PDFS AUS HTML-TEMPLATES ERZEUGEN

Chrome als PDF-Drucker

Mit Puppeteer Sharp und Paged.js lassen sich druckreife Dokumente erzeugen.

Aus einem Nachrichtenartikel oder einem Blog-Artikel mal eben ein PDF erzeugen: Das funktioniert in Chromium-basierten Browsern, ohne einen zusätzlichen PDF-Drucker im Rechner einrichten zu müssen. Das kann man sich zum Beispiel mit der .NET-Bibliothek Puppeteer Sharp zunutze machen, um aus HTML-Templates PDF-Dokumente zu erzeugen [1].

Dass Puppeteer Sharp sich als einfach zu verwendendes Tool für Web-UI-Tests mittels Chromium-basierten Browsern verwenden lässt (insbesondere mit Google Chrome und Microsoft Edge), hat die vorangegangene Ausgabe der dotnetpro gezeigt. Der Artikel unter [2] erläuterte das Zusammenspiel der Bibliothek mit dem Browser und es lohnt sich, dies zum besseren Verständnis zu lesen, doch es ist kein Muss. Kurz: Puppeteer Sharp ist eine Portierung des Node.js-Programms Puppeteer, mit dessen Hilfe sich Chromium-basierte Browser per API bequem aus .NET heraus steuern lassen.

Mobile first – auch für Druckerzeugnisse

Das Schlagwort „mobile first" ist bei der Webseiten-Entwicklung mittlerweile Standard, und auch das papierlose Büro rückt langsam anscheinend doch näher. Spätestens bei Auftragsbestätigungen oder Rechnungen muss es in den meisten Fällen zumindest ein PDF sein, das den Kunden zugeschickt wird. Wieso also nicht gleich die eigene Firmen-Website und die dort umgesetzten Design-Entscheidungen (Schriftarten und -größen, Farbschema) als Vorlage verwenden?

Puppeteer Sharp bietet für das Erzeugen von PDF-Dateien aus HTML-Seiten ein einfaches API. Im Vergleich zu anderen Template-Lösungen auf Basis von Word kann der (UI-)Entwickler hier mit den gewohnten Werkzeugen HTML und CSS arbeiten. Die Einarbeitung in eine eigene Template-Sprache und die Kosten einer Lizenz für MS Office oder eine Drittanbieter-Komponente entfallen.

Ein erstes PDF

Listing 1 zeigt das vollständige Programm für die Ausgabe der dotnetpro-Homepage als PDF. Der Code startet zuerst eine Chromium-Instanz, erzeugt in ihr eine neue Karteikarte und öffnet die Homepage der dotnetpro, bevor die Seite als PDF gespeichert wird. Das Ergebnis ist das gleiche wie bei der Auswahl der Option *Als PDF speichern* im *Drucken …*-Menü von Chromium. Das Drucken funktioniert mit Puppeteer allerdings nur im Headless-Modus. Setzt man *Headless = false*, schlägt das Drucken fehl. Die Optionen von Chromes Drucken-Dialog, der in Bild 1 zu sehen ist, lassen sich per *PdfOptions*-Objekt auch via Puppeteer übergeben.

Einfache Kopf- und Fußzeilen

Die Optionen des Druckmenüs lassen sich in einem *PdfOptions*-Objekt als zweiter Parameter an *PdfAsync()* übergeben. Listing 2 zeigt dafür ein Beispiel. Hier wird das Papierformat auf *A4* gesetzt; die Seitenränder werden für den Druck zuerst festgelegt, bevor Kopf- und Fußzeile angezeigt und mit Inhalten gefüllt werden.

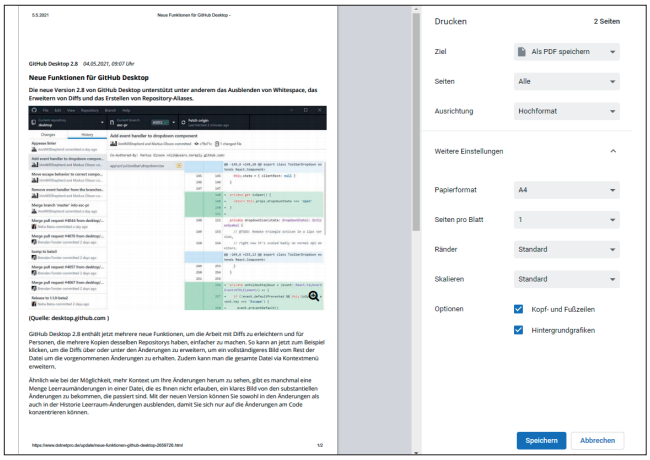
Der Renderer kennt die folgenden Klassen und füllt das innere HTML mit den entsprechenden Werten:

- *date*: das Druckdatum (ohne Uhrzeit),
- *title*: der Titel der Website,

Listing 1: Ein „Hello World"-PDF mit Puppeteer Sharp

```
using PuppeteerSharp;

var browser = await Puppeteer.LaunchAsync(
    new LaunchOptions
{
  Headless = true,
  ExecutablePath = @"C:\Program Files (x86)\"
    + @"Microsoft\Edge\Application\msedge.exe"
});
var page = await browser.NewPageAsync();
await page.GoToAsync("https://www.dotnetpro.de");
await page.PdfAsync(@"c:\temp\dotnetpro.pdf");
```



Der Drucken-Dialog von Chrome lässt sich mit einem PdfOptions-Objekt konfigurieren (Bild 1)

Listing 2: Die Seite mittels PdfOptions-Parameter layouten

```
PdfOptions pdfOptions = new() {
  Format = PuppeteerSharp.Media.PaperFormat.A4,,
    MarginOptions = new() { Top = "20mm",
    Bottom = "20mm", Left="14mm", Right="14mm" },
    DisplayHeaderFooter = true,
    HeaderTemplate = "<div style=\"font-size:10px;"
      + "margin: 0 14mm;\"<span class='title'/></"
      + "div>", FooterTemplate = "<div style=\""
      + "font-size:10px; margin: 0 14mm;\">Seite "
      + "<span class=\"pageNumber\"></span> von <span"
      + " class=\"totalPages\"></span></div>",
    PrintBackground = true,
};

await page.PdfAsync(@"c:\temp\dotnetpro.pdf",
  pdfOptions);
```

- *url*: die Webadresse der Seite,
- *pageNumber*: die aktuelle Seitenzahl,
- *totalPages*: die Gesamtzahl der Seiten.

Das Formatieren von Kopf- und Fußzeile muss (leider) „inline", also direkt im jeweiligen Element erfolgen, ein Zugriff auf CSS-Dateien ist nicht möglich.

Möchte man nur die Kopf- oder die Fußzeile anzeigen, ist der jeweils andere Wert mit einem leeren ** zu füllen, damit nicht der Standardwert angezeigt wird. *PrintBackground = true* sorgt dafür, dass auch via CSS mit *background-image* gesetzte Bilder gedruckt werden.

Vorlagen mit Puppeteer-Bordmitteln

Jetzt wird es Zeit für eine erste echte Vorlage. Folgendes Beispiel zeigt die Anschrift als Teil einer Rechnungsvorlage. Die zu ersetzenden Werte sind mit *id*-Tags versehen. Die Beispielinhalte können somit realistisch gewählt sein, was den Entwurf und die fachliche Abstimmung mit Endanwendern erheblich erleichtert:

```
<div class="Mahnung" id="mahnung">Mahnung</div>
<div class="anschrift">
  <p id="name">Max Mustermann</p>
  <p id="strasseHausnummer">Hauptstraße 1</p>
  <p id="plzOrt">12345 Musterstadt</p>
</div>
```

Der Code in Listing 3 zeigt das Laden der Vorlage und das Füllen mit echten Werten in den beiden Hilfsmethoden *ReplaceMarker()* und *RemoveMarker()*. Die Methoden manipulieren das DOM, um die gewünschten Werte zu schreiben – nicht ganz elegant, dafür lassen sich Referenzen auf verlinkte CSS-Dateien und eingebundene Schriftarten aber korrekt auflösen. Würde man stattdessen das HTML per *page.SetContent("<html>…")* injizieren, wäre die Auflösung relativer Referenzen deutlich aufwendiger.

Schriftarten bindet Chromium übrigens auch automatisch in das PDF ein. Was es dabei zu beachten gilt, erläutert der Kasten Große Dateien vermeiden und Schriftarten richtig referenzieren.

