Hitchhiker’s Guide to Docs-As-Code (Folge 7)

Beautiful Code

Ein halbes Dutzend Folgen dieser Kolumne, bevor wir endlich zum Thema Sourcecode kommen: Wir möchten in Architekturdokumentation an manchen Stellen Code integrieren, etwa zur Beschreibung wichtiger Schnittstellen. AsciiDoc bietet hierfür schicke Features. Wie immer finden Sie den Quellcode dieser Kolumne online unter [1]. Unsere Beispiele übersetzen wir (wie in den vorigen Folgen) mit Gradle, und wieder haben wir einen default-task für asciidoctor konfiguriert:

> cd folge-7

> gradle

Der Asciidoc-Prozessor erzeugt den HTML-Output im Verzeichnis build/asciidoc/html5/, auch das kennen Sie ja schon.

**Code in der (Architektur-)Doku?**

Auch wenn Architekturdokumentation für Überblick im Großen sorgen soll, benötigen wir an manchen Stellen doch detaillierte Infos auf Ebene von Quellcode. Daher möchten wir an manchen Stellen Code(-fragmente) in die Architekturdokumentation integrieren, beispielsweise für folgende Anwendungsfälle:

* Schnittstellen
* Erläuterung technischer Konzepte,
* spezielle Lösungsansätze für (wiederkehrende) Aufgaben mit Architekturrelevanz

Schauen wir uns an, wie das mit AsciiDoctor grundsätzlich funktioniert:

**Sourcecode anzeigen**

Bevor wir mit “echtem” Sourcecode loslegen, müssen wir AsciiDoc mitteilen, welche Bibliothek wir für Syntax-Highlighting verwenden möchten - es werden z.B. Coderay, highlightjs, pygments und diverse andere *Highlighter* unterstützt. Fügen Sie im Header Ihres AsciiDoc-Dokumentes folgende Deklaration ein:

= HHGDAC: Sourcecode integrieren

:source-highlighter: coderay

**<Anmerkung für’s Layout: diese beiden Zeilen bitte in den Fließtext, KEINEN eigenständigen Listing-Block dafür!>**

Die konkrete Wahl des Highlighters (siehe [6]) hängt hauptsächlich von den benötigten Ausgabeformaten und den in Ihrem Projekt verwendeten Sprachen ab.

So - jetzt sind wir bereit für *real stuff*: In AsciiDoc zeichnen Sie Sourcecode durch die Angabe von *[source, <language>]* aus. Der nachfolgende Block wird dann als Code-Block formatiert (die *----* markieren Anfang und Ende eines Blockes):

[source,python]

----

# This program prints Hello, world!

print('Hello, world!')

----

**<Anmerkung für’s Layout: auch die vorstehenden 5 Zeilen bitte in den Fließtext, KEINEN eigenständigen Listing-Block dafür!>**

Das Ergebnis finden Sie in Abbildung 1.



folge-7-bild-1.tif

Abbildung 1: So wird Code gerendert

**Die Sprache festlegen**

Um das Syntax-Highlighting von AsciiDoctor zu verbessern, können Sie die Programmiersprache als zweiten Parameter in der *[source, <language>]* Deklaration angeben - AsciiDoc (beziehungsweise die jeweiligen Source-Highlighter) können sich dann (noch) besser auf die jeweilige Syntax einstellen. In Listing 1 finden Sie das passende Beispiel - ausnahmsweise mal für die Sprache Go. Unserer Erfahrung nach sind die gängigen Highlighter relativ gut darin, die Sprache selbst zu erkennen - in vielen Fällen können Sie sich die explizite Deklaration also sparen. Probieren Sie’s einfach für Ihre Listings aus.

**Listing 1: andere Programmiersprachen (hier: Go)**

.Go gefällig?

