



Doc as Code-Tutorial

Technische Dokumentation mit asciidoc

Alex Kurzke <alex.kurzke@hiq.de>

Version 1.0, 23.02.2022: Erste Veröffentlichung des Tutorials

Inhaltsverzeichnis

1. Ziel des Tutorials	1
2. Einführung	2
2.1. Der "Doc as Code"-Ansatz in der Technischen Dokumentation	3
2.2. Was sind vereinfachte Auszeichnungssprachen?	4
3. asciidoc und wichtige Dokumentations-Tools	5
3.1. Die asciidoc-Syntax	5
3.1.1. Basale Gestaltungselemente in asciidoc	5
3.1.2. Cheat Sheet und weitere Dokumentation	8
3.2. Der ATOM Editor	9
3.2.1. Die Packages asciidoc-preview & language-asciidoc	9
3.3. Die Textprozessoren asciidoctor & asciidoctor-pdf	11
3.3.1. asciidoctor	11
3.3.2. asciidoctor-pdf	11
4. Das erste Dokument erstellen, previewen und rendern	12
4.1. Das erste asciidoc-Dokument schreiben	12
4.2. Die HTML-Preview via asciidoc-preview nutzen	13
4.3. Eine Webseite via asciidoctor erzeugen	14
4.4. Ein PDF via asciidoctor-pdf erzeugen	14
5. Modulares Arbeiten mit asciidoc	17
5.1. Die include-Funktion nutzen	17
5.2. Die Überschriften-Hierarchie kontrollieren	20
6. Ein komplexes Dokument mit einem Kopfdokument steuern	21
6.1. Ein Kopfdokument erstellen	21
6.1.1. Globale Attribute im Dokumentenkopf festlegen	22
6.2. Das Ergebnis	23
7. Ausblick und Vertiefung	25

1. Ziel des Tutorials

Das Ziel des Tutorials ist, euch den **Doc as Code**-Ansatz in der Technischen Dokumentation näherzubringen. In erster Linie wollen wir dazu die Arbeit mit der populären "vereinfachten Auszeichnungssprache" **asciidoc** und den wichtigsten Tools erlernen.



Auch dieses Tutorial wurde natürlich in **asciidoc** geschrieben. Die Rohdaten findet ihr in folgendem GIT-Repository:

https://github.com/peevey/doc_as_code_tutorial

Im GIT sind an der im README angegebenen Stelle auch Exporte dieses Tutorials als HTML und als PDF zu finden.

2. Einführung

Viele von euch haben es sicherlich schon bemerkt: Ruft man heutzutage eine Technische Dokumentation auf, befindet sich auf vielen Seiten ein Button, der zum Bearbeiten der aktuellen Seite einlädt. Klickt man auf diesen, wird man in der Regel auf ein GIT-Repository verwiesen, wo die gerade gelesene Datei nicht in unübersichtlichem HTML, sondern meist in einem einfachen und übersichtlichen Format vorliegt.

Schauen wir uns einmal ein konkretes Beispiel an:



<https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/types/generics>

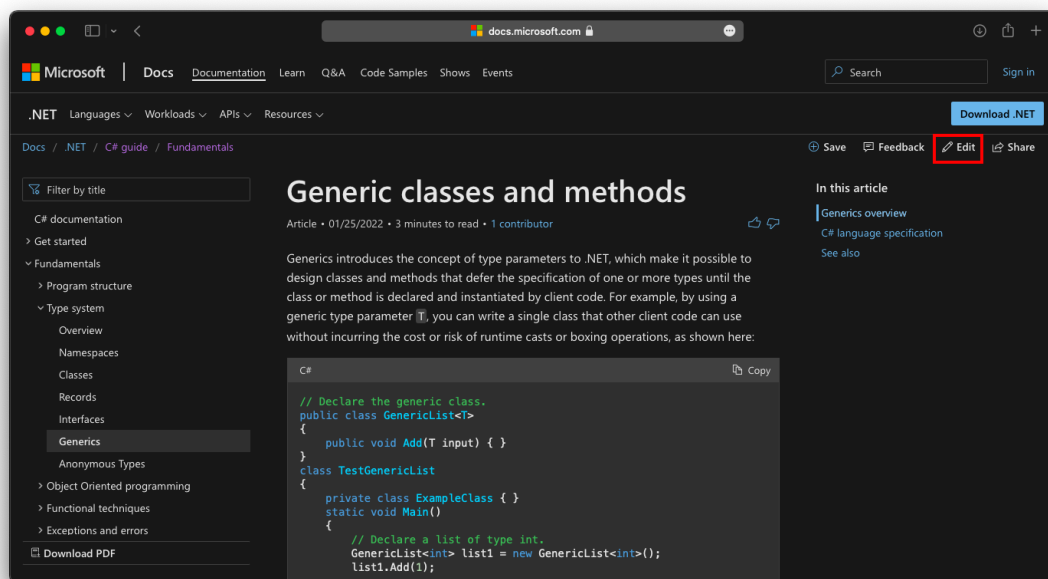


Abb. 1. Beispiel: Eine Seite der Microsoft-Dokumentation. Oben rechts sieht man den Button zum Bearbeiten der Seite.

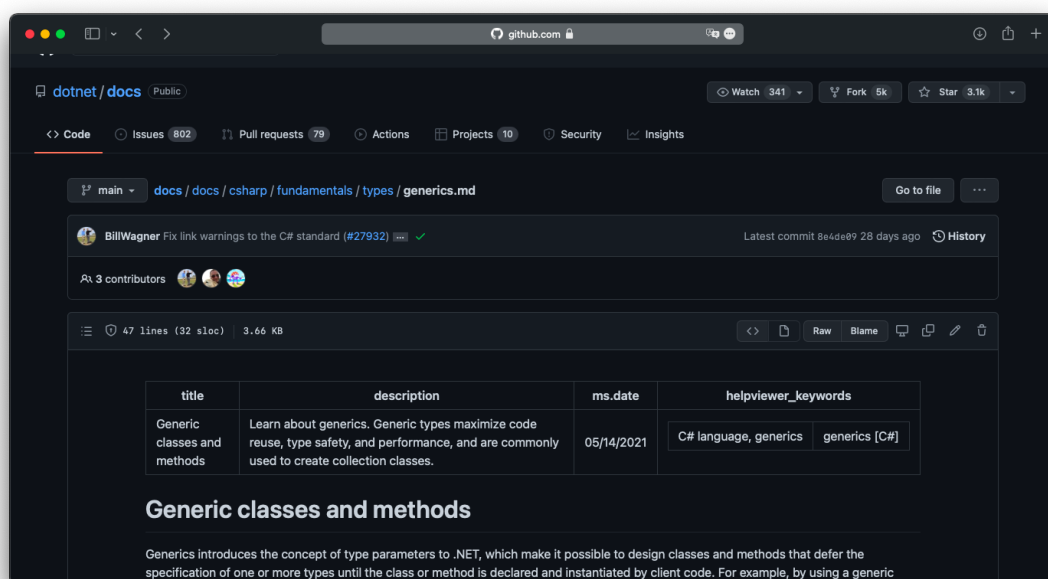
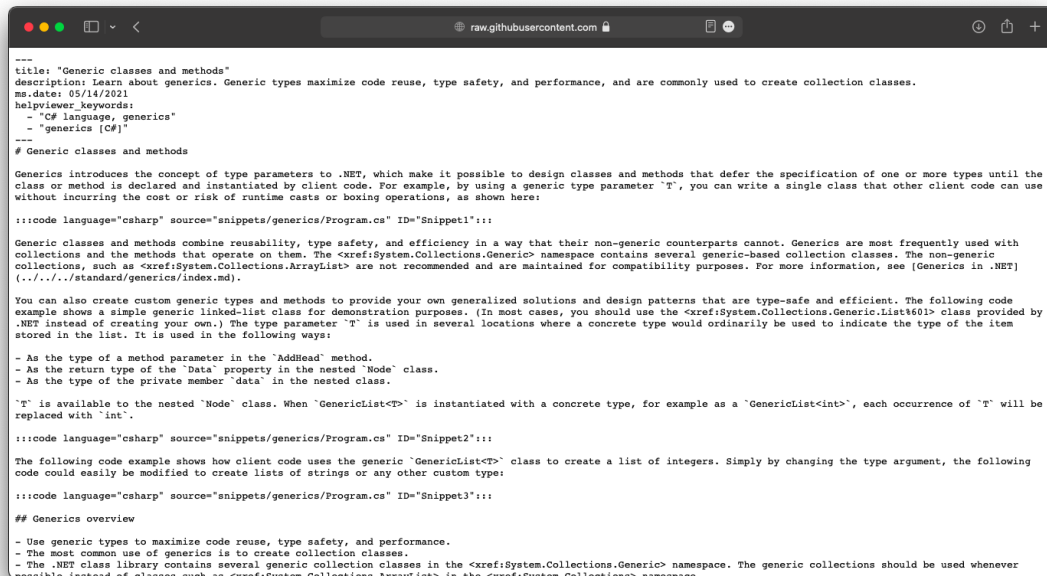