Mit diesem Rüstzeug ausgestattet lassen sich einfache PDF-Seiten erstellen. Was aber, wenn es doch ein längerer Bericht sein soll und die Anforderungen etwas anspruchsvoller sind? Wenn etwa unterschiedliche Kopf- und Fußzeilen für gerade und ungerade Seiten oder eine bessere Steuerung von Seitenwechseln, Fußnoten oder Tabellen erzielt werden sollen? Hier stößt Puppeteer (Sharp) an seine Grenzen. ▶

Listing 3: Die Vorlage laden und Marker ersetzen

```
var templatePath = Path.Combine(AppDomain.
  CurrentDomain.BaseDirectory,
  "RechnungTemplate.html");
// sicherstellen, dass die Seite vollständig geladen
// wurde
WaitUntilNavigation[] arg = { WaitUntilNavigation.
  Networkidle0, WaitUntilNavigation.Load };
await page.GoToAsync(templatePath, waitUntil: arg);
await ReplaceMarker("name", "Michael Meyer");
await ReplaceMarker("strasseHausnummer",
  "Maximiliansstraße 1");
await ReplaceMarker("plzOrt", "80434 München");

await RemoveMarker("mahnung");

await page.PdfAsync(@"c:\temp\Rechnung.pdf",
  pdfOptions);

async Task ReplaceMarker(string markerId,
    string content)
{
  var jSQuery = @$"document.getElementById(
    ""{markerId}"").innerHTML = ""{content}""";
  await page.EvaluateExpressionAsync(jSQuery);
}

async Task RemoveMarker(string markerId)
{
  var jSQuery = @$"document.getElementById(
    ""{markerId}"").remove()";
  await page.EvaluateExpressionAsync(jSQuery);
}
```

Listing 4: Eine temporäre HTML-Datei mit ersetzten Markern erzeugen

```
void ReplaceMarkers(string templatePath, string
    tempFile, IEnumerable<Replacement> replacements)
{
  var htmlDoc = new HtmlDocument();
  htmlDoc.Load(templatePath);

  foreach (var replacement in replacements)
  {
    var id = replacement.Id;
    var node = htmlDoc.DocumentNode.
      SelectSingleNode($"//*[@id='{id}']");
    if (node == null)
      continue; // alterantiv: ErrorHandling

    if (replacement.RemoveElement)
      node.Remove();
    else
      node.InnerHtml = replacement.InsertValue;
  }

  htmlDoc.Save(tempFile);
}
```

Komplexere Elemente einfügen

Die direkte Manipulation des DOM mit JavaScript funktioniert. Aber spätestens, wenn der einzufügende Inhalt Anführungszeichen oder eventuell auch dynamisch lange Tabellen enthält, wird diese Variante unnötig komplex: Erst müssen bestimmte Zeichen in C# maskiert und sie dann für JavaScript entsprechend umgewandelt werden. Weil im nächsten Schritt das vollständige Dokument mit allen Inhalten benötigt wird, ist an dieser Stelle der Einsatz des Html Agility Pack [3] und das Erzeugen einer temporären HTML-Datei sinnvoll; das erspart die Arbeit mit JavaScript.

Dazu wird zuerst eine Datenstruktur erzeugt, die alle nötigen Inhalte enthält, die es zu ersetzen gilt (Listing 4). Die Methode *ReplaceMarkers()* nutzt das Html Agility Pack, um das Template zu laden. Im *foreach*-Loop wird per XPath-Ausdruck das HTML-Element mit der entsprechenden ID ausgewählt und dieses Element entweder gelöscht oder der Wert für *InnerHtml* durch den Wert des *replacement* ersetzt. Anschließend wird die temporäre HTML-Datei gespeichert; entweder im selben Verzeichnis wie die Vorlage, oder es muss sichergestellt werden, dass die in der temporären Datei referenzierten Dateien korrekt verlinkt sind. Wie ein solches *Replacement* aufgebaut ist, zeigt der folgende Code:

```
public struct Replacement
{
  public string Id;
  public string? InsertValue;
  public bool RemoveElement;
}
```

Seitenlayout mit Paged.js

Mit der freien und quelloffenen JavaScript-Bibliothek Paged.js [4] kann die Vorlage zum echten Druckerzeugnis werden. Paged.js dient nach eigenen Aussagen dazu, ein Seitenlayout im Browser zu erzeugen, um aus HTML-Inhalten und CSS ein PDF-Dokument zu erzeugen. Dazu genügt es, das HTML-Template um die folgenden Zeilen zu ergänzen. Das Ergebnis ist in Bild 2 zu sehen:

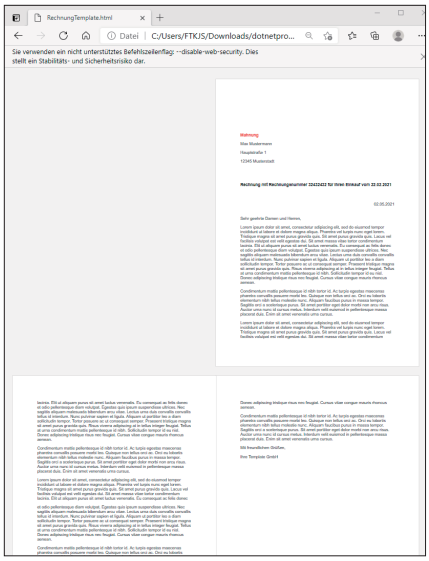
```
<script src="paged.polyfill.js"></script>
<link href="interface.css" rel="stylesheet"
  type="text/css">
```

Paged.js erzeugt automatisch eine Seitenansicht und fängt – wie bei Büchern gewohnt – mit der ersten (also der ungeraden) Seite auf der rechten Seite an. Damit das Tool auch lokal funktioniert, sind für die entsprechende Browser-Instanz einige Sicherheitsfunktionen zu deaktivieren, siehe dazu der Kasten Lokale JavaScript-Dateien im Browser verwenden.

Mit der Seitenansicht liefert Paged.js gleich auch eine Druckvorschau (Bild 2).

Das Tool besteht aus drei Modulen, die folgende Aufgaben haben:

- Der Chunker zerlegt die (eine, endlose) HTML-Seite in einzelne Papier-Seiten.
- Der Polisher übersetzt die Paged.js-eigenen CSS-Definitionen in für den Browser verständliches CSS.
- Der Previewer ruft Chunker und Polisher auf und erzeugt die Ansicht im Browser. Zusätzlich ergänzt er die Inhalte noch um weitere Referenzen, um beispielsweise die linke und rechte Seiten zu unterscheiden.



Aus HTML und CSS erzeugt Paged.js ein PDF-Dokument (Bild 2)

Dieser Ablauf erklärt auch, warum ein Ersetzen von Dummy-Texten via Puppeteer nur mit deutlich größerem Aufwand funktioniert und das Html Agility Pack sinnvoll ist: Der Chunker ist bereits gelaufen, wenn Puppeteer das DOM manipulieren kann. Werden jetzt der Text und insbesondere dessen Länge verändert, würden Seitenumbrüche nicht mehr an den richtigen Stellen sitzen.

Schlimmer noch: Aufgrund der Art, wie Paged.js die Texte formatiert, kann es dann dazu kommen, dass Text einfach nicht angezeigt wird, weil er sich außerhalb des sichtbaren (gedruckten) Bereichs befindet.

Paged.js das Seitenformat steuern lassen

Scharfe Augen haben es in Bild 2 vielleicht schon gesehen: Das Seitenformat ist nicht DIN A4, sondern das US-Letter-Format. Das ist leicht in CSS mit der folgenden Zeile anzupassen:

```
@page {size = A4;}
```

Die Steuerung des Layouts, der Seitenränder sowie von Kopf- und Fußzeilen sollte Puppeteer Sharp besser Paged.js überlassen, damit die Voransicht und der Chunker auch mit dem gewünschten Format arbeiten. Damit dieser Parameter auch beim Erzeugen des PDF berücksichtigt wird, ist *PreferCSSPageSize = true* zu setzen:

```
PdfOptions pdfOptions = new() {
  PreferCSSPageSize = true,
  PrintBackground = true,
  DisplayHeaderFooter = false};
```

Seitenränder flexibel gestalten

Seitenränder lassen sich ebenfalls über die *@page*-Regel in Verbund mit dem CSS-Attribut *margin* steuern. Sofern nicht alle Seiten gleiche Seitenränder haben sollen, kann der folgende Code den Abstand für ein Buch mit symmetrischen Abständen für gegenüberliegende Seiten festlegen:

```
@page:left {
  margin-left: 25mm;
  margin-right: 10mm;
}
@page:right {
  margin-left: 10mm;
  margin-right: 25mm;
}
```

Analog können *@page:first* die erste Seite und *@page:nth(n+2)* alle folgenden Seiten mit unterschiedlichen Seitenrändern definieren.