[source,go]

----

package main

import "fmt"

func main() {

fmt.Println("hello world")

}

----

**Ende Listing 1**

Jetzt haben Sie Sourcecode in Ihrer Dokumentation, aber oftmals möchten Sie bei der Erklärung Bezug auf bestimmte Stellen dieses Codes nehmen - was am einfachsten ganz klassisch über Zeilennummern in diesen Listings geht. Dazu nehmen Sie einfach nur den Parameter 'linenums' in die ‘[source]’ Deklaration auf - ein Beispiel zeigt Listing 2, das fertig gerenderte Ergebnis finden Sie in Abbildung 2! Leider funktionieren die Zeilennummern nicht in diversen Preview-Funktionen bei AsciiDoc-Editoren, u.a. auch beim Atom-Editor und Github-Preview. Das Plugin für IntelliJ IDEA bildet hier eine positive Ausnahme.

**Listing 2: Zeilennummern**

.Oder JavaScript?

[source,javascript,linenums]

--

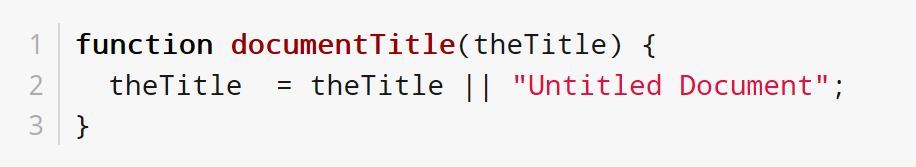
function documentTitle(theTitle) {

theTitle = theTitle || "Untitled Document";

}

--

**Listing 2: Zeilennummern**



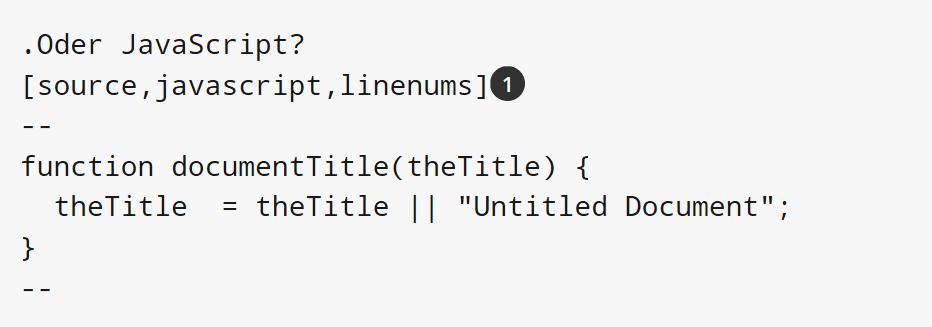
folge-7-bild-2.tiff

Abbildung 2: Listing mit Zeilennummern

Neben diesem klassischen Ansatz bietet AsciiDoctor allerdings noch die (elegante) Möglichkeit der sogenannten Callouts - das sind visuelle Hervorhebungen, die beim Rendering eingebracht werden. Betrachten Sie Abbildung 3 - darin finden Sie einen solchen Callout. Nach typischer AsciiDoc Manier können Sie solche Hervorhebungen ganz einfach vornehmen - indem Sie in spitzen Klammern am Ende einer Zeile im jeweiligen Sourcecode eine Ziffer setzen - beispielsweise wie folgt:

[source,javascript,linenums] <1>

**<Anmerkung für’s Layout: auch die vorstehende Zeile bitte in den Fließtext, KEINEN eigenständigen Listing-Block dafür!>**



folge-7-bild-3.tiff

Abbildung 3: Sourcecode mit Callout

Im Regelfall haben Sie natürlich Listings, in denen mehrere Zeilen interessant sind - ein Beispiel aus der Fantasy-Welt zeigt Listing 3. Darin haben wir drei Callouts untergebracht, sowie die zugehörigen Erklärungen - in AsciiDoc folgen die unmittelbar hinter der abschließenden Block-Begrenzung “----”.

**Listing 3: mehrere Callouts**

[[listing-ref-example]]

.Code mit Erklärungen

[source,java]

--

import org.harryp.Voldemort; // <1>

import org.harryp.Harry; // <2>

public class Stakeholder {

public String wandName; // <3>

// ...