Abb. 2. Dieselbe Seite im GIT, wo sie theoretisch von jedem bearbeitet werden kann.



```
---
title: "Generic classes and methods"
description: Learn about generics. Generic types maximize code reuse, type safety, and performance, and are commonly used to create collection classes.
ms.date: 05/14/2021
helpviewer_keywords:
  - "C# language, generics"
  - "generics [C#]"
---
# Generic classes and methods

Generics introduces the concept of type parameters to .NET, which make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code. For example, by using a generic type parameter `T`, you can write a single class that other client code can use without incurring the cost or risk of runtime casts or boxing operations, as shown here:

:::code language="csharp" source="snippets/generics/Program.cs" ID="Snippet1":::

Generic classes and methods combine reusability, type safety, and efficiency in a way that their non-generic counterparts cannot. Generics are most frequently used with collections and the methods that operate on them. The <xref:System.Collections.Generic> namespace contains several generic-based collection classes. The non-generic collections, such as <xref:System.Collections.ArrayList> are not recommended and are maintained for compatibility purposes. For more information, see [Generics in .NET](<xref:../standard/generics/index.md>).

You can also create custom generic types and methods to provide your own generalized solutions and design patterns that are type-safe and efficient. The following code example shows a simple generic linked-list class for demonstration purposes. (In most cases, you should use the <xref:System.Collections.Generic.ListT601> class provided by .NET instead of creating your own.) The type parameter `T` is used in several locations where a concrete type would ordinarily be used to indicate the type of the item stored in the list. It is used in the following ways:

- As the type of a method parameter in the `AddHead` method.
- As the return type of the `Data` property in the nested `Node` class.
- As the type of the private member `data` in the nested class.

`T` is available to the nested `Node` class. When `GenericList<T>` is instantiated with a concrete type, for example as a `GenericList<int>`, each occurrence of `T` will be replaced with `int`.

:::code language="csharp" source="snippets/generics/Program.cs" ID="Snippet2":::

The following code example shows how client code uses the generic `GenericList<T>` class to create a list of integers. Simply by changing the type argument, the following code could easily be modified to create lists of strings or any other custom type:

:::code language="csharp" source="snippets/generics/Program.cs" ID="Snippet3":::

## Generics overview

- Use generic types to maximize code reuse, type safety, and performance.
- The most common use of generics is to create collection classes.
- The .NET class library contains several generic collection classes in the <xref:System.Collections.Generic> namespace. The generic collections should be used whenever possible instead of classes such as <xref:System.Collections.ArrayList> in the <xref:System.Collections> namespace.
```

Abb. 3. Dieselbe Seite raw im **markdown**-Format, einer gängigen vereinfachten Markup-Language.

2.1. Der "Doc as Code"-Ansatz in der Technischen Dokumentation

Was heißt das?

Ganz offensichtlich sehen wir beim Aufbau der Technischen Dokumentation Parallelen zum Umgang mit Programmcode:

1. Die Dokumentation wird in einem GIT verwaltet und dort versioniert und kann nach den etablierten Verfahren wie Commits, Pull Requests und Reviews bearbeitet werden.
2. Die Dokumentation wird in einem Rohformat geschrieben, das in seinen Strukturen einer Programmiersprache ähnelt und unabhängig vom Ausgabeformat HTML ist.
3. Und: Die finale Dokumentation wird aus den Rohdaten bei jeder Veröffentlichung neu kompiliert, so wie ein Programm aus seinem Programmcode.

Dahinter steht eine Philosophie, die man als "Doc as Code" bezeichnet. "Doc as Code" heißt: Schreibe deine Dokumentation mit denselben Methoden und Tools wie deinen Programmcode.

Ein besonderer Vorteil dieser Vorgehensweise liegt darin, dass sie die Dokumentationstätigkeit nah an die Arbeitsrealität der Entwickler bringt. Anstatt mit "klassischen" Dokumentationstools wie Word, Confluence, etc. arbeiten zu müssen, können Entwickler ihren Beitrag zur Dokumentation mit den Tools leisten, mit denen sie ohnehin tagtäglich arbeiten. Elementare Voraussetzung dafür ist die Verwendung einer "Vereinfachten Auszeichnungssprache".



Vertiefende Lektüre

<https://www.writethedocs.org/guide/docs-as-code/>

<https://www.heise.de/hintergrund/Documentation-as-Code-mit-Asciidoctor-4642013.html>

2.2. Was sind vereinfachte Auszeichnungssprachen?

Wahrscheinlich ist schon jeder einmal mit einer vereinfachten Auszeichnungssprache in Kontakt gekommen, ohne es zu bemerken; sei es beim Schreiben von Issues in Jira oder beim Formatieren einer Textnachricht in einem Messenger.

Vereinfachte Auszeichnungssprachen (**lightweight markup languages**, bzw. **simple markup languages**) sind - wie der Name schon sagt – wie XML, HTML oder YAML Markup-Languages. Der Fokus vereinfachter Auszeichnungssprachen liegt dabei auf einer besonders einfachen Syntax, die auch im Rohformat leicht zu lesen und zu schreiben ist, so dass sie in der Regel in jedem Text Editor (und in jeder IDE) komfortabel bearbeitet werden können.

Die Konvertierung in die gewünschten Ausgabeformate wie HTML erfolgt grundsätzlich in einem zweiten Schritt, so dass der Fokus auf den Text und dessen Struktur gelegt werden kann.

Bekannte vereinfachte Auszeichnungssprachen sind:

- AsciiDoc
- Markdown
- reStructuredText

Im Rahmen dieses Tutorials werden wir uns **asciidoc** genauer anschauen.

3. asciidoc und wichtige Dokumentations-Tools

Die Ziele dieses Abschnitts sind:

- ☑ einfache Formatierungsmöglichkeiten in **asciidoc** kennenzulernen
- ☑ den **ATOM** Editor für die Arbeit mit **asciidoc** einzurichten
- ☑ die Textprozessoren **asciidoc** und **asciidoc-pdf** zu installieren, mit denen wir aus unseren **asciidoc**-Dokumenten die gewünschten Zielformate wie HTML5 oder PDF erstellen können.

3.1. Die asciidoc-Syntax

Die Möglichkeiten, die **asciidoc** zur Texterstellung bietet, sind schier endlos. Um ein grundlegendes Verständnis der Schauen wir uns als Erstes ein paar einfache Formatierungsmöglichkeiten in **asciidoc** an.

3.1.1. Basale Gestaltungselemente in asciidoc

Einfache Text-Formatierungen

```
*fett* ①  
_kursiv_ ②  
'mono' ③  
#markiert ④  
[.underline]#unterstrichen ⑤  
[.line-through]#durchgestrichen ⑥  
^hochgestellter^ Text ⑦  
~tiefgestellter~ Text ⑧
```

① **fett**

② *kursiv*

③ `mono`

④ **markiert**

⑤ unterstrichen

⑥ ~~durchgestrichen~~

⑦ ^{hochgestellter} Text

⑧ _{tiefgestellter} Text

Strukturierende Überschriften

```
= Der Titel des Dokuments ①
== Eine Überschrift erster Ordnung ②
=== Eine Überschrift zweiter Ordnung ③
==== Eine Überschrift dritter Ordnung
===== Eine Überschrift vierter Ordnung
===== Eine Überschrift fünfter Ordnung
== Eine weitere Überschrift erster Ordnung
```

① **Wichtig:** Da ein Dokument nur *einen* Titel haben kann, darf diese Art der Überschrift nur einmal im gesamten Dokument verwendet werden. Ansonsten kann das **asciidoc**-Dokument nicht in die Zielformate gerendert werden. Das fällt insbesondere dann ins Gewicht, wenn man seine Dokumentation aus verschiedenen Dateien zusammenstellt. Die Verwendung eines Dokumententitels ist aber optional.

② Das html-Äquivalent dieser Überschrift ist `<h1></h1>`.

③ Das html-Äquivalent dieser Überschrift ist `<h2></h2>`, usw.

