

html to pdf vs jspdf or pdfkit html to pdf i lack control jspdf is bad

When it comes to converting HTML to PDF, there are several tools and libraries available, each with its own strengths and weaknesses. Below is a comparison of **HTML to PDF conversion using libraries like jsPDF, PDFLib, and other tools**:

1. HTML to PDF using jsPDF

jsPDF is a popular JavaScript library for generating PDFs directly in the browser. It allows you to create PDFs from scratch or convert HTML/CSS content into PDFs.

Pros:

- **Client-side:** Works entirely in the browser, so no server-side processing is required.
- **Lightweight:** Easy to integrate into front-end projects.
- **Customizable:** You can manually design the PDF layout and add text, images, and shapes.
- **Supports basic HTML/CSS:** Can render simple HTML and CSS content.

Cons:

- **Limited HTML/CSS support:** Complex layouts, advanced CSS (e.g., flexbox, grid), and external stylesheets may not render correctly.
 - **No native HTML rendering:** jsPDF does not natively support HTML rendering. You need to use plugins like html2canvas or html2pdf to convert HTML to an image or PDF, which can lead to quality issues.
 - **Performance issues:** Rendering large or complex HTML content can be slow and may crash the browser.
 - **Lack of control:** Fine-grained control over PDF generation (e.g., pagination, headers, footers) is limited.
-

2. HTML to PDF using PDFLib

PDFLib is a powerful library for creating and manipulating PDFs programmatically. It is often used in server-side environments (e.g., Node.js).

Pros:

- **High control:** You have full control over the PDF generation process, including text, images, and vector graphics.
- **Server-side:** Works well in backend environments, making it suitable for large-scale or complex PDF generation.

- **No HTML/CSS limitations:** Since it doesn't rely on HTML/CSS rendering, you avoid issues with unsupported features.

Cons:

- **No native HTML support:** PDFLib does not natively support converting HTML to PDF. You need to manually recreate the HTML layout using PDFLib's API, which can be time-consuming.
- **Steeper learning curve:** Requires more effort to design and implement PDF layouts compared to HTML-based tools.
- **Not ideal for dynamic content:** If your content is highly dynamic or relies on complex HTML/CSS, PDFLib may not be the best choice.

3. HTML to PDF using dedicated tools (e.g., Puppeteer, wkhtmltopdf)

There are dedicated tools and libraries specifically designed for converting HTML to PDF with better support for HTML/CSS rendering.

Examples:

- **Puppeteer:** A Node.js library that controls a headless Chrome browser to render HTML and generate PDFs.
- **wkhtmltopdf:** A command-line tool that uses the WebKit rendering engine to convert HTML to PDF.

Pros:

- **Full HTML/CSS support:** These tools can render complex HTML and CSS, including modern layouts (flexbox, grid) and external stylesheets.
- **High-quality output:** The resulting PDFs are visually accurate and match the HTML content.
- **Server-side:** Can handle large or complex documents without browser limitations.

Cons:

- **Server-side only:** Requires a backend environment to run.
- **Performance overhead:** Rendering HTML in a headless browser can be resource-intensive.
- **Configuration required:** May need additional setup and configuration for optimal results.

Comparison Table

Feature/Tool	jsPDF	PDFLib	Puppeteer/wkhtmltopdf
HTML/CSS Support	Limited	None (manual layout)	Full support

Ease of Use	Easy (client-side)	Moderate (manual design)	Moderate (server-side)
Control over PDF	Low	High	High
Performance	Poor for complex content	Good	Good (but resource-heavy)
Use Case	Simple client-side PDFs	Custom PDFs (no HTML)	Complex HTML to PDF

Which Should You Choose?

- **jsPDF:** Use for simple, client-side PDF generation where you don't need advanced HTML/CSS support.
- **PDFLib:** Use when you need full control over PDF creation and don't rely on HTML/CSS for layout.
- **Puppeteer/wkhtmltopdf:** Use for high-quality HTML to PDF conversion with full support for modern web standards.

