Text in PDF: Unicode

This post is Part 3 of a five-part series on understanding the representation of text in PDF. In this article, I explain how Unicode characters are represented in PDF content and how that relates to the embedding of *font subsets*. Take a look at Part 1: *Text in PDF: Introduction*, for a synopsis of (and links to) all the parts as well as a general introduction to the series.

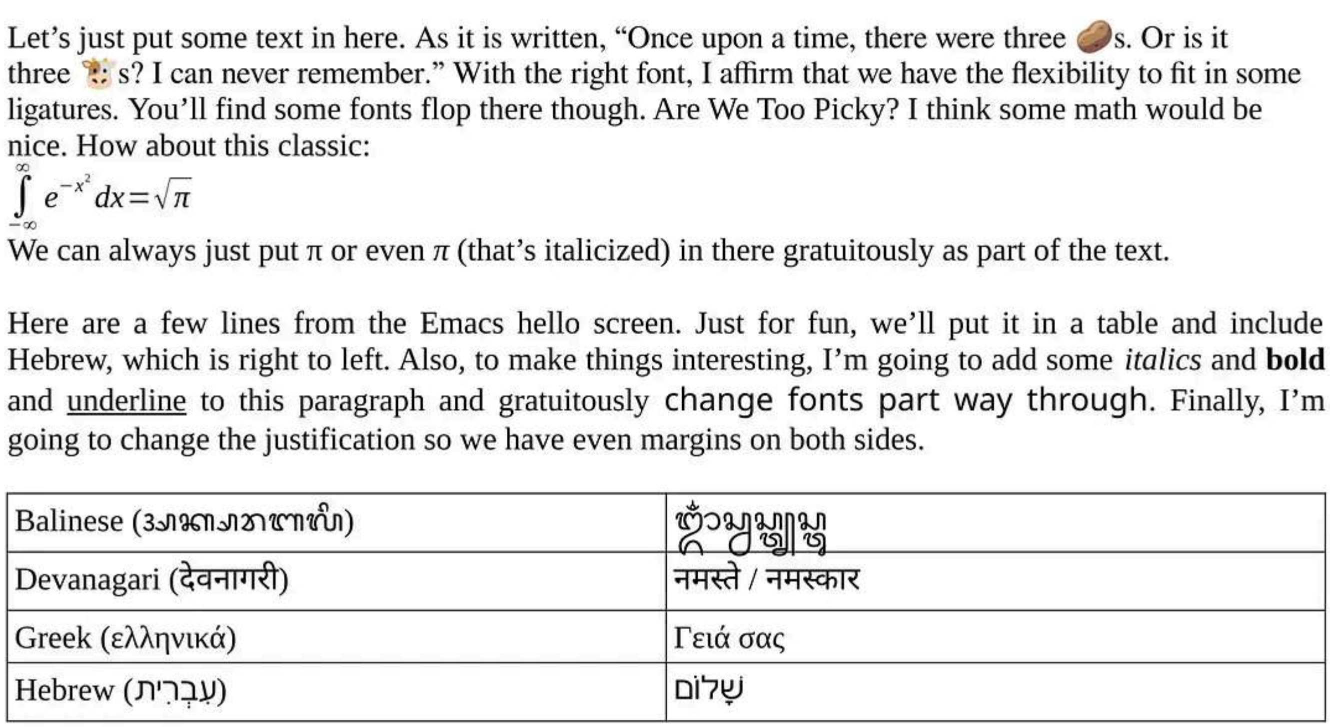
The previous part, *Text in PDF: Basic Operators*, contains information about the basic operators used to position and display text. That article uses a simple PDF that uses a built-in font and displays only English text. This article uses a file generated by the LibreOffice word processing software and that uses several fonts and contains math and languages that use non-Latin alphabets. All articles in this series use the convention of marking paragraphs or parenthetical remarks with the 🔍 symbol when they contain deeper information that will be of interest to some readers but can be safely skipped without losing the flow of the material.