URLs und Links

Links erzeugen wir, in dem wir sie einfach als solche in unseren Text einfügen:

```
https://hiq.de
```

<https://hiq.de>

Wenn wir einen bestimmten Text mit einem Link hinterlegen wollen, geht das so:

```
https://hiq.de[HiQ Projects GmbH]
```

[HiQ Projects GmbH](https://hiq.de)

Alphanumerische Listen

```
. Punkt 1
. Punkt 2
.. Unterpunkt 2.1
... Noch ein Unterpunkt
```

1. Punkt 1

2. Punkt 2

a. Unterpunkt

i. Noch ein Unterpunkt

Quellcode-Beispiele

```
[source,java]
----
public class HelloWorld
{
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
----
```

```
public class HelloWorld
{
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Infoboxen (Admonition Blocks)

Infoboxen eignen sich, um die Aufmerksamkeit des Lesers auf besonders wichtige oder interessante Informationen zu lenken. **AsciiDoc** unterstützt von Haus aus verschiedene Arten von Infoboxen, die sehr leicht zu erzeugen sind. Folgende Arten von Infoboxen sind möglich: **[NOTE]**, **[TIP]**, **[IMPORTANT]**, **[WARNING]** & **[CAUTION]**.

[WARNING]

Ein Text, auf den der Leser mittels eines Hinweis-Icons hingewiesen werden soll.



Ein Text, auf den der Leser mittels eines Warnhinweis hingewiesen werden soll.

3.1.2. Cheat Sheet und weitere Dokumentation

Einen Überblick über die wichtigsten Syntax-Eigenschaften von **asciidoc** bietet folgendes Cheat Sheet.



Cheat Sheet

<https://docs.asciidoctor.org/asciidoc/latest/syntax-quick-reference/>

Für die Zwecke dieses Tutorials reichen die hier angegebenen Hinweise vollkommen aus. Wer aber wirklich tief in **asciidoc** einsteigen möchte, findet eine umfängliche Dokumentation hier:



asciidoc-Dokumentation

<https://docs.asciidoctor.org/asciidoc/latest/>

3.2. Der ATOM Editor

Asciidoc-Dateien können wir in jedem beliebigen Plain Text Editor schreiben. Für unser Vorhaben eignet sich aber besonders der **ATOM**-Editor. Der Editor bietet zahlreiche Vorteile, bspw. können wir mit **ATOM** unsere Dokumentation in einem GIT-Repository verwalten.

Für uns zum jetzigen Zeitpunkt aber am wichtigsten: Der **ATOM**-Editor kann durch verschiedenste Packages individualisiert und an spezifische Bedürfnisse angepasst werden – darunter auch explizit für die Arbeit mit **asciidoc**.



<https://atom.io>

3.2.1. Die Packages asciidoc-preview & language-asciidoc

Sobald wir **ATOM** installiert haben, müssen wir noch die notwendigen Packages hinzufügen.

Dafür öffnen wir den Package-Manager (unter "Settings") und suchen nach "asciidoc".

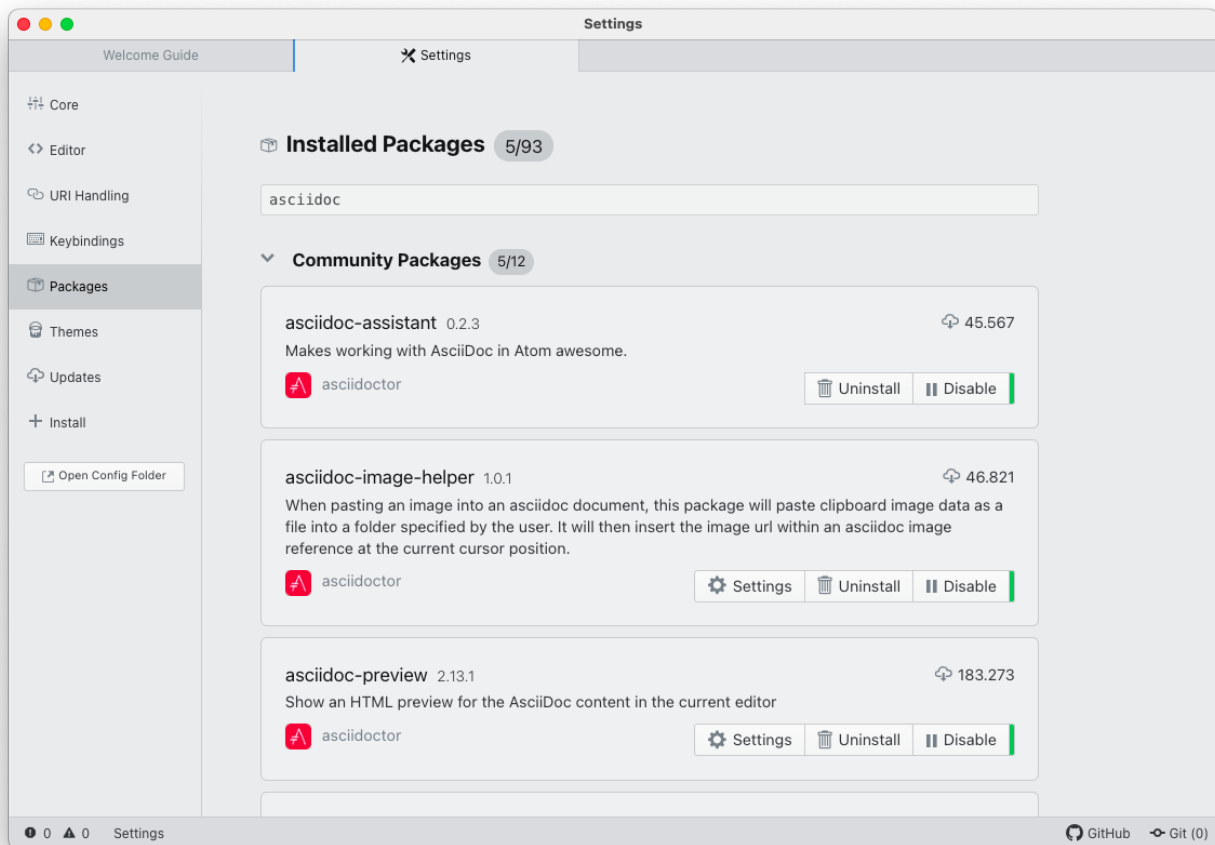


Abb. 4. Der Package-Manager von ATOM

Hier installieren wir mindestens die Packages **language-asciidoc** und **asciidoc-preview**.



Alternativ kann auch das Package **asciidoc-assistant** installiert werden, das die wichtigsten **asciidoc**-Erweiterungen für ATOM in einem Package bündelt und **language-asciidoc** und **asciidoc-preview** bereits beinhaltet.

Durch **language-asciidoc** wird die **asciidoc**-Syntax im **ATOM**-Editor farblich hervorgehoben, was die Arbeit sehr viel einfacher macht.

asciidoc-preview wiederum erlaubt unter anderem die HTML-Vorschau von **asciidoc**-Dokumenten in **ATOM**. Sobald wir in **ATOM** eine Datei mit der Endung **.adoc** geöffnet haben, können wir uns mittels **Shift** + **Strg** + **A** bzw. **Shift** + **Cmd** + **A** die HTML-Vorschau der aktuellen Seite anzeigen lassen.

Auch der Export der HTML-Datei ist möglich: Bei Rechtsklick auf die HTML-Vorschau öffnet sich ein Kontextmenü, über das wir unser Dokument mittels **Save as HTML...** exportieren können.



Der Export nach HTML mittels **asciidoc-preview** reicht für unsere Zwecke zwar zunächst aus, präzisere Ergebnisse erreicht man aber, indem man die **asciidoc**-Dokumente direkt über die Konsole mittels **asciidoctor**, bzw. **asciidoctor-pdf** rendert. Insbesondere der Export als PDF ist über **asciidoc-preview** nicht immer zuverlässig und setzt ohnehin voraus, dass **asciidoctor-pdf** auf dem Rechner installiert ist. Siehe [Die Textprozessoren asciidoctor & asciidoctor-pdf](#).

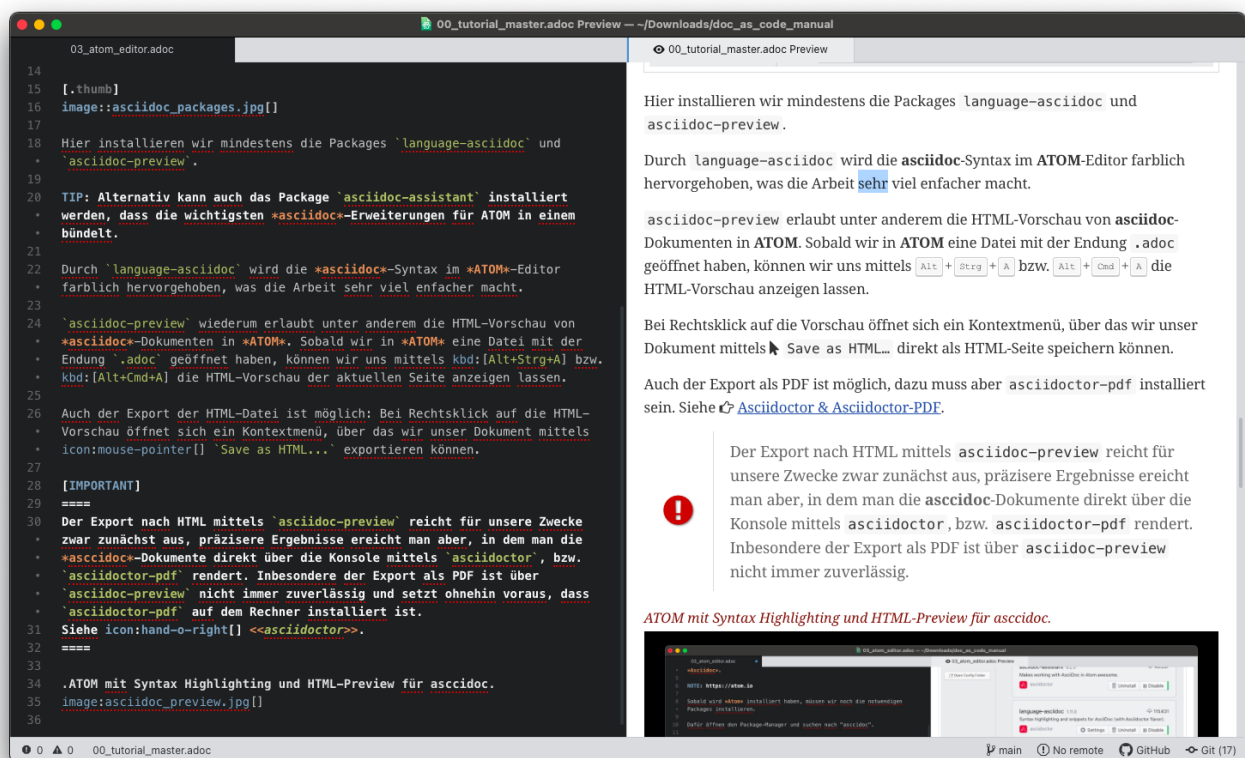


Abb. 5. Syntax Highlighting via **language-asciidoc** und HTML-Preview mit **asciidoc-preview** in **ATOM**.

3.3. Die Textprozessoren asciidoctor & asciidoctor-pdf

Asciidoc ist die vereinfachte Auszeichnungssprache, mit der wir unsere Texte im Rohformat schreiben. Um aus **asciidoc**-Dokumenten die gewünschten Ausgabeformate zu rendern, benötigen wir jetzt noch die entsprechenden **Textprozessoren**.

3.3.1. asciidoctor

asciidoctor unterstützt von Haus aus die Ausgabeformate **HTML5**, **DocBook 5** und **man page**. Außerdem integriert **asciidoctor** ein Standard-Styleheet, durch das die generierten HTML-Dateien direkt in einem ansprechenden Design daherkommen und einen Syntax Highlighter, durch den Codebeispiele im HTML farblich hervorgehoben werden.

Bitte folgt der Anleitung auf der folgenden Seite, um **asciidoctor** über das Terminal zu installieren.



<https://docs.asciidoctor.org/asciidoctor/latest/install/>


3.3.2. asciidoctor-pdf

asciidoctor-pdf ist eine Erweiterung von **asciidoctor** und erlaubt uns, auch PDF-Dateien aus unseren **asciidoc**-Dateien zu rendern. Da wir im Laufe des Tutorials sehen wollen, wie aus den selben Rohdaten verschiedene Ausgabeformate erzeugt werden können, benötigen wir auch diese Erweiterung.


Bitte folgt der Anleitung auf der folgenden Seite, um **asciidoctor-pdf** über das Terminal zu installieren.



<https://github.com/asciidoctor/asciidoctor-pdf/tree/main#getting-started>

Sobald beide Prozessoren installiert sind, können wir über das Terminal jedes beliebige **asciidoc**-Dokument in das gewünschte Zielformat konvertieren. Wir werden das im Laufe des Tutorials auch direkt ausprobieren, siehe dazu auch  [Eine Webseite via asciidoctor erzeugen](#).



Wenn euch die Installation der Prozessoren Probleme bereitet, könnt ihr das Tutorial auch auch ersteinmal nur mittels **ATOM** und den unter  [Der ATOM Editor](#) beschriebenen Packages fortsetzen.

4. Das erste Dokument erstellen, previewen und rendern

Die Ziele dieses Abschnitts sind:

- ☑ das erste **asciidoc**-Dokument in **ATOM** zu erstellen
- ☑ aus demselben Ursprungsdokument eine Webseite und ein PDF zu generieren

Nachdem wir alle notwendigen Tools installiert haben, wollen wir sie natürlich direkt zur Anwendung bringen.

4.1. Das erste asciidoc-Dokument schreiben

Erstellen wir zunächst ein **asciidoc**-Dokument in **ATOM**.