Recommendation

If your primary goal is to convert HTML to PDF with high fidelity, **Puppeteer** or **wkhtmltopdf** is the best choice. For more control over PDF generation (without relying on HTML), **PDFLib** is a great option. **jsPDF** is suitable for lightweight, client-side use cases but may not meet your needs if you require precise HTML/CSS rendering.

i dont have server as am working on a bex

If you don't have access to a server and are working entirely in a **browser environment** (e.g., a BEX or browser extension), your options for converting HTML to PDF are limited to **client-side libraries**. Here's a breakdown of the best approaches for your scenario:

1. jsPDF with html2canvas

This is a common combination for generating PDFs directly in the browser. html2canvas renders the HTML content as an image, and jsPDF converts that image into a PDF.

Steps:

1. Use html2canvas to capture the HTML element as an image.
2. Use jsPDF to add the image to a PDF document.

Example Code:

```
import jsPDF from 'jspdf';
import html2canvas from 'html2canvas';

const element = document.getElementById('content-to-export');
```

```
html2canvas(element).then((canvas) => {  
  const imgData = canvas.toDataURL('image/png');  
  const pdf = new jsPDF('p', 'mm', 'a4');  
  const imgWidth = 210; // A4 width in mm  
  const imgHeight = (canvas.height * imgWidth) / canvas.width;  
  
  pdf.addImage(imgData, 'PNG', 0, 0, imgWidth, imgHeight);  
  pdf.save('output.pdf');  
});
```

Pros:

- Works entirely in the browser.
- Easy to implement for simple use cases.
- No server required.

Cons:

- **Quality issues:** Since the content is rendered as an image, text may appear blurry or pixelated.
 - **Limited interactivity:** Links, buttons, and other interactive elements won't work in the PDF.
 - **Performance:** Large or complex HTML content can cause performance issues.
-

2. html2pdf.js

html2pdf.js is a wrapper around jsPDF and html2canvas that simplifies the process of converting HTML to PDF. It handles many of the complexities for you.

Steps:

1. Include the html2pdf.js library.
2. Use its API to convert HTML to PDF.

Example Code:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/html2pdf.js/0.10.1/html2pdf.bundle.min.js"></script>  
<script>  
  const element = document.getElementById('content-to-export');  
  html2pdf().from(element).save();  
</script>
```

Pros:

- Simple and easy to use.
- Handles pagination and scaling automatically.
- No server required.

Cons:

- Still relies on html2canvas, so the same quality and performance issues apply.
 - Limited control over advanced PDF features (e.g., headers, footers).
-

3. PDFMake

PDFMake is another client-side library for generating PDFs. Unlike jsPDF, it doesn't rely on HTML/CSS rendering. Instead, you define the PDF structure programmatically.

Example Code:

```
import pdfMake from 'pdfmake/build/pdfmake';
import pdfFonts from 'pdfmake/build/vfs_fonts';

pdfMake.vfs = pdfFonts.pdfMake.vfs;

const docDefinition = {
  content: [
    { text: 'Hello, this is a PDF!', style: 'header' },
    { text: 'This is a paragraph.' },
  ],
  styles: {
    header: {
      fontSize: 18,
      bold: true,
    },
  },
};

pdfMake.createPdf(docDefinition).download('output.pdf');
```

Pros:

- Full control over PDF content and layout.
- No reliance on HTML/CSS rendering.
- Works entirely in the browser.

Cons:

- Requires manual definition of the PDF structure (not ideal for complex HTML content).
 - Limited support for dynamic HTML-to-PDF conversion.
-

4. Browser Print to PDF

Modern browsers have built-in support for saving web pages as PDFs using the **Print dialog**. You can trigger this programmatically in your BEX.

Steps:

1. Use the `window.print()` method to open the browser’s print dialog.
2. Select “Save as PDF” as the destination.

Example Code:

```
window.print();
```

Pros:

- No additional libraries required.
- High-quality output with full HTML/CSS support.
- Works entirely in the browser.

Cons:

- Requires user interaction (cannot be fully automated).
- Limited control over the PDF output (e.g., margins, headers, footers).

Comparison Table

Feature/Tool	jsPDF + html2canvas	html2pdf.js	PDFMake	Browser Print to PDF
HTML/CSS Support	Limited (image-based)	Limited (image-based)	None (manual layout)	Full support
Ease of Use	Moderate	Easy	Moderate	Very easy
Control over PDF	Low	Low	High	Low
Quality	Low (image-based)	Low (image-based)	High	High
Automation	Fully automated	Fully automated	Fully automated	Requires user interaction

Recommendation

- If you need **quick and simple HTML-to-PDF conversion** and can tolerate some quality loss, use **html2pdf.js**.
- If you need **full control over the PDF layout** and don’t mind defining the structure manually, use **PDFMake**.
- If you want **high-quality output** and don’t mind user interaction, use the **browser’s Print to PDF** feature.

For most BEX use cases, **html2pdf.js** is likely the best balance between ease of use and functionality. However, if quality is critical, consider prompting the user to use the browser’s built-in Print to PDF feature.

we forgot pdfmake

You're absolutely right! I did mention **PDFMake** in the previous response, but let's dive deeper into it since it's a powerful client-side library for generating PDFs without needing a server. Here's a more detailed look at **PDFMake** and how it compares to other options like **jsPDF** and **html2pdf.js**:

PDFMake Overview

PDFMake is a JavaScript library that allows you to create PDFs programmatically in the browser. Unlike tools that rely on HTML/CSS rendering (e.g., jsPDF + html2canvas), PDFMake uses a **declarative approach** where you define the PDF structure using a JavaScript object.

Key Features:

- **Client-side:** Works entirely in the browser.
 - **Declarative API:** Define the PDF structure using a simple JSON-like object.
 - **Full control:** Supports advanced features like headers, footers, tables, columns, and custom fonts.
 - **No HTML/CSS limitations:** Since it doesn't rely on HTML/CSS, you avoid rendering issues.
-

How to Use PDFMake

1. Installation

You can include PDFMake via a CDN or install it via npm:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/pdfmake/0.2.7/pdfmake.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/pdfmake/0.2.7/vfs_fonts.js"></script>
```

Or via npm:

```
npm install pdfmake
```

2. Basic Example

Here's an example of creating a simple PDF with text, styles, and a table:

```
const docDefinition = {
  content: [
    { text: 'Hello, this is a PDF!', style: 'header' },
    { text: 'This is a paragraph.' },
  ],
  table: {
    body: [
      ['Column 1', 'Column 2', 'Column 3'],
      ['Row 1', 'Row 1', 'Row 1'],
      ['Row 2', 'Row 2', 'Row 2'],
    ],
  },
}
```

```
},
},
],
styles: {
  header: {
    fontSize: 18,
    bold: true,
    margin: [0, 0, 0, 10], // [left, top, right, bottom]
  },
},
};

// Generate and download the PDF
pdfMake.createPdf(docDefinition).download('output.pdf');
```

3. Advanced Features

- **Headers and Footers:** Add headers and footers to every page.
- **Columns:** Split content into multiple columns.
- **Images:** Embed images in the PDF.
- **Custom Fonts:** Use custom fonts by loading them into the `vfs_fonts.js` file.

Pros of PDFMake

1. **Full Control:** You can define every aspect of the PDF layout, including margins, styles, and page breaks.
2. **No HTML/CSS Limitations:** Since it doesn't rely on HTML/CSS, you avoid rendering issues with complex layouts.
3. **Client-Side:** Works entirely in the browser, so no server is required.
4. **Rich Features:** Supports tables, columns, headers, footers, and more.

Cons of PDFMake

1. **Manual Layout:** You need to define the PDF structure manually, which can be time-consuming for complex documents.
2. **No HTML Support:** If your content is already in HTML, you'll need to convert it to PDFMake's format, which can be tedious.
3. **Learning Curve:** The declarative API requires some time to learn, especially for advanced layouts.