}

--

<1> diesen Namen besser vermeiden

<2> der Knabe mit der Brille

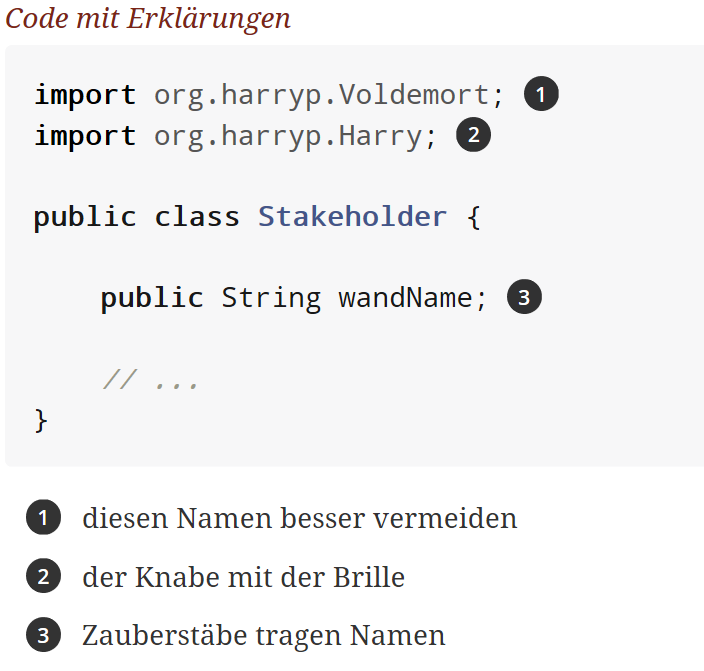
<3> Zauberstäbe tragen Namen

**Ende Listing 3**

Wie sie in Listing 3 auch sehen können, unterstützt AsciiDoc die Callouts in Kommentare zu setzen, damit der Code weiterhin korrekt ist. Die Kommentarzeichen werden beim Rendern des Blocks jedoch weggelassen (siehe Abbildung 4). Das ist wichtig, wenn Sie - wie im nächsten Abschnitt erklärt - den Sourcode nicht in der Dokumentation duplizieren wollen.

folge-7-bild-3.tiff

Abbildung 3: Sourcecode mit Callout



folge-7-bild-4.tiff

Abbildung 4: komplexeres Callout Beispiel

**Code aus dem Repository**

Sourcecode mit Copy-und-Paste in die Dokumentation zu übernehmen verletzt das DRY (don’t repeat yourself) Prinzip - das sollten Sie daher tunlichst vermeiden. Wie aber soll der Code denn in die Doku kommen, wenn wir ihn dort nicht hineinkopieren sollen? Sie haben es sicherlich erwartet, AsciiDoc bietet auch für dieses Problem eine elegante und produktive Lösung: Sie *inkludieren* Code einfach aus Ihrem Repository (genau genommen aus einer Working-Copy, auf die der jeweilige Build Zugriff hat). Im Grunde funktioniert das genauso wie das bereits vorgestellte Konzept der modularisierten Dokumentation - wir verwenden den schon bekannten ‘include’ Befehl.

Zusätzlich schlagen wir vor, den Pfad zum Code-Repository als eine Variable zu definieren.

[source, groovy]

----

include::{main\_sourcepath}/org/aim42/CheckResults.groovy[]

----

**<Anmerkung für’s Layout: auch die vorstehende Zeile bitte in den Fließtext, KEINEN eigenständigen Listing-Block dafür!>**

Über die Variable *{main\_sourcepath}* wird ein include von einem beliebigen Dateipfad

ermöglicht. Für unser Beispiel haben wir *{main\_sourcepath}* zu Beginn der AsciiDoc Datei definiert:

= HHGDAC: Sourcecode integrieren

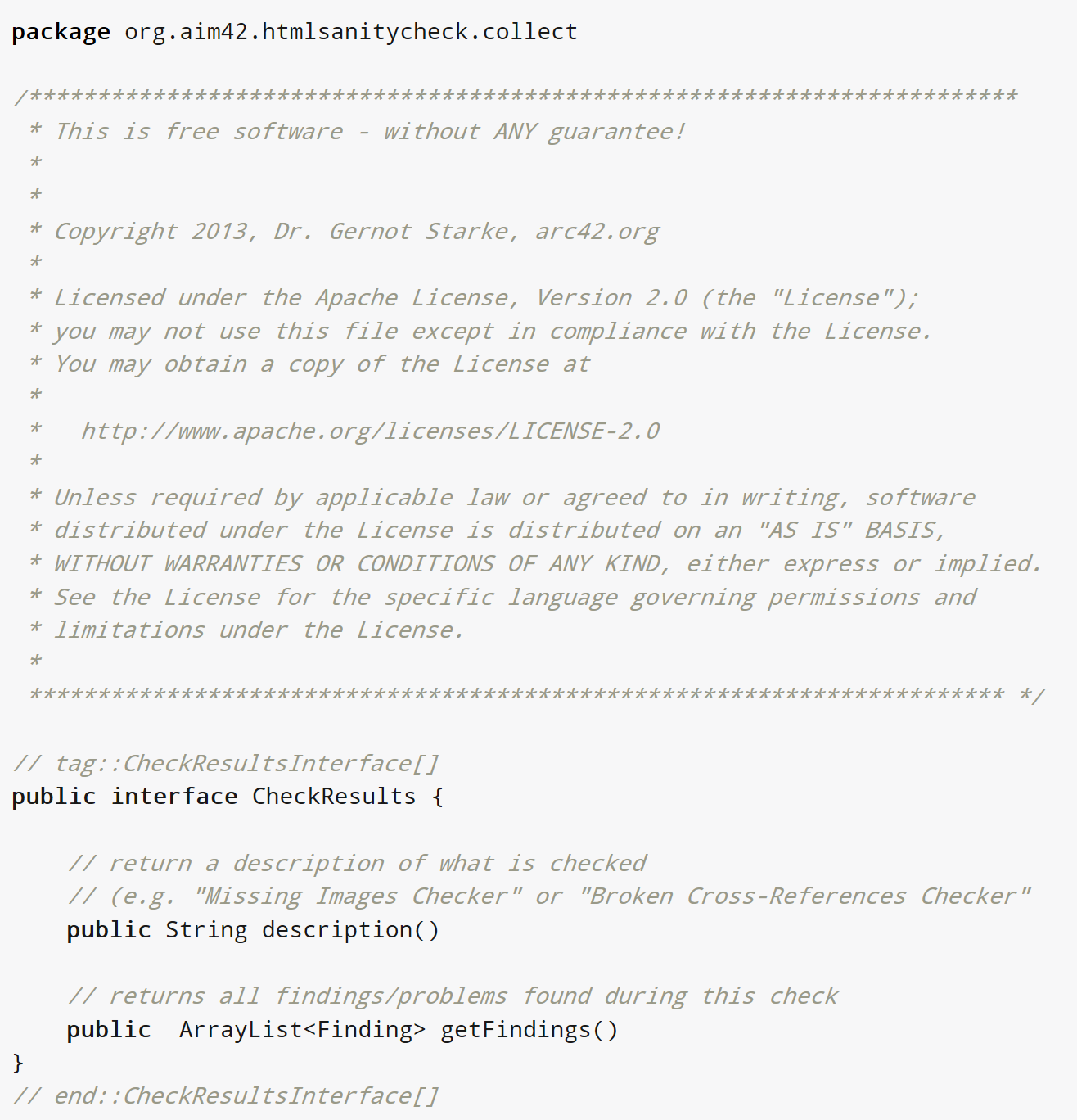
:source-highlighter: coderay

:coderay-linenums-mode: inline

:main\_sourcepath: ../main/

**<Anmerkung für’s Layout: auch die vorstehende Zeile bitte in den Fließtext, KEINEN eigenständigen Listing-Block dafür!>**

Das Ergebnis sehen Sie in Abbildung 5.



folge-7-bild-5.tiff

Abbildung 5: direkt eingebundener Code

**Pfadvariablen + Verzeichnisbäume**

Die Pfade für den `include`-Befehl des AsciiDoc Prozessors werden im Gegensatz zum `image`-Befehl relativ zum Dokument aufgelöst. Für die Strukturierung einer Dokumentation ist das praktisch - für das Einbinden von Code nicht.