```
== Der Eiffelturm

Der *Eiffelturm* ist ein 324 Meter hoher Turm in Paris, der zum Anlass der
https://de.wikipedia.org/wiki/Weltausstellung\_Paris\_1889[Weltausstellung 1889] von
*Gustave Eiffel* errichtet worden ist.

=== Gustave Eiffel
Gustave Eiffel war ein französischer Ingenieur.
```

1. Erzeuge eine neue Datei in **ATOM**.
2. Kopiere den oben stehenden Text in das Dokument.
3. Speichere das Dokument als **eiffelturm.adoc**.

In **ATOM** sollten wir jetzt folgendes sehen:

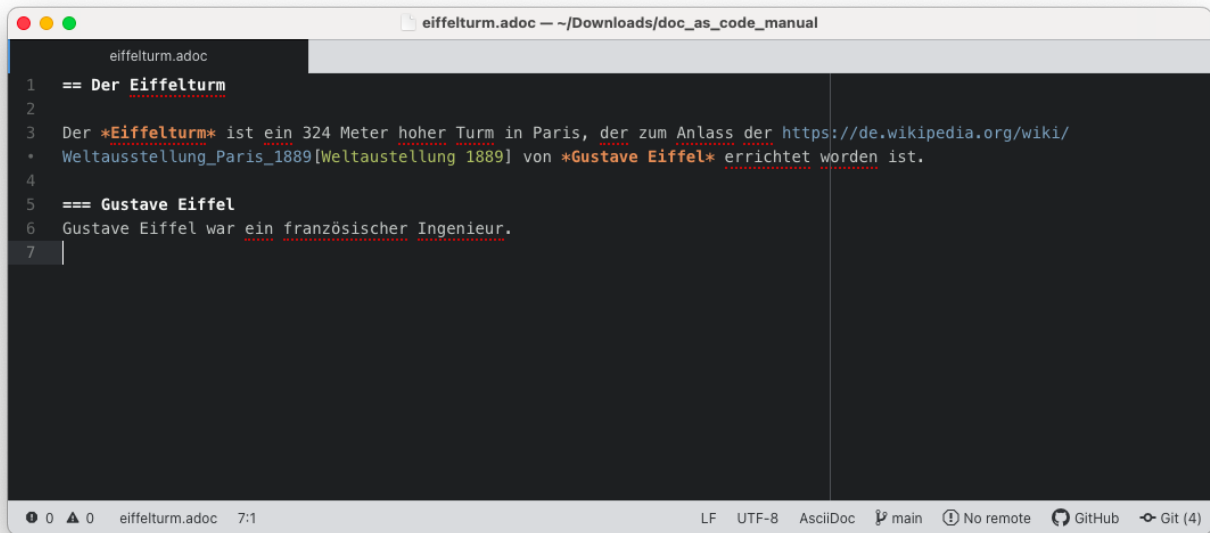


Abb. 6. Die asciidoc-Syntax wird farblich hervorgehoben, sobald das Dokument anhand der Dateierdung als asciidoc zu erkennen ist.

Wie wir sehen können, wird die **asciidoc**-Syntax durch das **language-asciidoc**-Plugin farblich hervorgehoben.

4.2. Die HTML-Preview via asciidoc-preview nutzen

Jetzt möchten wir natürlich sehen, wie unser Text im HTML-Format aussehen würde. Dazu öffnen wir in **ATOM** die HTML-Preview mittels der Tastenkombination `shift + strg + a`, bzw. `shift + cmd + a`.

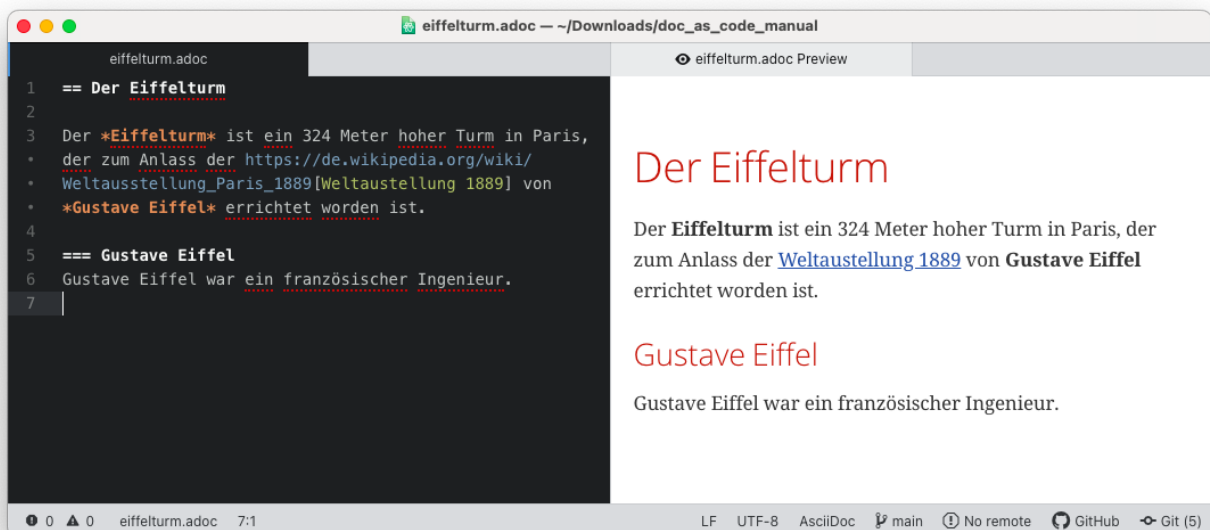


Abb. 7. Links zu sehen: Der Text im asccidoc-Rohformat, rechts die HTML-Preview.

Wenn die Installation von **asciidoc-preview** erfolgreich war, sehen wir, wie unsere Datei als Webseite aussehen würde.



Probiert doch direkt ein paar weitere Gestaltungsmöglichkeiten aus, die ihr unter [Die asciidoc-Syntax](#) kennengelernt habt. Bei jeder Veränderung der asciidoc-Datei links wird die Vorschau sofort aktualisiert.

4.3. Eine Webseite via asciidoctor erzeugen

Natürlich wollen wir unsere Dateien nicht nur als HTML previewen, wir wollen sie auch als HTML-Datei ablegen können. Dazu nutzen wir jetzt den **asciidoctor**-Textprozessor, den wir unter [Die Textprozessoren asciidoctor & asciidoctor-pdf](#) installiert haben.

1. Öffnet ein Terminal.
2. Navigiert zu dem Verzeichnis, in dem die Datei `eiffelturm.adoc` liegt.
3. Führt jetzt folgenden Befehl aus:

```
asciidoctor eiffelturm.adoc
```

asciidoctor erzeugt aus der **asciidoc**-Datei jetzt eine HTML-Seite und legt sie als `eiffelturm.html` im selben Verzeichnis wie die Ursprungsdatei ab.

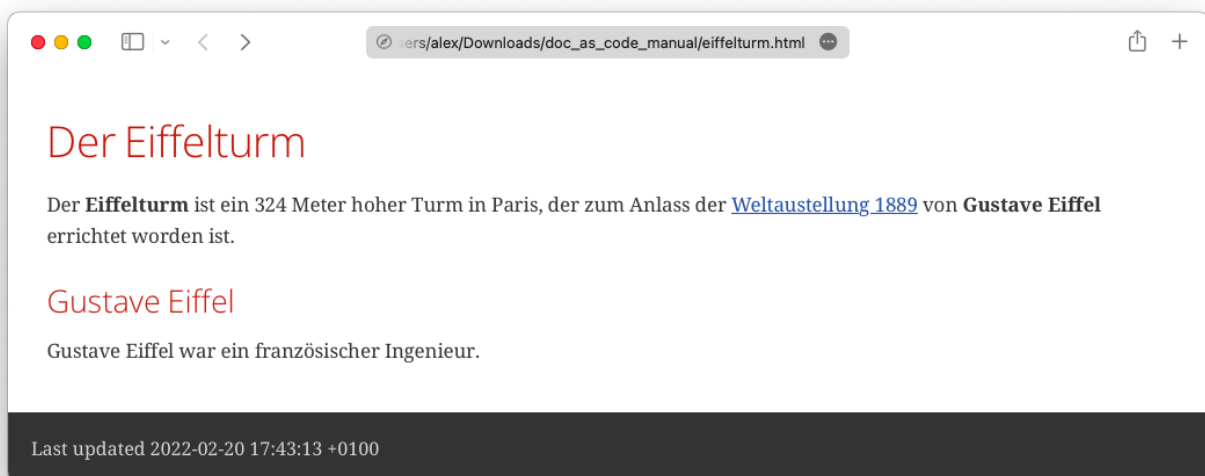


Abb. 8. So sollte die gerenderte HTML-Datei aussehen, wenn wir sie im Browser öffnen.



Wie oben bereits erwähnt, kann die HTML-Datei auch über Rechtsklick auf die **ATOM**-Vorschau und Auswahl von **Save as HTML** ... abgespeichert werden.

Voilà! Eine kleine Webseite, inklusive Formatierung und Link – und das, ohne eine einzige Zeile HTML geschrieben zu haben.

4.4. Ein PDF via asciidoctor-pdf erzeugen

Fangen wir an, das Potenzial von **asciidoc** etwas auszuschöpfen. Nehmen wir an, wir wollen unser

Dokument lieber als PDF in Umlauf bringen, schließlich ist eine Webseite je nach Kontext nicht immer das geeignete Medium für eine Technische Dokumentation.

1. Öffnet dazu erneut ein Terminal.
2. Navigiert wieder zu dem Verzeichnis, in dem die Datei `eiffelturm.adoc` liegt.
3. Führt jetzt folgenden Befehl aus:

```
asciidoctor-pdf eiffelturm.adoc
```

Jetzt generiert `asciidoctor-pdf` ein PDF-Dokument und legt dieses als `eiffelturm.pdf` ab.

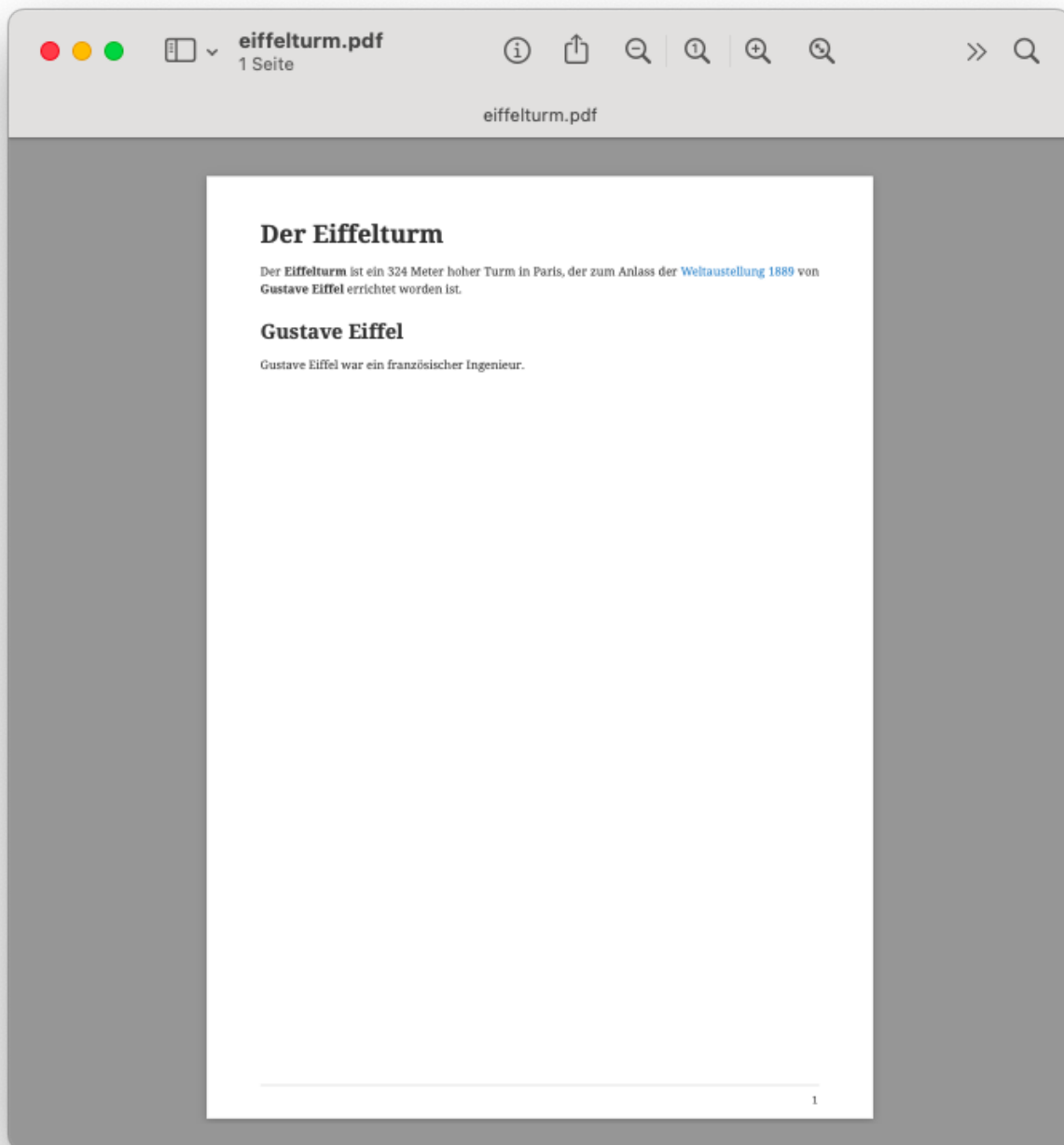


Abb. 9. So sollte das gerenderte PDF aussehen.

Wie ihr sehen könnt, sind auch im PDF die Formatierungen erhalten geblieben – auch der Link ist wunderbar eingebettet und funktioniert, ohne das wir nur eine einzige Anpassung an unserem Ursprungsdokument vornehmen mussten!

👍 **Herzlichen Glückwunsch!**

Wenn alles funktioniert hat, habt ihr gerade in wenigen Schritten eure erste Datei im **asciidoc**-Format geschrieben und daraus eine simple Webseite und ein PDF generiert. Und hoffentlich dabei auch ein Gefühl dafür bekommen, welches Potenzial in dieser Vorgehensweise liegt!

5. Modulares Arbeiten mit asciidoc

Die Ziele dieses Abschnitts sind:

- ☑ ein grundsätzliches Verständnis des modularen Textaufbaus zu bekommen
- ☑ die `include`-Funktion von **asciidoc** und den `leveloffset`-Parameter kennenzulernen

Natürlich wird eine Technische Dokumentation nie aus nur wenigen Sätzen bestehen, die auf eine einzige DIN-A-4-Seite passen. Ganz im Gegenteil, umfangreichere Dokumentationen können sich über hunderte Seiten erstrecken.

Bei größeren Vorhaben sollte man daher modular arbeiten. Das heißt, seine Dokumentation nicht in einer einzigen Datei zu erstellen, sondern in sinnvolle Einheiten aufzuteilen, die einzeln viel einfacher zu warten sind.

Hier kommt eine besondere Stärke von **asciidoc** ins Spiel: Die `include`-Funktion. Mit ihr können wir jedes Text-Dokument in ein **asciidoc**-Dokument einbetten und auf diese Weise aus verschiedenen Textbestandteilen unseren gewünschten Gesamttext zusammenstellen.

5.1. Die include-Funktion nutzen

Ein einfaches Beispiel: Nehmen wir an, wir möchten unserem kleinen Text zum Eiffelturm noch die Adresse und die Öffnungszeiten hinzufügen. Es könnte ja sein, dass wir diese Informationen später einmal an verschiedenen Stellen unseres Textes benötigen – es wäre doch sehr umständlich, dann bei jeder Veränderung der Öffnungszeiten die einzelnen Stellen jeweils einzeln anpassen zu müssen.

Stattdessen erzeugen wir für die Öffnungszeiten ein eigenes Dokument.

1. Kopiere den unten stehenden Text.
2. Speichere ihn als `eiffelturm_adresse.adoc`

```

== Adresse und Öffnungszeiten des Eiffelturms
*Adresse:* +
Champ de Mars +
5 Av. Anatole France +
75007 Paris, Frankreich

*Öffnungszeiten:*
[cols="1,1"]
|===
|Montag
|09:30 - 22:30 Uhr
|Dienstag
|09:30 - 22:30 Uhr
|Mittwoch
|09:30 - 22:30 Uhr
|Donnerstag
|09:30 - 22:30 Uhr
|Freitag
|09:30 - 22:30 Uhr
|Samstag
|09:30 - 22:30 Uhr
|Sonntag
|09:30 - 22:30 Uhr
|===

```

In unserem Projektordner haben wir jetzt zwei Dokumente:

```

└─ "Unser Projekt"
   └─ eiffelturm.adoc
   └─ eiffelturm_adresse.adoc

```

Wir öffnen jetzt erneut das Dokument `eiffelturm.adoc` und fügen unten folgende Zeile hinzu:

```
include::eiffelturm_adresse.adoc[]
```

Wir sagen damit, dass wir an dieser Stelle das Dokument `eiffelturm_adresse.adoc` einbetten wollen. Dabei geben wir den Pfad zum einzubettenden Dokument relational zu der Datei ein, in die der Text inkludiert werden soll. Da `eiffelturm.adoc` und `eiffelturm_adresse.adoc` auf einer Ebene in unserem Projektordner liegen, reicht die Angabe des Dokumentennamens.

Würde `eiffelturm_adresse.adoc` in einem Unterordner namens `partials` liegen, würde der Befehl wie folgt aussehen:

```
include::partials/eiffelturm_adresse.adoc[]
```



Würde das einzubettende Element in einem übergeordneten Ordner liegen, wäre die Pfadangabe wie folgt:

```
include::../eiffelturm_adresse.adoc[]
```

Schauen wir uns jetzt die HTML-Preview von `eiffelturm.adoc` an:

The screenshot shows a code editor with two panes. The left pane displays the source file `eiffelturm.adoc` with the following content:

```
1 == Der Eiffelturm
2
3 Der *Eiffelturm* ist ein 324 Meter hoher Turm in Paris, der zum Anlass der
4 * https://de.wikipedia.org/wiki/Weltausstellung_Paris_1889[Weltausstellung 1889]
5 von *Gustave Eiffel* errichtet worden ist.
6
7 == Gustave Eiffel
8 Gustave Eiffel war ein französischer Ingenieur.
9
10 include::eiffelturm_adresse.adoc[]
```

The right pane shows the HTML preview of the document. It features the title "Der Eiffelturm" in red, followed by a paragraph: "Der **Eiffelturm** ist ein 324 Meter hoher Turm in Paris, der zum Anlass der [Weltausstellung 1889](https://de.wikipedia.org/wiki/Weltausstellung_Paris_1889) von **Gustave Eiffel** errichtet worden ist." Below this is a section titled "Gustave Eiffel" with the text "Gustave Eiffel war ein französischer Ingenieur." Further down is another section titled "Adresse und Öffnungszeiten des Eiffelturms". Under "Adresse:", it lists "Champ de Mars", "5 Av. Anatole France", and "75007 Paris, Frankreich". Under "Öffnungszeiten:", it contains a table with opening hours for Monday through Thursday.

Öffnungszeiten:	
Montag	09:30 - 22:30 Uhr
Dienstag	09:30 - 22:30 Uhr
Mittwoch	09:30 - 22:30 Uhr
Donnerstag	09:30 - 22:30 Uhr

Abb. 10. In der Vorschau gut zu erkennen: Das Dokument mit Adresse und Öffnungszeiten wird in unseren Text zum Eiffelturm eingebettet.

Die `include`-Funktion kann für verschiedene Zwecke genutzt werden:

- wir können je nach Anwendungsfall aus denselben Ursprungsdateien unterschiedliche Zieltexte zusammenstellen
- wir können oft wiederverwendete Informationen an einer Stelle verwalten und pflegen und diese dann an verschiedenen Stellen inkludieren (Stichwort: Glossar)
- wir können Quellcode-Beispiele aus den tatsächlichen Code-Dokumenten einbetten

5.2. Die Überschriften-Hierarchie kontrollieren

Vielleicht ist es euch schon aufgefallen: Sowohl in unserem Dokument `eiffelturm.adoc` als auch `eiffelturm_adresse.adoc` haben wir jeweils eine Überschrift erster Ordnung verwendet. Deshalb sind die Überschriften "Der Eiffelturm" und "Adresse und Öffnungszeiten des Eiffelturms" in unserem zusammengeführten Dokument auf einer Ebene. Sinnvoller wäre aber, wenn die Adresse ein Unterpunkt im Kapitel zum Eiffelturm wäre.

Mit dem `leveloffset`-Parameter können wir beim Inkludieren von Dateien kontrollieren, auf welcher Hierarchieebene sich die beinhalteten Überschriften einfügen. Wir ändern unsere `include`-Funktion daher wie folgt ab.