As discussed in Part 2, you can visit the jberkenbilt/pdf-text-blog GitHub repository to see the sample file used in this post in both PDF form and as a textual representation that's tweaked for easy viewing in a browser or text editor.

This post refers to advanced.pdf and contains links to specific parts of the tweaked textual representation of its code. In the textual representation, I have redacted some binary streams and re-encoded some characters so it displays correct as text. (Note that the line numbers in the real PDF and in the text representation diverge after object 54, but we don't refer to anything after the point of divergence in this post.)

Overview of Sample File

Here you can see a rendered version of advanced.pdf:


Let's just put some text in here. As it is written, "Once upon a time, there were three 🥔s. Or is it three 🐮s? I can never remember." With the right font, I affirm that we have the flexibility to fit in some ligatures. You'll find some fonts flop there though. Are We Too Picky? I think some math would be nice. How about this classic:

$$\int_{-\infty}^{\infty} e^{-x^2} dx = \sqrt{\pi}$$

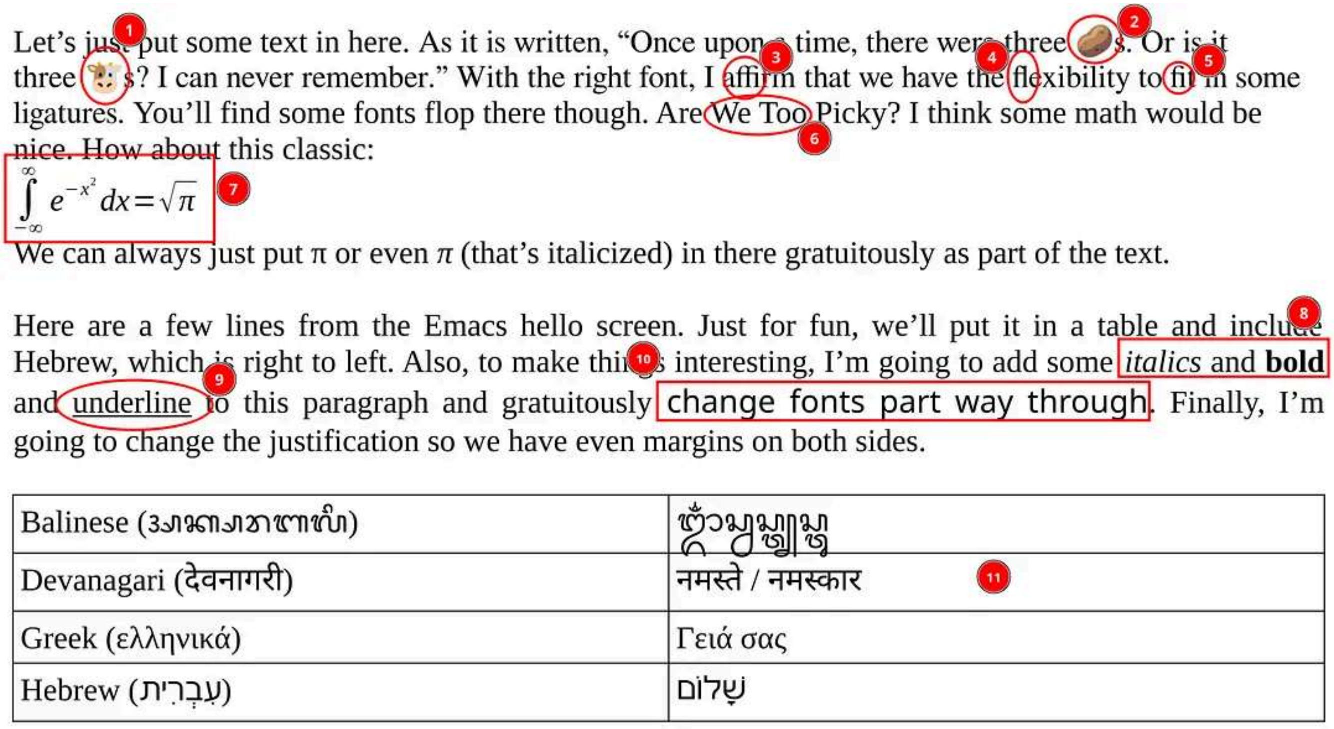
We can always just put π or even *π* (that's italicized) in there gratuitously as part of the text.

