INVOICE PRINTING OPTIONS

When generating a sales invoice in JavaScript with approximately 100 items and pagination, you need to ensure the invoice is properly formatted and paginated for printing. While there are different ways to handle this, here are some best practices and external libraries that can help streamline the process.

**Best Practices for Generating a Sales Invoice with Pagination:**

1. **HTML Structure**: Structure your invoice in HTML, ensuring that each item is in a table or a list format. This is crucial for proper styling and pagination.

Example structure:

<div id="invoice">

<div class="header">

<!-- Invoice Header -->

</div>

<div class="items">

<!-- Invoice Items will go here -->

</div>

</div>

1. **Pagination**: For pagination, you can break the items into chunks. If you have 100 items, for example, you can break them into groups of 20 per page. You'll need to manage pagination logic by creating multiple pages dynamically.
2. **CSS for Printing**: Use CSS to control the appearance when printing. Specifically, you can use the @media print query to style the content for print and adjust page breaks.

Example CSS for page breaks:

@media print {

.page {

page-break-before: always;

}

.items {

display: block;

width: 100%;

}

}

1. **JavaScript for Dynamic Pagination**: Use JavaScript to dynamically handle the pagination. You can generate a set of pages based on the number of items in your invoice.

Example JavaScript:

const itemsPerPage = 20;

const items = [...]; // Array of invoice items

function generateInvoicePages() {

let pages = [];

for (let i = 0; i < items.length; i += itemsPerPage) {

const pageItems = items.slice(i, i + itemsPerPage);

pages.push(pageItems);

}

return pages;

}

function printInvoice() {

const pages = generateInvoicePages();

let printWindow = window.open('', '\_blank', 'width=800,height=600');

let htmlContent = `<html><head><title>Invoice</title><style>@media print { .page { page-break-before: always; } }</style></head><body>`;

pages.forEach((page, index) => {

htmlContent += `<div class="page"><h2>Page ${index + 1}</h2><table>`;

page.forEach(item => {

htmlContent += `<tr><td>${item.name}</td><td>${item.price}</td></tr>`;

});

htmlContent += `</table></div>`;

});

htmlContent += `</body></html>`;

printWindow.document.write(htmlContent);

printWindow.document.close();

printWindow.print();

}

**External Libraries and Tools for Pagination and Printing**

1. **jsPDF**: jsPDF is a popular JavaScript library that can generate PDFs, which can be used for invoices. It has good support for pagination and text formatting.

Example usage of jsPDF:

const { jsPDF } = require("jspdf");

const doc = new jsPDF();

const items = [...]; // Your invoice items

const itemsPerPage = 20;

function createInvoice() {

let yPosition = 10;

for (let i = 0; i < items.length; i++) {

if (i > 0 && i % itemsPerPage === 0) {

doc.addPage();

yPosition = 10;

}

doc.text(`${items[i].name} - ${items[i].price}`, 10, yPosition);

yPosition += 10;

}

doc.save("invoice.pdf");

}

* + **Link**: [jsPDF Documentation](https://github.com/parallax/jsPDF)

1. **Print.js**: Print.js is a simple and effective JavaScript library that helps print HTML elements or PDFs. It handles both HTML printing and file generation for printing purposes.

Example usage of Print.js:

printJS({

printable: 'invoice',

type: 'html',

style: '@media print { .page { page-break-before: always; }}',

});

* + **Link**: [Print.js Documentation](https://printjs-4de6.kxcdn.com/)

1. **Page.js or Pagination Libraries**: For handling pagination, you could also use a pagination library like **Page.js** or **jQuery Pagination Plugin** to handle the display of multiple pages of content in the browser before sending it to the print dialog.
   * **Page.js Link**: [Page.js](https://github.com/visionmedia/page.js)
   * **jQuery Pagination Plugin**: [jQuery Pagination Plugin](https://github.com/rossmark/page)
2. **Puppeteer (for PDF Generation)**: Puppeteer is a headless Chrome Node.js library that can be used for rendering complex HTML into a PDF with pagination. It's more of a backend solution, but it works extremely well if you need server-side PDF generation.

Example usage of Puppeteer:

const puppeteer = require('puppeteer');

async function generatePDF() {

const browser = await puppeteer.launch();

const page = await browser.newPage();

await page.setContent('<html>Your HTML content for invoice</html>');

await page.pdf({ path: 'invoice.pdf', format: 'A4' });

await browser.close();

}

generatePDF();

* + **Link**: [Puppeteer Documentation](https://pptr.dev/)

**Summary**

* **jsPDF** is great for generating a PDF with pagination and custom formatting.
* **Print.js** is a lightweight solution for printing HTML content with easy styling.
* **Puppeteer** is ideal for generating PDFs server-side, especially if you want to handle complex rendering or need a headless browser.
* **Pagination Libraries** (like **Page.js**) can help manage the item display before printing.

These libraries can make your task easier depending on whether you're working with the client-side browser, generating a downloadable file (PDF), or managing a print preview.

To print the sales invoice using the jsPDF library, we need to make sure we are able to handle pagination and different paper sizes or orientations. Below is a detailed solution that shows how to print the invoice, handle pagination, and provide paper size and orientation options.