Comparison with jsPDF and html2pdf.js

Feature/Tool	PDFMake	jsPDF + html2canvas	html2pdf.js
--------------	---------	---------------------	-------------

HTML/CSS Support	None (manual layout)	Limited (image-based)	Limited (image-based)
Ease of Use	Moderate	Easy	Very easy
Control over PDF	High	Low	Low
Quality	High	Low (image-based)	Low (image-based)
Use Case	Custom PDFs (no HTML)	Simple client-side PDFs	Simple client-side PDFs

When to Use PDFMake

- You need **full control** over the PDF layout and design.
 - Your content is **not already in HTML** or you're willing to manually define the structure.
 - You want to avoid the **quality issues** associated with image-based rendering (e.g., jsPDF + html2canvas).
-

Example Use Case for PDFMake

If you're working on a BEX (browser extension) and need to generate a **custom report** or **invoice** with a specific layout, PDFMake is a great choice. For example:

```
const docDefinition = {
  content: [
    { text: 'Invoice', style: 'header' },
    { text: 'Date: ' + new Date().toLocaleDateString() },
    {
      table: {
        headerRows: 1,
        widths: ['*', 'auto', 'auto'],
        body: [
          ['Item', 'Quantity', 'Price'],
          ['Product A', 2, '$20.00'],
          ['Product B', 1, '$15.00'],
          ['Total', '', '$35.00'],
        ],
      },
    },
  ],
  styles: {
    header: {
      fontSize: 18,
      bold: true,
      alignment: 'center',
      margin: [0, 0, 0, 10],
    },
  },
};

pdfMake.createPdf(docDefinition).download('invoice.pdf');
```

Conclusion

If you're working in a **browser-only environment** (like a BEX) and need to generate **high-quality, custom PDFs**, **PDFMake** is an excellent choice. However, if your content is already in HTML and you need a quick solution, **html2pdf.js** might be more suitable despite its limitations. Choose the tool that best fits your use case!

html2pdf.js i had issue with a while back some pdf readers failed to display the pages they would just be blank 2 / 2

The issue you experienced with **html2pdf.js** where some PDF readers displayed blank pages is a known problem. This typically happens because **html2pdf.js** relies on **html2canvas** to render HTML content as an image, and then embeds that image into a PDF using **jsPDF**. While this approach works in many cases, it can lead to compatibility issues with certain PDF readers, especially if:

1. **The PDF is image-based:** Some PDF readers struggle with image-heavy PDFs or improperly embedded images.
2. **Fonts or styles are missing:** If fonts or styles aren't properly embedded, the content may not render correctly.
3. **Large or complex content:** Rendering large or complex HTML content can cause issues with memory or rendering.

Why This Happens

- **html2canvas Limitations:** html2canvas renders HTML as an image, which can result in large file sizes and compatibility issues.
- **jsPDF Limitations:** jsPDF is not designed for complex HTML/CSS rendering, and its image embedding can sometimes fail in certain PDF readers.
- **PDF Reader Differences:** Some PDF readers (e.g., older versions of Adobe Acrobat Reader) may not handle image-based PDFs well.

Workarounds and Solutions

If you're experiencing issues with **html2pdf.js**, here are some potential solutions:

1. Use a Different Library

If **html2pdf.js** is causing issues, consider switching to a more robust library or tool. Here are some alternatives:

- **Puppeteer (if you can use a server):** Puppeteer is a Node.js library that uses a headless Chrome browser to generate high-quality PDFs. It supports full HTML/CSS rendering and produces PDFs that are compatible with all readers.