Here are a few lines from the Emacs hello screen. Just for fun, we'll put it in a table and include Hebrew, which is right to left. Also, to make things interesting, I'm going to add some *italics* and **bold** and underline to this paragraph and gratuitously change fonts part way through. Finally, I'm going to change the justification so we have even margins on both sides.

Balinese (ᬅᬓ᭄ᬱᬭᬩᬮᬶ)	ᬒᬁᬲ᭄ᬯᬲ᭄ᬢ᭄ᬬᬲ᭄ᬢᬸ
Devanagari (देवनागरी)	नमस्ते / नमस्कार
Greek (ελληνικά)	Γειά σας
Hebrew (עִבְרִית)	שָׁלוֹם

rendering of advanced.pdf

Below, I'll draw your attention to several noteworthy features in this file. I'll discuss these in detail in Parts 4 and 5. In this document, I will cover the rest of the material you need to follow those examples.



Let's just put some text in here. As it is written, "Once upon a time, there were three 🥔s. Or is it three 🐮s? I can never remember." With the right font, I affirm that we have the flexibility to fit in some ligatures. You'll find some fonts flop there though. Are We Too Picky? I think some math would be nice. How about this classic:

$$\int_{-\infty}^{\infty} e^{-x^2}\,dx = \sqrt{\pi}$$

We can always just put π or even π (that's italicized) in there gratuitously as part of the text.

Here are a few lines from the Emacs hello screen. Just for fun, we'll put it in a table and include Hebrew, which is right to left. Also, to make things interesting, I'm going to add some *italics* and **bold** and underline to this paragraph and gratuitously change fonts part way through. Finally, I'm going to change the justification so we have even margins on both sides.

Balinese (ᬅᬓ᭄ᬱᬭᬩᬮᬶ)	ᬒᬁᬲ᭄ᬯᬲ᭄ᬢ᭄ᬬᬲ᭄ᬢᬸ
Devanagari (देवनागरी)	नमस्ते / नमस्कार
Greek (ελληνικά)	Γειά σας
Hebrew (עִבְרִית)	שָׁלוֹם

annotated rendering of advanced.pdf

- 1, 2: emoji
- 3, 4, 5: ligatures
- 6: kerning
- 7: mathematical formula
- 8: bold and italics
- 9: underlined text
- 10: multiple fonts
- 11: characters from non-Latin alphabets

- Also notice that the right margin is even on the second paragraph

Unicode Maps and Glyphs in Font Subsets

As we did in Part 2 with the basic file, let's start by finding the text at the beginning of the page. Since this is a QDF file, we can take the shortcut of search for the comment `%% Contents of page`, and then we should find a text block near the beginning of the contents. In there, we should see a `Tj` operator preceded by the string "Let's just put some text here," or at least something close to that, maybe with some extra numbers, right? Let's see. (Spoiler alert: it's not going to be that way.)

The content stream begins at line. 62 with the following text, which starts right after the stream keyword and ends at the end of the first q/Q block:

```
0.1 w
q 0 0.028 611.971 791.971 re
W* n
q 0 0 0 rg
BT
56.8 724.1 Td /F1 12 Tf[<01020304>55<0506070805030609080306050a0b020603020c03060
ET
Q
```

The first three lines have operators `w`, `re`, `W*`, and `n`. I'm going to gloss over that by just stating that it's setting the clipping region to something very close to the size of a US letter page (which 612×792 points). We'll ignore that. The fourth line contains `q 0 0 0 rg`, which we can recognize as beginning a graphics state isolation block and setting the fill color to black. Next we see `BT` to open a text block, a `Td` operator to set the beginning of the line

(relative to the origin since there is no preceding `Td` or `Tm` in this text object), a `Tf` operator to set the font to `/F1` at 12 points (noting that this is a different file from the one in Part 2, so we should not expect `/F1` to be the same font), and then we get to our `TJ` operator. But wait…something's not right. Instead of seeing "Let's just put some text here.", we have something that starts with `[<01020304>55<0506070805030609080306 05` …what's going on here? It looks like a binary string whose characters are `01`, `02`, `03`, `04`, then some spacing, then `05`, `06`, `07`, `08`, `05`, `03` …where's our text? Well, this is where it starts getting interesting.