Umbrüche steuern

Umbrüche können durch die CSS-Eigenschaften *break-before* und *break-after* erzwungen werden. Als Werte sind jeweils *page*, *left*, *right* und *avoid* interessant. Während *page* immer zu einem einfachen Seitenumbruch führt, sorgen *left* und *right* dafür, dass der Text auf einer linken oder rechten Seite startet. Etwaig notwendige leere Seiten erzeugt Paged.js automatisch.

avoid versucht, den Absatz mit dem vorherigen zusammenzuhalten, funktioniert jedoch nur eingeschränkt [5] und sollte entsprechend intensiv vor dem Einsatz getestet werden. Um einen Absatz oder Bereich beim Seitenumbruch zusammenhalten, hilft die Angabe von *page-break-inside: avoid*.

Dynamische Inhalte

Seine wahre Stärke spielt Paged.js bei dynamischen Inhalten aus, und es sei hier auch auf die gute Dokumentation auf der Homepage des Projekts verwiesen. Die folgenden Beispiele können nur einen ersten Eindruck der Möglichkeiten vermitteln. Ein typischer Einsatzzweck für dynamische Inhalte sind Seitenzahlen oder zum Beispiel sich wiederholende Titel in der Kopfzeile. In Paged.js ist der Seitenrand dafür in 16 Boxen unterteilt (Bild 3), die per Selektor angesprochen werden können.

Folgendes Listing zeigt ein Beispiel für eine Seitennummerierung. Da es sich hierbei um Inhalte und nicht um deren Darstellung handelt, empfiehlt der Autor, diese nicht in der CSS-Datei, sondern direkt in der HTML-Datei abzulegen:

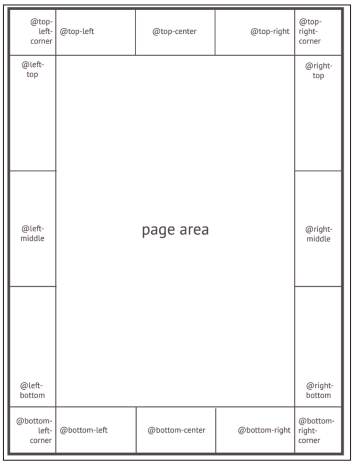
```
<style>
  @page {
    @bottom-right {
      content: "Seite " counter(page) " von "
        counter(pages);
    }
  }
</style>
```

16 Boxen an den Seitenrändern lassen sich per Selektor durch Paged.js ansprechen [6] (Bild 3). Um dynamische Inhalte in einer der 16 Boxen abzulegen, sollten diese Inhalte durch Ersetzen des inneren HTML eines **-Elements in die Vorlage eingebracht werden (siehe Listing 3). Anschließend lassen sie sich per *string-set* in der gewünschten Box anzeigen:

```
<head><style>
  @page {
    @top-center {
      content: string(sonderaktion)
    }
  }

  #sonderaktion {
    string-set:
      sonderaktion content(text)
  }

  .collapsed {
    visibility: collapse;
  }
</style></head>
<body>
<span id="sonderaktion"
  class="collapsed">Nur im April: 10%
    Rabatt!
</span>
```



Paged.js teilt eine Druckseite in 16 Bereiche ein, die sich über Selektoren ansprechen lassen (Bild 3)

Große Dateien vermeiden und Schriftarten richtig referenzieren

Das Projekt war abgeschlossen, der Kunde davon überzeugt, dass verlorene Flexibilität durch die Umstellung von Word auf ein generiertes PDF durch eingesparte Zeit aufgewogen wird. Ist nun alles gut?

Fast, denn die PDF-Dateien waren auf einmal 2 Megabyte groß statt 200 KByte – trotz gleichen Inhalts. Schuld daran waren die verwendeten Schriftarten und wie der Chromium-PDF-Druckertreiber damit umgeht. Verwendet man eigene Schriftarten und enthalten diese keine Definition für Fett oder Kursiv, erzeugt der Treiber diese selbst – leider für jede Instanz einer nicht verfügbaren Stil-Kombination separat. Dabei spielt es auch keine Rolle, ob man *<b>* in HTML oder *font-weight:bold* in CSS verwendet.

Anstatt *<b>*- und *<i>*-Tags zu verwenden, können Sie alle Schriftartenstile in einer Schriftarten-Datei bereitstellen; oder Sie benennen eine Schriftart explizit per *font-familiy* und binden sie per *font-face* in in CSS ein, also zum Beispiel *.bold { font-family: MeineSchriftArtBold; } .boldItalic .{ font-family: MeineSchriftArt-BoldItalic; }*.