**Solution:**

1. **Define the click event for printing the invoice.**
2. **Create the PDF using jsPDF.**
3. **Handle pagination if the items exceed one page.**
4. **Provide options for paper size and orientation.**

**HTML Structure:**

<button id="printSalesInvoice">Print Sales Invoice</button>

**JavaScript Code:**

document.getElementById('printSalesInvoice').addEventListener('click', async () => {

alert('Print Sales Invoice');

// Collect the header data

const headerData = [

`Location: ${currentRec.LocaName}`,

`Ref. No.: ${currentRec.ReferDoc}`,

`OR Date: ${formatDate(currentRec.DateFrom,'MM/DD/YYYY')}`,

`Customer: ${currentRec.CustName}`,

`Remarks: ${currentRec.Remarks\_}`

];

console.log(headerData);

// Collect the items data

console.log(itemsDtl);

// Initialize jsPDF

const { jsPDF } = window.jspdf;

const doc = new jsPDF({ unit: 'mm', format: 'a4', orientation: 'portrait' }); // You can set format to 'letter', 'a4', or 'a3' and orientation to 'portrait' or 'landscape'

const pageMargin = 10; // Page margin (left, top, right, bottom)

const lineHeight = 8; // Line height for content

const itemLineHeight = 6; // Line height for items

const startY = 20; // Start Y position for the content

let currentY = startY;

// Add the header to the PDF

doc.setFontSize(12);

headerData.forEach((line, index) => {

doc.text(line, pageMargin, currentY);

currentY += lineHeight;

});

// Add the table headers

const tableHeaders = ['Qty', 'Stock No.', 'Bar Code', 'Item Description', 'Unit Price', 'Gross', 'Discount', 'Net'];

currentY += lineHeight; // Extra space before the table

doc.setFontSize(10);

doc.setFont("helvetica", "bold");

tableHeaders.forEach((header, index) => {

doc.text(header, pageMargin + (index \* 30), currentY); // Adjust x position for each column

});

currentY += lineHeight;

// Add the items to the table

doc.setFont("helvetica", "normal");

itemsDtl.forEach(item => {

if (currentY + itemLineHeight > doc.internal.pageSize.height - pageMargin) {

// If we reach the bottom of the page, add a new page

doc.addPage();

currentY = startY;

}

doc.text(item.Quantity.toFixed(0), pageMargin, currentY);

doc.text(item.UsersCde, pageMargin + 30, currentY);

doc.text(item.OtherCde, pageMargin + 60, currentY);

doc.text(item.Descript.substring(0, 30), pageMargin + 90, currentY); // Limiting item description to 30 chars

doc.text(formatter.format(item.ItemPrce), pageMargin + 150, currentY, { align: 'right' });

doc.text(formatter.format(item.Quantity \* item.ItemPrce), pageMargin + 180, currentY, { align: 'right' });

doc.text(formatter.format(item.Quantity \* (item.ItemPrce - item.Amount\_\_)), pageMargin + 210, currentY, { align: 'right' });

doc.text(formatter.format(item.Quantity \* item.Amount\_\_), pageMargin + 240, currentY, { align: 'right' });

currentY += itemLineHeight;

});

// Add totals at the bottom of the page

const totals = {

totalQty: itemsDtl.reduce((sum, item) => sum + item.Quantity, 0),

totalPrice: itemsDtl.reduce((sum, item) => sum + item.Quantity \* item.ItemPrce, 0),

totalDiscount: itemsDtl.reduce((sum, item) => sum + item.Quantity \* (item.ItemPrce - item.Amount\_\_), 0),

totalAmount: itemsDtl.reduce((sum, item) => sum + item.Quantity \* item.Amount\_\_, 0)

};

doc.setFontSize(12);

doc.text(`Totals:`, pageMargin, currentY);

doc.text(totals.totalQty.toFixed(0), pageMargin + 30, currentY);

doc.text(formatter.format(totals.totalPrice), pageMargin + 150, currentY, { align: 'right' });

doc.text(formatter.format(totals.totalDiscount), pageMargin + 180, currentY, { align: 'right' });

doc.text(formatter.format(totals.totalAmount), pageMargin + 240, currentY, { align: 'right' });

// Save or print the document

doc.save('sales\_invoice.pdf');

});

**Explanation:**

1. **Header Section:** The header is printed with details like location, reference number, date, customer, and remarks.
2. **Table Section:** The invoice items are listed in a table format. Each item includes columns such as quantity, stock number, item description, unit price, gross, discount, and net.
3. **Pagination:** If the items do not fit on one page, the code will add a new page when the current Y position exceeds the bottom margin of the page (currentY + itemLineHeight > doc.internal.pageSize.height - pageMargin).
4. **Totals Section:** After listing the items, the totals are displayed at the bottom of the invoice.
5. **Paper Size and Orientation:** The jsPDF document is initialized with an A4 paper size and portrait orientation (new jsPDF({ unit: 'mm', format: 'a4', orientation: 'portrait' })). You can change the size and orientation as needed.