I'll spoil the ending and tell you the punchline first. Then I'll go back and explain. Here's what's going on. The code points in this string, starting with `01`, are being assigned to the characters in the order in which they appear in the document. This is definitely not a standard encoding like ASCII or PDF Doc Encoding or Unicode. It's some one-off encoding unique to this font in this file! I like to call this "order-of-appearance" encoding (though I've never heard anyone else call it that), and it's extremely common in generated PDF files for reasons I'm about to explain. Let's do a quick sanity check. The first ten characters of our text are "Let's just". That includes the apostrophe and the space. If we look at the first ten code points, we start by counting up from `01` to `08`, and then we go back to `05` and `03`. Does that check out? Well, the first eight characters, "Let's ju" are all distinct. Then the ninth character "s" matches the fifth character "s", and the tenth character "t" matches the third character "t". So, yeah, it checks out. But how are you ever supposed to know? Surely you don't have to solve a puzzle to decode text strings, right? In a well-behaved PDF file, you don't. What do I mean by well-behaved? I'll come back to that. But first, let's take this apart.

In the basic example PDF we used in Part 2, the font labeled `/F1` was the built-in Helvetica font using PDF Doc Encoding. The font labels are arbitrary,

but it's a common convention for the first font to be `/F1`, the second one to be `/F2`, etc. Noticing any patterns here? It looks like this file is using order-of-appearance font numbering as well as order-of-appearance character encoding. Here's the definition of the font `/F1` from the previous example:

```
<<
  /BaseFont /Helvetica
  /Encoding /PdfDocEncoding
  /Subtype /Type1
  /Type /Font
>>
```

Pretty straightforward, right? Let's compare that with what we have for this file.

To trace this, I'll need to go to the page dictionary for page 1. I could trace it through from trailer to document root to `/Pages`, but since this post is not about PDF structure (see my earlier post, *The Structure of a PDF File*) and this is a QDF file, I can take a shortcut and search for the comment `%% Page 1`. This brings us to line 29, with the page dictionary covering lines 31 through 42. Here it is for reference.

Page dictionary for page 1

This shows the location of the content stream, sets the page bounding box with `/MediaBox`, and shows us that the resource dictionary is in object 7. Fonts are defined in the resource dictionary, so let's find that. We can find 7 0 obj at line 475:

Resource dictionary for page 1

Closer…but the font directory is still indirect and can be found in object 8, which is right below this starting on line 485:

Fonts used in page 1

On line 3 above, we can see that font `/F1` is defined in object 9, which starts at line 500 and which I have reformatted here, putting the widths together on fewer lines for brevity:

Font dictionary for /F1

There is a `/BaseFont` key that shows the font name as some six-character string followed by `+LiberationSerif`. I'm not going to get into the six-character prefix other than to say that these are assigned to allow multiple subsets of the same font. The existence of the prefix tells us that this is a *font subset*, and we can see that the original font is called Liberation Serif. The font descriptor in object 19 provides some additional metadata about the

font including a link to the binary data containing the contents of actual font file. I'm going to ignore that because it's not relevant to this article.

We also have an array called `/Widths` which contains the width of each character in the font in the same 1/1000ths of the font size units as in `TJ`. PDF generators often need to know character widths to do things like center or highlight text, and while drawing characters and interpreting font files requires specialized software, making calculations about string widths based on character widths is simple. Notice that the numbers in `/Widths` are different from each other. That's because this is a proportional font, meaning each character only takes up as much width as is needed. If this were a fixed-width font, all the numbers would be the same. I'll come back to widths later. (🔍 Also notice that the first entry is 0. This would correspond to code point 0, which is not being used in this file. You can use 0 as a valid code point, but embedding the `NUL` character in a string is hard in many programming languages, so people tend to avoid using it.)