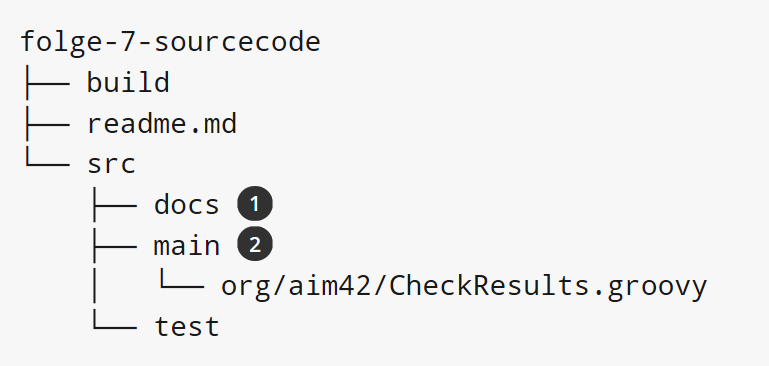
Je nachdem wo das AsciiDoc-Dokument liegt, müsste ein anderer Pfad gesetzt werden, weshalb es sich für größere Dokumentationen anbietet die Variable `main\_sourcepath` im `build.gradle` als absoluten Pfad zu setzen (siehe Abbildung 6).



folge-7-bild-6.tiff

Abbildung 5: direkt eingebundener Code

Abbildung 6 verdeutlicht dies nochmal: 1) ist unser Root-Verzeichnis, im Buildfile *build.gradle* definiert und zu 2) zeigt die Pfadvariable *main\_sourcepath*. Zum Rendern dieses Verzeichnisbaumes benutzen wir eine AsciiDoc Erweiterung namens MonoTree[4].



folge-7-bild-7.tiff

Abbildung 6: direkt eingebundener Code

**Ein Listing referenzieren**

Sie können natürlich auch bestimmte Listings Ihrer Dokumentation referenzieren. Das schreiben Sie in Asciidoc als normale Referenz in doppelten spitzen Klammern: *<<listing-ref-example>>*. Asciidoctor ersetzt dies dann gegen eine Referenz der entsprechender Überschrift als Text.

**Bestimmte Stellen aus Code inkludieren**

Oftmals genügt es, bestimmte (kurze) Teile aus Source-Files zu inkludieren. Auch dafür hat AsciiDoc eine elegante Antwort gefunden - nachfolgend sehen Sie einen Extrakt der Datei aus Listing 3 oben, diesmal beschränkt auf zwei Methoden:

include::{main\_sourcepath}/org/aim42/CheckResults.groovy[tags=CheckResultsInterface]

**<Anmerkung für’s Layout: auch die vorstehende Zeile bitte in den Fließtext, KEINEN eigenständigen Listing-Block dafür!>**

Wie Sie in Abbildung 5 sehen, haben wir dafür die betreffenden Methoden mit den Kommentaren *// tag::CheckResultsInterface[]* und *// end::CheckResultsInterface[]* markiert. Im ersten Moment ist dies gewöhnungsbedürftig. Bei der Verwendung ist es aber praktisch, da Sie im Code dadurch immer direkt erkennen können, welche Zeilen in der Dokumentation referenziert sind.

**JavaDoc mit AsciiDoc**

Wie wäre es, wenn Sie JavaDoc-Kommentare endlich nicht mehr als HTML schreiben müssten? Richtig! Auch hier können Sie ab sofort AsciiDoc einsetzen.

Der Ansatz basiert auf dem AsciiDoclet (siehe [3]), das bei Github unter der AsciiDoctor-Organisation entwickelt wird. Um diese AsciiDoc Erweiterung zu nutzen, müssen wir unser Buildfile deutlich erweitern (siehe Listing 4).