Auch das Anzeigen der Überschrift des aktuellen Kapitels der jeweiligen Seite in der Kopfzeile ist möglich. Aus Platzgründen sei hier auf die Anleitung verwiesen. Zusätzlich ist über Hooks der Eingriff in den Rendering-Prozess von Paged.js an verschiedenen Stellen möglich, um gewünschte Funktionen zu ergänzen.

Fazit

Puppeteer Sharp, Html Agility Pack und Paged.js bilden ein starkes Gespann, um professionelle PDF-Dokumente zu erzeugen. Das Einbinden von Bildern hat der Artikel zwar nicht gezeigt, es ist aber durch Einfügen von Image-Tags und deren Layout mit CSS ebenfalls möglich. Die Bildbearbeitung kann dann beispielsweise mit ImageSharp erfolgen [7].

Lokale JavaScript-Dateien im Browser verwenden

Aus Sicherheitsgründen ist der Zugriff auf lokale Dateien in Chromium deaktiviert. Gleiches gilt für das Ausführen von JavaScript-Dateien, die mittels *file://*-Protokoll eingebunden sind. Das heißt, man muss entweder die HTML-Dateien erst auf einen Webserver schieben, um sie von dort per *http(s)://* abzurufen, oder man nutzt das Startup-Flag *--disable-web-security*.

In Puppeteer Sharp lässt sich das mit der folgenden Ergänzung bei den Launch-Optionen bewerkstelligen: *Args = new[] { "--disable-web-security" }*

Beim Entwickeln der Vorlage können Sie den Browser mit folgendem Befehl starten, um das JavaScript von Paged.js zu nutzen:

```
"C:\Program Files (x86)\Microsoft\Edge\Application\
    msedge.exe"  --disable-web-security
  --allow-access-from-files
  --user-data-dir="c:/temp"
  file:///C:/Pfad/Zum/Template.html
```

Aber Vorsicht: Diese Funktionalität ist normalerweise nicht ohne Grund deaktiviert. Mit diesen Browser-Instanzen sollten Sie nur die Templates aufrufen und keine externen Webseiten.

Was noch fehlt, sind eine Silbentrennung für die deutsche Sprache sowie die Möglichkeit, durch punktuelles Verringern von Zeichen- oder Zeilenabständen Texte in den dafür vorgesehenen Platz einzupassen. Bei der Silbentrennung besteht die Chance, dass diese Funktion in Chromium noch dieses Jahr ergänzt wird; für Englisch ist dies bereits möglich. Alternativ wäre es möglich die Texte in .NET zu parsen und um weiche Trennzeichen (in HTML) zu ergänzen. Der Autor konnte jedoch keine Bibliothek finden, die dies bietet.

Die hier vorgestellte Lösung ist in sehr ähnlicher Form trotz Einschränkungen im produktivem Einsatz und hat einen bestehenden Prozess abgelöst, der zuerst ein Word-Dokument erstellt hat, das anschließend manuell als PDF gedruckt wurde. Die Nutzer ließen sich davon überzeugen, dass die verlorene Flexibilität und die noch bestehenden Einschränkungen durch die Zeitersparnis mehr als aufgewogen werden.

Der dritte und abschließende Teil dieser Artikelreihe zu Puppeteer Sharp wird das Web-Crawling mit der Bibliothek demonstrieren. ■

[1] Puppeteer Sharp, www.puppeteersharp.com

[2] Jan Hendrik Schreier, Mit Googles Werkzeugkiste, Web-UI- und End-to-End-Tests, dotnetpro 7/2021, Seite 71 ff., www.dotnetpro.de/A2107PuppeteerSharp

[3] Html Agility Pack (HAP), https://html-agility-pack.net

[4] Paged.js, https://pagedjs.org

[5] Paged Media, [MIX] Fix `break-avoid: after`, www.dotnetpro.de/SL2108PuppeteerSharp1

[6] Paged.js, Generated Content in Margin Boxes, www.dotnetpro.de/SL2108PuppeteerSharp2

[7] ImageSharp, www.dotnetpro.de/SL2108PuppeteerSharp3

Jan Hendrik Schreier
ist Wirtschaftsinformatiker und lebt im Münchner Osten. Beruflich entwickelt er Line-of-Business-Anwendungen (WPF, Full-Stack). Sein Fokus liegt auf Datenstrukturen und fachlichem Prozessdesign.
mail@janschreier.de

dnpCode A2108PuppeteerSharp