path="jruby-complete-1.7.26.jar"/>
Ê   </resources>
Ê   <dependencies>
Ê       <module name="sun.jdk" export="true">
Ê           <imports>
Ê               <include path="sun/misc/Unsafe" />
Ê           </imports>
Ê       </module>
Ê       <module name="javax.management.j2ee.api"/>
Ê       <module name="javax.api"/>
Ê       <module name="org.slf4j"/>
Ê   </dependencies>
</module>
```

4. Copy the jar files into the same folder as the *module.xml* file.
5. Make sure the version numbers of the jar files agree with what's in the current set. Restart WildFly for the new module to take effect.
6. Add a dependency on your Java archive to this WildFly module using one of the following options:

   a. Add the dependency just into the *MANIFEST.MF* file.

      MANIFEST.MF file example with dependency to Asciidoctor module

```
Manifest-Version: 1.0
Dependencies: org.asciidoctor
...
```

   b. *Or*, configure the dependency into the *pom.xml* with the *Maven JAR/WAR plugin*.

pom.xml file example with Maven WAR plugin configuration to add a dependency

```
...
<dependencies>
Ê <dependency>
Ê   <groupId>org.asciidoctor</groupId>
Ê   <artifactId>asciidoctorj</artifactId>
Ê   <version>1.5.6</version>
Ê   <scope>provided</scope> !
Ê   ...
Ê </dependency>
</dependencies>
...
<plugin>
Ê <groupId>org.apache.maven.plugins</groupId>
Ê   <artifactId>maven-war-plugin</artifactId>
Ê   <version>2.4</version>
Ê   <configuration>
Ê     <archive>
Ê       <manifestEntries>
Ê         <Dependencies>org.asciidoctor</Dependencies> "
Ê       </manifestEntries>
Ê     </archive>
Ê   </configuration>
</plugin>
...
```

! The AsciidoctorJ dependency and the transitives dependencies don't need to be added to the final WAR since all JARs are available through the module.

" The module dependency will be added to the *MANIFEST.MF* file.

Using a pre-release version

Pre-release versions of AsciidoctorJ are published to Bintray. You can find them in https://bintray.com/asciidoctor/maven/asciidoctorj/view. Final releases are released to both Maven Central and Bintray.

Here's how to use a pre-release version in Maven:

```
<repositories>
Ê <repository>
Ê   <id>central</id>
Ê   <name>bintray</name>
Ê   <url>http://dl.bintray.com/asciidoctor/maven</url>
Ê   <snapshots>
Ê     <enabled>false</enabled>
Ê   </snapshots>
Ê </repository>
</repositories>
```

Development

AsciidoctorJ is built using Gradle. The project is structured as a multi-module build.

Project layout

The root folder is the root project and there are several subproject folders, each prefixed with *asciidoctorj-*. Each subproject produces a primary artifact (e.g., jar or zip) and its supporting artifacts (e.g., javadoc, sources, etc).

The subprojects are as follows:

asciidoctorj
: The main Java bindings for the Asciidoctor RubyGem (asciidoctor). Also bundles optional RubyGems needed at runtime, such as coderay, tilt, haml and slim. Produces the asciidoctorj jar.

asciidoctorj-distribution
: Produces the distribution zip that provides the standalone `asciidoctorj` command.

asciidoctorj-epub3
: Bundles the Asciidoctor EPUB3 RubyGem (asciidoctor-pdf) and its dependencies as the asciidoctorj-epub3 jar.

asciidoctorj-pdf
: Bundles the Asciidoctor PDF RubyGem (asciidoctor-pdf) and its dependencies as the asciidoctorj-pdf jar.

The Gradle build is partitioned into the following files:

```
build.gradle
gradle.properties
settings.gradle
gradle/
Ê wrapper/
Ê   ...
Ê deploy.gradle
Ê eclipse.gradle
Ê idea.gradle
Ê publish.gradle
Ê sign.gradle
asciidoctorj-core/
Ê build.gradle
asciidoctorj-distribution/
Ê build.gradle
asciidoctorj-epub3/
Ê build.gradle
asciidoctorj-pdf/
Ê build.gradle
```

Build the project

You invoke Gradle on this project using the `gradlew` command (i.e., the Gradle Wrapper).

> " We strongly recommend that you use Gradle via the Gradle daemon.

To clone the project, compile the source and build the artifacts (i.e., jars) locally, run:

```
$ git clone https://github.com/asciidoctor/asciidoctorj
Ê cd asciidoctorj
Ê ./gradlew assemble
```

You can find the built artifacts in the *asciidoctorj-*/build/libs* folders.

To execute tests when running the build, use:

```
$ ./gradlew build
```

To only execute the tests, run:

```
$ ./gradlew check
```

You can also run tests for a single module:

```
$ cd asciidoctorj-core
Ê ../gradlew check
```

To run a single test in the asciidoctorj-core subproject, use:

```
$ ../gradlew -Dsingle.test=NameOfTestClass test
```

To create the distribution, run:

```
$ ./gradlew distZip
```

You can find the distribution in the *asciidoctorj-distribution/build/distributions* folder.

To install a built project (e.g. asciidoctorj-epub3) into your local M2 repository, run:

```
$ ./gradlew asciidoctorj-epub3:publishToMavenLocal -Pskip.signing
```

Develop in an IDE

IntelliJ IDEA

To import the project into IntelliJ IDEA 14, simply import the project using the import wizard. For more information, see the Gradle page in the IntelliJ IDEA Web Help.

Eclipse

To open the project in Eclipse, first generate the Eclipse project files:

```
$ cd asciidoctorj-core
Ê ./gradlew eclipse
```

Then, import the project into Eclipse using File Ý Import Ý General Ý Existing Project into Workspace.

Continuous integration

Continuous integration for the AsciidoctorJ project is performed by Travis CI. You can find recent build results, including the build status of pull requests, on the asciidoctor/asciidoctorj page.

Publish the artifacts

Artifacts are published to Maven Central and jCenter by way of Bintray's *Distribution as a Service*

platform.

Before publishing, you need to configure your gpg signing and Bintray credentials. Create the file *$HOME/.gradle/gradle.properties* and populate the following properties.

```
signing.keyId=
signing.password=
signing.secretKeyRingFile=/home/YOUR_USERNAME/.gnupg/secring.gpg
bintrayUsername=
bintrayApiKey=
```

To build, assemble and sign the archives (jars and distribution zip), run:

```
$ ./gradlew -PpublishRelease=true signJars
```

" The `publishRelease=true` property is technically only required if the version is a snapshot.

To build, assemble (but not sign) and install the archives (jars and distribution zip) into the local Maven repository, run:

```
$ ./gradlew -PpublishRelease=true install
```

To build, assemble, sign and publish the archives (jars and distribution zip) to Bintray, run:

```
$ ./gradlew clean
Ê ./gradlew -i -x pMNPTML bintrayUpload
```

! Don't run the `clean` task in the same execution as the `bintrayUpload` because it will not upload one of the signatures.

If you want to first perform a dry run of the upload, add the `dryRun=true` property.

```
$ ./gradlew -i -PdryRun=true -x pMNPTML bintrayUpload
```

$ The `-x pMNPTML` is necessary to work around a bug in the publishing plugin that prevents it from signing the archives.

Bintray does not allow you to publish snapshots. You have to first update the version in *gradle.properties* to a release (or pre-release) version number. Currently, Gradle is not configured to automatically tag a release, so you have to create the git tag manually.

Resources

The source code for AsciidoctorJ, including the latest developments and issues, can be found in the project's repository on GitHub. If you identify an issue while using AsciidoctorJ, please don't hesitate to file a bug report. Also, don't forget to join the Asciidoctor discussion list, where you can ask questions and leave comments.