Unlike in our basic PDF, there is no `/Encoding` key here. This is not a great shock since we have already noticed that there is no standard encoding in use for this font. Instead, we have the key `/ToUnicode` in object 20. Let's take a look. We can find this at line 870. Below is the first several lines.

From line 5 above, we can see that this is a stream. This stream contains some things that look kind of like PDF objects, but it's not PDF. It's actually its own separate syntax that's based more closely on PostScript, which is a predecessor to PDF and a language that still used in many printers. (🔍 PostScript bumper sticker: `honk { PS ❤️ } if`)

This stream contains the mapping between the code points in strings and the Unicode characters that they refer to. Line 11 establishes the ordering as

UCS (Universal Character System — we humans, and my apologies if you are not a human, think we're the only ones in the Universe, but that's another story). Line 16 tells us that the next one line (that's what the "`1`" is) is a range of characters in the map. In this case, we have `<00> <FF>`, which tells us that this is a single-byte encoding.

The good stuff starts on line 19 (which is line 888 in the actual file), which says the next 61 lines will contain mappings from code points in the font to Unicode characters. Let's see…we have `01 = 004C`, which is "L", `02 = 0065`, which is "e", `03 = 0074`, which is "t", `04 = 2019`, which is the apostrophe, and so on. I'm not going to repeat the whole table…you can see it above, and the codes do actually map to the characters we expect.

But wait, there's more. This `/ToUnicode` map is entirely optional. If you remove it, it doesn't change the appearance of the PDF file at all, though it does break the ability to cut and paste or otherwise extract text from the file because the PDF viewer no longer has any idea what character a code point refers to. If that's the case, you may ask, how can it display the right character? To answer that, you have to understand a little more about the font subset. A font (at least most font formats including the ones that can be embedded in PDF) is basically a collection of tables of information about the font. One of those tables maps an entry in the font to its *glyph*. A glyph is just the graphical representation of a character or symbol, and PDF fonts contain descriptions of glyphs in the form of vector drawing commands. So, a simplified picture is that a font is an array of glyphs. Now for the punchline: a code point in a font subset is just an index into the table of glyphs!

🔍 What happens if `/ToUnicode` is missing? The behavior varies by the PDF viewer, but I've seen a handful of things. Most viewers will treat the characters as if they are using a standard encoding, which is a sensible

default in case the PDF writer just forgot to indicate the encoding. That means that, when you cut and paste, you will get garbage. In the standard encodings, printable characters start with hex `20`, which is 32, so sometimes the first 31 character to appear won't be selectable, and subsequent characters will paste as whatever character has that code point in the standard encoding. If you have a PDF file where you can select text and paste it but it pastes as garbage characters, there's a good chance you're dealing with a file where `/ToUnicode` is missing for one or more fonts. If you want to recover the text in these cases, you will need to fall back to OCR. In principle, it would be possible to do slightly better than OCR. For one thing, the characters are perfectly drawn, so you don't have noise or extra dots. Also, you know with certainty that two characters with the same code in the same font are the same character, and that would give you additional clues. I'm not aware of whether anyone has written software to try to reconstruct a missing `/ToUnicode` map. It's also worth noting that, while `/ToUnicode` is important, it's not the only thing. There are other things you can do in PDF around accessibility to make it possible to unambiguously extract text from a PDF in a way that preserves reading order. This is useful for archival and accessibility purposes.

So how does this work? And why is it done this way? Unicode is big. Fonts are big. A key feature of PDF is that everyone who looks at a particular PDF file will see the same thing. That means everything that is needed to render the PDF must be contained in the PDF, and that includes all the font information (unless you stick with the 14 built-in fonts, but no PDF that's generated from word processing or typesetting software will do that). Contrast this with programs like Microsoft Word. Word processors typically require you to have the font on your system. Have you ever opened a word file and had the text not be readable because of a missing font? Well, that should never happen with PDF because fonts are embedded. But if a PDF file had to contain every

single glyph that the font can draw, the PDF file would be needlessly large and would contain a lot of data that is not needed to render the file.