- **PDFMake:** As discussed earlier, PDFMake allows you to define PDFs programmatically without relying on HTML/CSS rendering. This avoids the issues associated with image-based PDFs.
-

2. Improve html2pdf.js Output

If you want to stick with **html2pdf.js**, try these optimizations to reduce the likelihood of blank pages:

- **Reduce Image Size:** Compress images in your HTML to reduce the size of the resulting PDF.
 - **Simplify HTML/CSS:** Avoid complex layouts, animations, or external stylesheets that might not render correctly.
 - **Use Custom Fonts:** Ensure fonts are properly embedded in the PDF. You can use pdfMake or jsPDF with custom fonts for better compatibility.
 - **Test in Multiple Readers:** Test the generated PDF in different PDF readers (e.g., Adobe Acrobat, Foxit Reader, Chrome's built-in PDF viewer) to identify specific compatibility issues.
-

3. Use Browser Print to PDF

Modern browsers have a built-in **Print to PDF** feature that produces high-quality, text-based PDFs. You can trigger this programmatically in your BEX:

```
window.print();
```

Pros:

- High-quality output with full HTML/CSS support.
- No additional libraries required.
- Works in all browsers.

Cons:

- Requires user interaction (cannot be fully automated).
 - Limited control over the PDF output (e.g., margins, headers, footers).
-

4. Use PDFMake for Text-Based PDFs

If your content is mostly text-based, **PDFMake** is a great alternative. It generates text-based PDFs that are compatible with all readers. Here's an example:

```
const docDefinition = {
  content: [
    { text: 'Hello, this is a PDF!', style: 'header' },
    { text: 'This is a paragraph.' },
  ],
  styles: {
```

```
header: {
  fontSize: 18,
  bold: true,
},
},
};

pdfMake.createPdf(docDefinition).download('output.pdf');
```

5. Debugging html2pdf.js Issues

If you want to debug the issue with **html2pdf.js**, try the following:

- **Check the Console:** Look for errors or warnings in the browser console.
- **Test with Simple HTML:** Start with a simple HTML snippet and gradually add complexity to identify the root cause.
- **Update Dependencies:** Ensure you're using the latest versions of html2pdf.js, html2canvas, and jsPDF.

Comparison of Solutions

Solution	Pros	Cons
html2pdf.js	Easy to use, client-side	Image-based PDFs, compatibility issues, large file sizes
PDFMake	Text-based PDFs, full control, compatible with all readers	Manual layout, no HTML/CSS support
Puppeteer	High-quality PDFs, full HTML/CSS support	Requires a server, not client-side
Browser Print to PDF	High-quality, no libraries required	Requires user interaction, limited control

Recommendation

If you're working in a **browser-only environment** (like a BEX) and need a reliable solution:

- Use **PDFMake** for text-based PDFs or custom layouts.
- Use **Browser Print to PDF** for high-quality, HTML-based PDFs (if user interaction is acceptable).
- Avoid **html2pdf.js** if you're experiencing consistent issues with blank pages.

If you can use a server (even temporarily), **Puppeteer** is the best option for high-quality, HTML-to-PDF conversion.

is there a way to simulate an http url form the same app like to use fonts in the assets folder since point path wont work pdfMake.fonts = { yourFontName: { normal:

'https://example.com/fonts/fontFile.ttf', bold: 'https://example.com/fonts/fontFile2.ttf', italics: 'https://example.com/fonts/fontFile3.ttf', bolditalics: 'https://example.com/fonts/fontFile4.ttf' },

Yes, you can simulate an HTTP URL to load fonts from your local assets folder in a browser-based application (like a BEX or a web app). This is particularly useful for **PDFMake**, which requires fonts to be loaded via URLs. Here's how you can achieve this:

1. Use a Local Web Server

If you're developing locally, you can use a simple web server to serve your assets folder. This will allow you to access fonts via an HTTP URL.

Options:

- **http-server**: A zero-configuration command-line HTTP server.
- **Live Server**: A VS Code extension that serves your project folder.
- **Python Simple HTTP Server**: Built-in Python module to serve files.

Example with http-server:

1. Install http-server globally:

```
npm install -g http-server
```

2. Navigate to your project folder and start the server:

```
http-server
```

3. Access your fonts via:

```
http://localhost:8080/assets/fonts/fontFile.ttf
```

2. Use a Data URL

If you don't want to set up a local server, you can embed the font files directly into your JavaScript code using **Data URLs**. This approach encodes the font file as a base64 string.