```
include::eiffelturm_adresse.adoc[leveloffset=+1]
```

Durch die Angabe von `leveloffset=+1` bestimmen wir, dass alle Überschriften des eingebetteten Dokuments genau eine Hierarchieebene nach unten sinken. Jetzt sieht unser HTML-Dokument wie folgt aus:

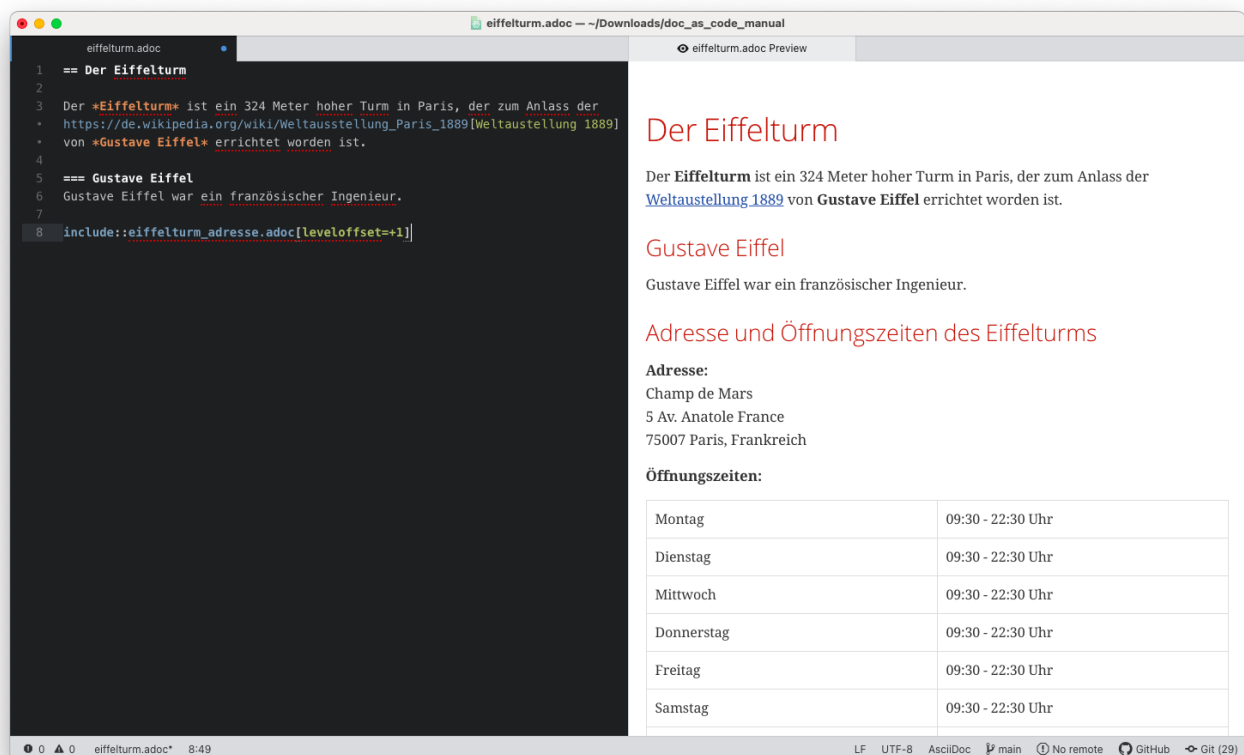


Abb. 11. Durch Verwendung von `leveloffset=+1` werden die Überschriften des eingebetteten Dokuments eine Hierarchieebene abgesenkt.

Der `leveloffset`-Parameter ist ein sehr hilfreiches Werkzeug, wenn man die Gliederung von zusammengeführten Dokumenten kontrollieren und neu strukturieren möchte, ohne dafür die Überschriftenebenen in den einzelnen Dokumenten separat anpassen zu müssen.

6. Ein komplexes Dokument mit einem Kopfdokument steuern

Die Ziele dieses Abschnitts sind:

- ☑ zu lernen, wie man ein komplexeres Dokument über ein Kopfdokument zusammenstellt
- ☑ zu erlernen, wie man ein Inhaltsverzeichnis einbettet
- ☑ globale Eigenschaften kennenzulernen, die über das Kopfdokument festgelegt werden können

Wir haben jetzt gelernt, wie wir mittels **asciidoc** Texte miteinander verknüpfen können. Als nächstes möchten wir nun ein etwas komplexeres Dokument erstellen und dieses mit ein paar besonderen Eigenschaften wie einem Inhaltsverzeichnis ausschmücken.

Damit wir genug Material zum Zusammenfügen haben, erstellen wir zunächst ein weiteres **asciidoc**-Dokument und speichern es als `turm_von_pisa.adoc` ab.

```
== Der schiefe Turm von Pisa
Der schiefe Turm von Pisa ist das Wahrzeichen der Stadt Pisa in Italien und wegen
seiner starken Neigung weltberühmt. Die Grundsteinlegung des Turms erfolgte am
9.August 1173.

CAUTION: Der schiefe Turm von Pisa hat eine Gesamtneigung von ca. 4 Grad.
```

In unserem Projektordner sollten wir jetzt drei **asciidoc**-Dokumente haben:

```
└─ "Mein Projekt"
   └─ eiffelturm.adoc
      └─ eiffelturm_adresse.adoc
         └─ turm_von_pisa.adoc
```

6.1. Ein Kopfdokument erstellen

Diese Dokumente möchten wir jetzt zu einem zusammenfügen. Wir erstellen dazu jetzt noch ein weiteres **asciidoc**-Dokument, das den "**Kopf**" unseres Textes darstellen wird und mittels dem wir die Gesamtstruktur unseres Textes festlegen werden. Ein Kopfdokument bietet darüber hinaus den Vorteil, dass wir in ihm einige globale Attribute unseres Dokuments zentral bestimmen können.

1. Kopiert folgenden Text in eine neue Datei.
2. Speichert die Datei unter `bauwerke_europas.adoc` ab.

```

= Berühmte Bauwerke Europas
Max Mustermann <max.mustermann@hiq.de>
:description: Eine Übersicht der berühmtesten Bauwerke Europas
//:title-page:
:toc:
:toc-title: Inhaltsverzeichnis
:sectnums:
:icons: font

include::eiffelturm.adoc[]

<<<

include::turm_von_pisa.adoc[]

```

6.1.1. Globale Attribute im Dokumentenkopf festlegen

Schauen wir uns die einzelnen Elemente des Kopfdokuments einmal näher an:

```

= Berühmte Bauwerke Europas ①
Max Mustermann <max.mustermann@hiq.de> ②
:description: Eine Übersicht der berühmtesten Bauwerke Europas ③
//:title-page: ④
:toc: ⑤
:toc-title: Inhaltsverzeichnis ⑥
:sectnums: ⑦
:icons: font ⑧
⑨
include::eiffelturm.adoc[] ⑩