What happens instead is that, as the PDF is being generated, the first time a font is used, a subset is created (hence `/F1` being the first font used), and the first time a character is used in that font, that character is plucked out of the complete font file from the system and dropped into a font subset. The character is assigned the next available code point, typically starting with 1, and that code point is an index into the font so the PDF viewer can get the glyph and any other relevant information about the font. In the example above, the file starts with text in Liberation Serif, so that font gets assigned `/F1`. Then the first character, "L", gets assigned code point 1, the second "e" gets 2, and so on.

Ready for Part 4

That covers what you know to understand text encoding, but examples teach better than words by themselves, so continue with Part 4, *Text in PDF: Fonts and Spacing*, and Part 5, *Text in PDF: Non-Latin Alphabets* to see an examination of each of the special features in our PDF file. I'll also show you a technique I use to find my way around PDF files: adding color change operators at various points in a file.

Pdf | Pdf Text Extraction | Unicode

Written by Jay Berkenbilt

129 followers · 3 following

Follow

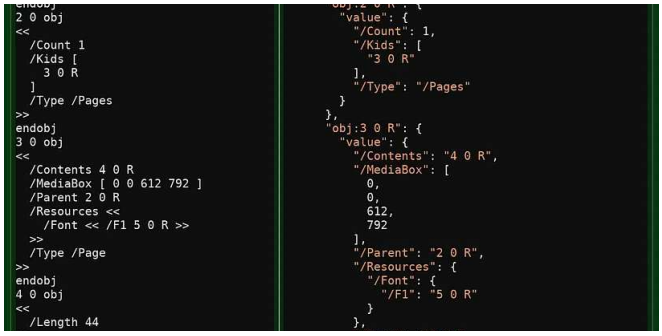
Jay is a software engineer/architect with a focus on low-level coding and infrastructure. He is the creator of qpdf, an open source PDF manipulation library.

No responses yet

Mark Tan

What are your thoughts?

More from Jay Berkenbilt



Jay Berkenbilt

Examining a PDF File with qpdf

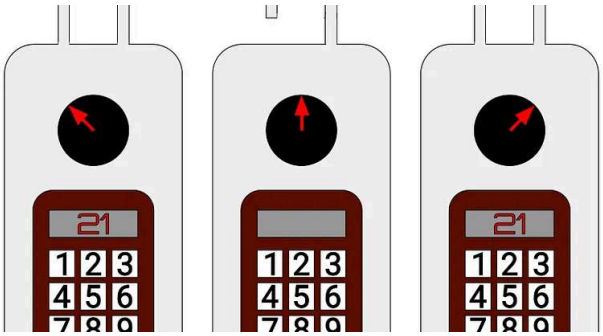
Introduction

In AWS in Plain English by Jay Berkenbilt

Rotating Secrets with AWS RDS Proxy

AWS makes it easy for you to rotate your RDS credentials using SecretsManager. But what ...