**Customization:**

* **Paper size and orientation:** You can change the paper size and orientation when initializing the jsPDF instance:
  + Paper sizes: 'a4', 'letter', 'a3', etc.
  + Orientation: 'portrait' or 'landscape'.
* **Font settings:** Adjust the font size and type as needed using doc.setFontSize(size) and doc.setFont(fontFamily, fontStyle).

This approach should give you a fully functional sales invoice that can handle pagination and provide customizable paper size and orientation.

CDN Link

<script src="https://cdnjs.cloudflare.com/ajax/libs/jspdf/2.5.1/jspdf.umd.min.js"></script>

CONTROL BATCH OF DATA

Yes, you can control the data in batches (e.g., 500 records at a time) by implementing pagination on both the frontend and the backend. This will allow you to load smaller chunks of data to improve performance, and provide a user experience where they can request the next batch of data, as seen in apps that show progress (e.g., "500 of 7500").

**How to Implement Pagination:**

**1. Frontend Changes (Pagination Controls)**

In the frontend, you'll need to add controls to handle the batch size and allow users to navigate between batches (pages). You can display the current batch number and the total count of records, and fetch the next set of records when the user clicks a "Next" button.

Here’s how to modify your frontend:

* Add pagination parameters (like page and limit) to your query.
* Fetch data for the appropriate page and limit on each request.

Example:

// Keep track of current page and batch size

let currentPage = 1;

const batchSize = 500;

const loadBatch = async () => {

try {

// Build query parameters

const url = new URL('http://localhost:3000/sales/SalesDtlLst');

const params = new URLSearchParams();

if (cCtrlNum\_) params.append('CtrlNum\_', cCtrlNum\_);

params.append('page', currentPage); // Add page parameter

params.append('limit', batchSize); // Add limit parameter

// Send request with query parameters

const response = await fetch(`${url}?${params.toString()}`);

if (!response.ok) {

throw new Error('Network response was not ok');

}

const itemsDtl = await response.json(); // Store the batch of records

salesDtlCounter.innerHTML = `${itemsDtl.length} Records (Page ${currentPage})`; // Show current page

updateItemTable(itemsDtl); // Render the batch of items using <tr>

} catch (error) {

console.error('Fetch error:', error);

}

};

// Load the first batch of data

loadBatch();

// Pagination button click

const nextButton = document.getElementById('nextButton');

nextButton.addEventListener('click', () => {

currentPage++;

loadBatch();

});

In this code, the currentPage variable determines which set of records you want to fetch. The limit is set to 500 records per page, and each time the "Next" button is clicked, the currentPage is incremented and the new batch of data is fetched.

**2. Backend Changes (Implement Pagination in SQL)**

On the backend, you need to modify the SQL query to return a specific page of records, based on the page and limit parameters received from the frontend.

Here’s how you can adjust your backend to support pagination:

1. Modify your SQL query to include LIMIT and OFFSET based on the page and limit values.
2. Calculate the offset based on the current page ((page - 1) \* limit).

Example:

const SalesDtlLst = async (req, res) => {

const cCtrlNum\_ = req.query.CtrlNum\_;

const page = parseInt(req.query.page) || 1; // Default to page 1

const limit = parseInt(req.query.limit) || 500; // Default to 500 records per page

const offset = (page - 1) \* limit; // Calculate the offset

let cSql = `SELECT

SALESDTL.RecordId,

SALESDTL.CtrlNum\_,

SALESDTL.ItemCode,

ITEMLIST.UsersCde,

ITEMLIST.OtherCde,

ITEMLIST.Descript,

SALESDTL.Quantity,

SALESDTL.ItemPrce,

SALESDTL.DiscRate,

SALESDTL.Amount\_\_,

SALESDTL.LandCost

FROM SALESREC, SALESDTL, ITEMLIST

WHERE SALESREC.CtrlNum\_ = SALESDTL.CtrlNum\_

AND SALESDTL.ItemCode = ITEMLIST.ItemCode

AND 1=1

`;

// Add any filters

if (cCtrlNum\_) {

cSql += " AND SALESREC.CtrlNum\_ LIKE @cCtrlNum\_";

}

cSql += ` ORDER BY SALESDTL.RecordId

LIMIT ${limit} OFFSET ${offset}`; // Add LIMIT and OFFSET for pagination

const params = {};

if (cCtrlNum\_) {

params.cCtrlNum\_ = `%${cCtrlNum\_}%`;

}

try {

// Execute query with pagination

const result = await queryDatabase(cSql, params);

res.json(result); // Return the paginated results

} catch (err) {

console.error('Database query error:', err.message); // Log the error message

res.status(500).send('Error fetching sales data');

}

};

**Key Points:**

* The page and limit query parameters control the pagination behavior.
* The OFFSET is calculated based on the current page: OFFSET = (page - 1) \* limit.
* LIMIT specifies how many records to return per request.

**Example Flow:**

* On the first page, the backend fetches records 1-500.
* On the second page, the backend fetches records 501-1000.
* When the user clicks the "Next" button, the frontend increments the currentPage, and a new batch of data is fetched.

With this approach, you will have efficient handling of large datasets, loading data in smaller, manageable chunks.