Steps:

1. Convert your font files to base64 using an online tool or a command-line utility:

```
base64 fontFile.ttf > fontFile.txt
```

2. Use the base64 string in your code:

```
pdfMake.fonts = {  
  yourFontName: {  
    normal: 'data:font/truetype;base64,<base64-string>',  
    bold: 'data:font/truetype;base64,<base64-string>',  
    italics: 'data:font/truetype;base64,<base64-string>',  
    bolditalics: 'data:font/truetype;base64,<base64-string>',  
  },  
}
```

```
};
```

Example:

```
pdfMake.fonts = {  
  Roboto: {  
    normal: 'data:font/truetype;base64,AAEAAAASAQA...',  
    bold: 'data:font/truetype;base64,AAEAAAASAQA...',  
    italics: 'data:font/truetype;base64,AAEAAAASAQA...',  
    bolditalics: 'data:font/truetype;base64,AAEAAAASAQA...',  
  },  
};
```

3. Use a Blob URL

If you don't want to use base64 encoding, you can create a **Blob URL** for the font file. This approach dynamically generates a URL for the font file stored in memory.

Steps:

1. Fetch the font file from your assets folder.
2. Create a Blob URL for the font file.
3. Use the Blob URL in PDFMake.

Example:

```
async function loadFonts() {  
  const response = await fetch('/assets/fonts/fontFile.ttf');  
  const fontData = await response.blob();  
  const fontUrl = URL.createObjectURL(fontData);
```

```
pdfMake.fonts = {  
  yourFontName: {  
    normal: fontUrl,  
    bold: fontUrl, // Repeat for other styles if needed  
  },  
};  
}
```

```
loadFonts();
```

4. Use a Service Worker (Advanced)

If you're building a Progressive Web App (PWA) or a browser extension, you can use a **Service Worker** to intercept requests and serve files from your local assets folder.

Steps:

1. Register a service worker in your app.
2. Use the service worker to intercept requests for fonts and serve them from the local assets folder.

Example:

```
// Service Worker (sw.js)
self.addEventListener('fetch', (event) => {
  const url = new URL(event.request.url);
  if (url.pathname.startsWith('/fonts/')) {
    event.respondWith(
      fetch(`/assets${url.pathname}`)
    );
  }
});
```

```
// Register the Service Worker
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/sw.js');
}
```

Then, use the fonts in PDFMake:

```
pdfMake.fonts = {
  yourFontName: {
    normal: '/fonts/fontFile.ttf',
    bold: '/fonts/fontFile2.ttf',
  },
};
```

5. Use a Browser Extension API (for BEX)

If you're working on a **Browser Extension (BEX)**, you can use the extension's API to access local files and serve them via a simulated HTTP URL.

Example:

1. Use the `chrome.runtime.getURL` method to generate a URL for the font file.
2. Use the generated URL in PDFMake.

```
const fontUrl = chrome.runtime.getURL('/assets/fonts/fontFile.ttf');
```

```
pdfMake.fonts = {
  yourFontName: {
    normal: fontUrl,
    bold: fontUrl, // Repeat for other styles if needed
  },
};
```

Summary of Approaches

Approach	Pros	Cons
Local Web Server	Simple to set up, works for development	Requires running a server
Data URL	No server required, fonts embedded in code	Increases JavaScript file size
Blob URL	No server required, dynamic URL generation	Requires fetching the font file
Service Worker	Advanced, works for PWAs and browser extensions	Requires service worker setup

Browser Extension API	Works for BEX, no server required	Limited to browser extensions
------------------------------	-----------------------------------	-------------------------------

Recommendation

- For **local development**, use a **local web server** (e.g., http-server).
- For **production**, use **Data URLs** or **Blob URLs** to avoid dependency on a server.
- For **browser extensions**, use the **Browser Extension API** or **Service Workers**.

Let me know if you need further clarification or help implementing any of these solutions!