Nov 7, 2022 94 1



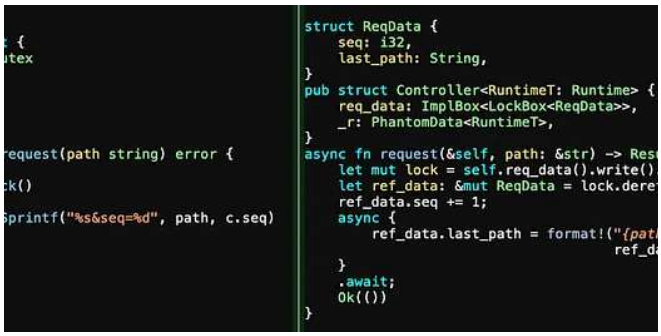
Jay Berkenbilt

Understanding Why PKI Works

A Mental Model

Jan 14, 2024 3 2

Sep 26, 2023 25 4



In Rustaceans by Jay Berkenbilt

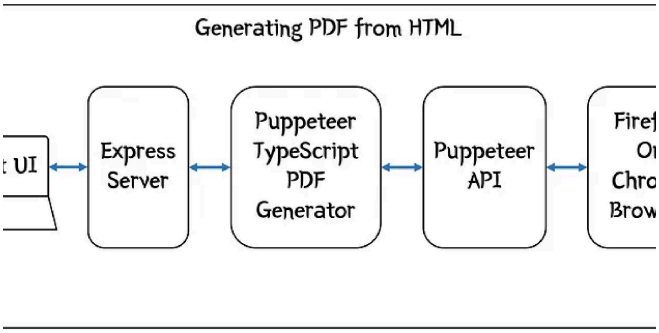
From Go to Rust 2: Generic async Lock

Generic, async Runtime-Agnostic Locks and impl Types

Jan 3 109 1

See all from Jay Berkenbilt

Recommended from Medium

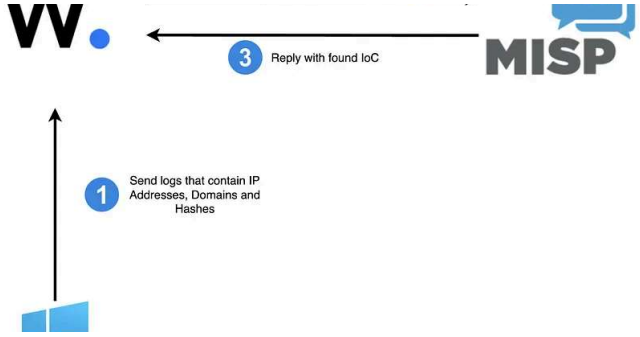


Martin Hodges

HTML to PDF Service

Whilst developing my applications, I sometimes want to create and save a PDF fil...

Sep 13 2



Arbnor Mustafa

WAZUH x MISP Integration

This blog post discusses how to integrate Wazuh with the MISP API, making threat...

Aug 20 71 2

Viacheslav Klavdiiev

Zoneless Change Detection in Angular 20: How to Remove Zone....

With the latest versions of Angular, it's finally possible to fully disable Zone.js and switch...

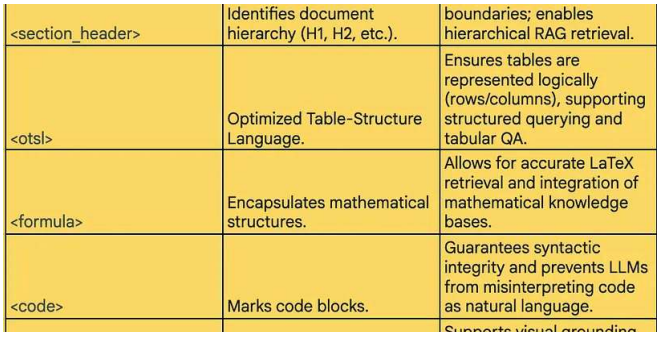
Jul 4 83 1

Kartikey Kumar

LangChain x TypeScript: Build Real-World AI Workflows Like a Pro

If you're looking to move beyond toy prompts and build real-world AI tools, LangChain in...

Jul 29 10



<section_header>	Identifies document hierarchy (H1, H2, etc.).	boundaries; enables hierarchical RAG retrieval.
<otsl>	Optimized Table-Structure Language.	Ensures tables are represented logically (rows/columns), supporting structured querying and tabular QA.
<formula>	Encapsulates mathematical structures.	Allows for accurate LaTeX retrieval and integration of mathematical knowledge bases.
<code>	Marks code blocks.	Guarantees syntactic integrity and prevents LLMs from misinterpreting code as natural language.

Usman Writes

Web Scraping vs Web Crawling: Understanding the Difference

You might have heard of web scraping and web crawling before. That is no surprise...

Dec 9 2 1

Mustafa Genc

Revolutionizing Document Intelligence: The Strategic...

1. The Foundational Deficit in Sequential Document Layout Analysis (DLA)

Nov 16 11 1

See more recommendations