<<< ⑪

include::turm_von_pisa.adoc[] ⑫

```

- ① Das ist der Titel des Dokuments, zu erkennen an dem einfachen `=`. Ein Dokument darf immer nur *einen* Titel haben.
- ② Unmittelbar unter dem Dokumententitel (keine Leerzeile) folgen die Attribute, diese sind grundsätzlich optional. In die erste Zeile setzen wir den Autor des Dokuments und ggfs. eine E-Mail-Adresse ein.
- ③ Unter dem Attribut `:description:` haben wir die Möglichkeit, eine Beschreibung unserer Datei anzugeben.
- ④ Hier auskommentiert und deshalb nicht wirksam: Bei Angabe des Attributs `:title-page:` wird das PDF mit einer Titelseite ausgestattet.
- ⑤ Mit `:toc:` geben wir an, dass unserem Dokument ein Inhaltsverzeichnis ("Table of Contents") vorangestellt werden soll.
- ⑥ Unter `:toc-title:` bestimmen wir, dass das Inhaltsverzeichnis als "Inhaltsverzeichnis" betitelt

werden soll. Ansonsten wird standardmäßig "Table of Contents" gesetzt.

- ⑦ `:sectnums:` bestimmt, dass unser Inhaltsverzeichnis nummeriert werden soll.
- ⑧ `:icons: font` ermöglicht u.a., dass unsere Hinweisboxen im Export mit einem Icon versehen werden.
- ⑨ Die erste Leerzeile schließt den Kopf des Dokuments ab.
- ⑩ Hier inkludieren wir die erste Datei `eiffelturm.adoc`.
- ⑪ Die Zeichenkette `<<<` erzwingt einen Seitenumbruch im PDF-Format.
- ⑫ Hier inkludieren wir unseren Text zum Turm von Pisa.



Im Abschnitt [🔗 Die include-Funktion nutzen](#) haben wir ja bereits dafür gesorgt, dass das Dokument `eiffelturm_adresse.adoc` vom Dokument `eiffelturm.adoc` eingebettet wird, deswegen müssen wir die Datei hier kein weiteres mal inkludieren.

Nebenebei: Die hier aufgeführten Eigenschaften sind nur einige wenige Attribute, die wir in **asciidoc** im Dokumentenkopf bestimmen können.

6.2. Das Ergebnis

Schauen wir mal, wie sich unsere Einstellungen auf das Endergebnis auswirken. Wir nutzen erneut `asccidoctor` und `asccidoctor-pdf`, um aus `bauwerke_europas.adoc` eine Webseite, bzw. ein PDF zu rendern.

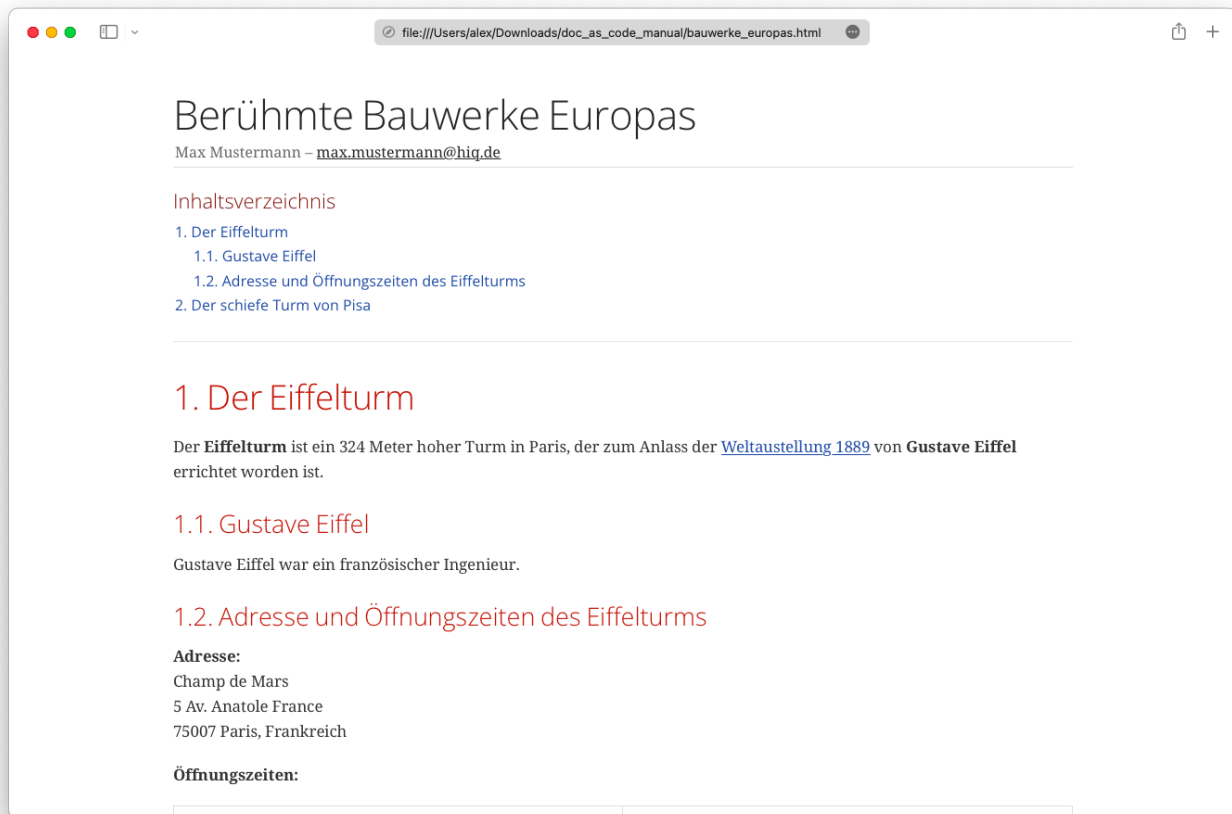


Abb. 12. 👍 Die fertige Webseite mit interaktivem Inhaltsverzeichnis und unseren Autorinformationen

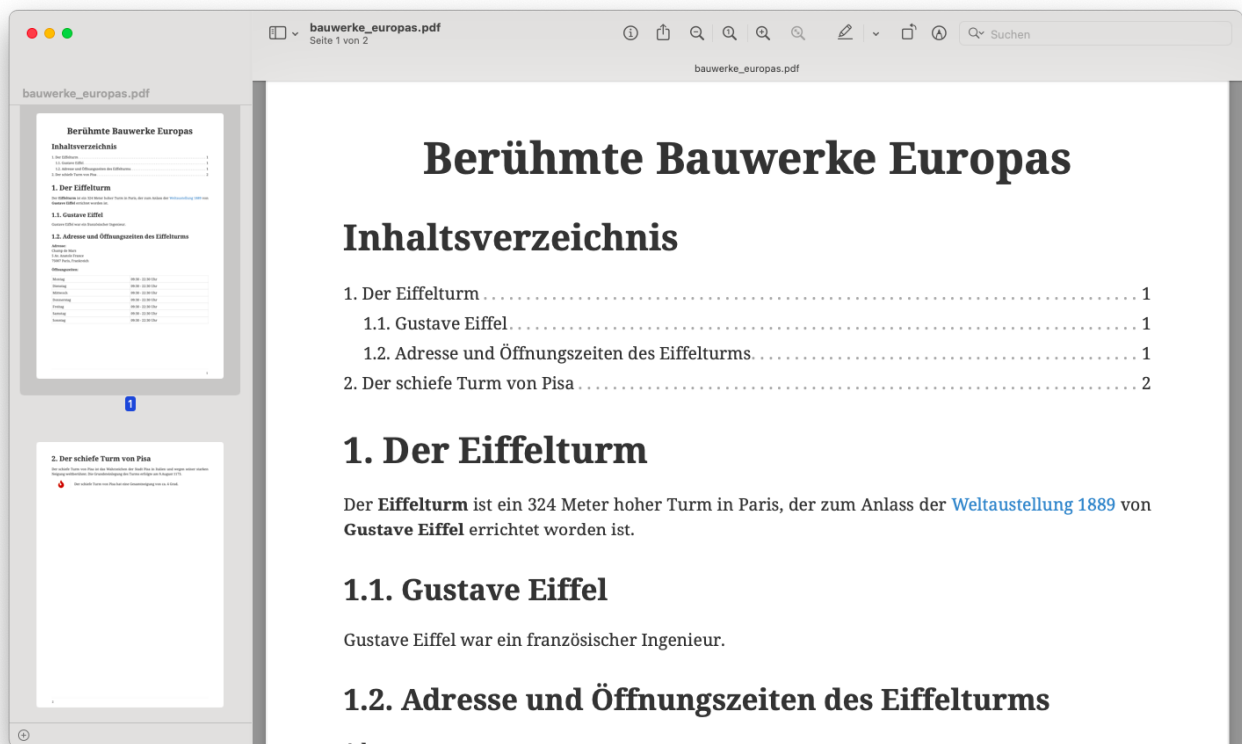


Abb. 13. 👍 Das gerenderete PDF, ebenfalls mit interaktivem Inhaltsverzeichnis und dem gewünschten Seitenumbruch.






7. Ausblick und Vertiefung

Hiermit habt ihr das kleine Tutorial zu **Doc as Code** erfolgreich beendet! 

Ich hoffe, dass ihr einen guten Einblick in die Arbeit mit **asciidoc** gewinnen konntet, und das Tutorial die Grundzüge des **Doc as Code**-Ansatzes vermittelt konnte.

Bei Fragen könnt ihr euch gerne melden: alex.kurzke@hiq.de

Natürlich konnten wir vieles nur oberflächlich anschauen, wer aber Gefallen an der Arbeit mit **asciidoc** gefunden hat, dem seien daher als Impuls noch folgende interessante Themen und Links mitgegeben:

- individuelles Styling des HTML mittels css-Styleheets
 -  <https://github.com/darshandsoni/asciidoctor-skins>
- individuelles Styling des PDF-Outputs über Themes
 -  <https://github.com/asciidoctor/asciidoctor-pdf/blob/v1.6.0/docs/theming-guide.adoc>
- Einbinden von Diagrammen und Grafiken, bspw. auf PlantUML-Basis
 -  <https://docs.asciidoctor.org/diagram-extension/latest/>
- Icons aus dem Font-Awesome-Set nutzen, um unsere Dokumentation freundlicher und interessanter zu gestalten
 -  <https://docs.asciidoctor.org/asciidoc/latest/macros/icons/>
- Die im GIT verwaltete Dokumenten an einen Static Site Generator anbinden, aus dem die Dokumentationswebseite automatisch generiert werden
 -  <https://antora.org>