**Listing 4: erweiterung des Builds um Javadoc**

configurations {

asciidoctorDoclet

}

dependencies {

asciidoctorDoclet "org.asciidoctor:asciidoclet:1.5.4"

}

configurations {

asciidoclet

}

javadoc {

options.with {

docletpath = configurations.asciidoctorDoclet.files.asType(List)

doclet = 'org.asciidoctor.Asciidoclet'

addStringOption '-base-dir', projectDir.toString()

def attributes = [projectName: project.name,

projectVersion: project.version]

addStringOption '-attribute', attributes\*.toString().join(',')

overview = 'src/docs/overviewJavadoc.adoc'

version = true

author = true

}

}

**Ende Listing 4**

Ruft man nun `gradle javadoc` auf, so wird wie gewohnt die Dokumentation der Klassen erzeugt, wobei die einzelnen Elemente jetzt aber angenehm in AsciiDoc und nicht mehr in HTML beschrieben werden.

Die `javadoc` Dokumentation und die Architekturdokumentation können durch gegenseitige Verlinkung kombiniert werden. Dazu wird in der `javadoc` Konfiguration ein Übersichts-Dokument angegeben. Dieses Übersichtsdokument verlinkt dann auf Ihre Architekturdokumentation und umgekehrt. Theoretisch könnte hier auch direkt die Architekturdokumentation referenziert werden - aufgrund der verschieden konfigurierten AsciiDoc-Prozessoren raten wir davon jedoch ab.

**Schnittstellen dokumentieren**

Schnittstellen sind für eine Softwaredokumentation natürlich ein ganz wichtiges Thema. Verwenden Sie REST basierte Schnittstellen mit Swagger in Ihrem Projekt, dann haben wir zum Abschluss noch eine coole Option. Mit Swagger2Markup [5] - einem Open Source Projekt von Robert Winkler - wandeln Sie Ihre Swagger-Definitionen ganz einfach in AsciiDoc um und ergänzen so über Ihren Build die Architekturdokumentation!

**Fazit**

Sourcecode in AsciiDoc-Manier integrieren stellt sicher, dass Doku und Code zueinander passen (zumindest an den betroffenen Stellen). Mit dem passendem Build-Prozess können wir sicherstellen, dass der inkludierte Code seine Testfälle bestanden hat - das ist gegenüber dem herkömmlichen copy-paste eine signifikante Verbesserung.

Wenn sie das alles selbst ausprobieren möchten, finden Sie wie gewohnt unsere Beispiele im Repository, siehe [1].

In der nächsten Folge stellen wir Ihnen einige Werkzeuge vor, mit denen Sie Docs-as-Code effektiv und effizient erstellen und pflegen können. Bis dahin – happy docu-coding.

**Die Autoren**

*Ralf D. Müller arbeitet als Architekt und Entwickler – und erlebt täglich die Notwendigkeit effektiver Dokumentation. Er ist erklärter AsciiDoc-Fan und Committer bei arc42  sowie Gründer des docToolchain Projektes [2].*

*Gernot Starke (INNOQ Fellow) verbessert seit Jahren Softwarearchitekturen und deren Dokumentation. Er ist Mitgründer und Betreiber http://arc42.org und http://aim42.org.*

**Links & Literatur**

[1] <https://github.com/arc42/HHGDAC>

[2] DocToolChain - die freie Toolkette für docs-as-code, basierend auf AsciiDoc.

[3] AsciiDoclet, damit Sie endlich JavaDoc-Kommentare in AsciiDoc schreiben können: <https://github.com/asciidoctor/asciidoclet>

[4] MonoTree: https://github.com/allati/asciidoctor-extension-monotree

[5] Swagger2Markup: <https://github.com/Swagger2Markup/swagger2markup>

[6] Die verschiedenen AsciiDoc Highlighter: http://asciidoctor.org/docs/user-manual/#available-source-highlighters