ChatGPT

Visual Page-Builder for Dynamic Subproduct Pages

The best way to satisfy these requirements is to use an open-source drag-and-drop page builder like **GrapesJS**. GrapesJS is a multi-purpose web builder framework designed for "building HTML templates without any knowledge of coding" [1]. It provides built-in drag-and-drop of components (text, images, columns, etc.), a style manager for resizing/styling elements, and a storage system to save layouts. Crucially, GrapesJS is extensible via plugins or custom blocks, so you can add features (e.g. advanced tables) as needed [2]. Because it is fully client-side and open-source, it can be self-hosted with your stack and integrated into your React/Tailwind CMS.

How GrapesJS Meets the Requirements

- **Free-form Layout (non-block elements):** GrapesJS allows arbitrary positioning of components on a "canvas" and supports nested containers (columns, etc.). For example, you can drag an image into a column container and then place text in an adjacent column, achieving a two-column layout. The editor's *Style Manager* lets you adjust width/height or use drag-handles to resize components. Under the hood, GrapesJS components can be arranged flexibly (not forced to full-width), satisfying the "no element should be block" requirement [2] [3].
- **Image Upload & Manipulation:** GrapesJS includes an **Asset Manager** for images. Users can click or drag-drop to upload images from their device (which you configure with an upload endpoint). Once on the canvas, images can be resized and moved. The built-in "resize" control lets users drag corners to change dimensions [3], and you can also set exact width/height via the style editor. By configuring `assetManager.upload` to point to your Express file-upload API, you enable server-side image saving [4].
- **Advanced Tables:** You can add table elements either via a custom **Table** block or a plugin. For example, you can create a custom block that inserts an HTML `<table>` and then use the Style Manager to adjust cell padding/background. For even richer features, community plugins like `grapesjs-table` provide configurable rows/columns, cell coloring, and border toggling. GrapesJS's plugin system makes it easy to integrate such tools [2].
- **WYSIWYG Content:** GrapesJS includes a Rich Text component for editing paragraphs. If you need more advanced text editing (or want TipTap specifically), GrapesJS allows replacing the default RTE with a custom one (see its "Replace Rich Text Editor" guide). However, for most product pages, the built-in text block with formatting toolbar (bold, lists, etc.) suffices.
- **Dynamic Saving & Rendering:** GrapesJS's StorageManager supports saving the page structure as JSON (components + styles). You can configure it to use **remote storage** via REST endpoints [5]. When the admin clicks "Save", GrapesJS can `POST` the project JSON to your Express API. On load, it can fetch the JSON back into the editor. The resulting saved data can then be rendered on the front-end: for instance, fetch the JSON from your server and use `editor.getHtml()` / `editor.getCss()` [6] to obtain the final HTML/CSS for display. Alternatively, you can directly store the final HTML/CSS in your database for fast rendering.
- **Integration and Self-Hosting:** Because GrapesJS is purely client-side (no SaaS), you can bundle it into your React admin panel (styled with Tailwind/shadcn) and serve everything from your own

infrastructure. The code below shows integrating GrapesJS in a React+TypeScript component, configuring plugins, storage, etc.

Implementation Guide

Below is a step-by-step implementation plan, including key code snippets. We assume a React 18 + TypeScript admin panel, using Wouter for routing. This uses Tailwind for basic styling and follows your stack preferences.

1. Install Dependencies

Add GrapesJS and related presets to your React app:

```
npm install grapesjs grapesjs-preset-webpage @types/grapesjs
```

- `grapesjs` – core builder framework.
- `grapesjs-preset-webpage` – provides common blocks (columns, text, images, videos, etc.).
- You can also add any table plugin later (e.g. `grapesjs-table`).

Also ensure you have Tailwind, React, etc., already set up per your stack.

2. Create the Editor Component in React

In your admin panel, create a component (e.g. `PageBuilder.tsx`) where GrapesJS will render. This component mounts a `<div>` container and initializes the editor in `useEffect`. For example:

```
import React, { useRef, useEffect } from 'react';
import grapesjs from 'grapesjs';
import 'grapesjs/dist/css/grapes.min.css';
import 'grapesjs-preset-webpage/dist/grapesjs-preset-webpage.min.css';

interface PageBuilderProps {
  pageId: string;  // e.g. product ID or slug
}

const PageBuilder: React.FC<PageBuilderProps> = ({ pageId }) => {
  const editorRef = useRef<HTMLDivElement>(null);

  useEffect(() => {
    if (!editorRef.current) return;
    // Initialize GrapesJS editor
    const editor = grapesjs.init({
      container: editorRef.current,
      fromElement: false,
      height: '100%',
```

```
      width: '100%',
      plugins: ['gjs-preset-webpage'],
      pluginsOpts: {
        'gjs-preset-webpage': {
          // Only include needed blocks: text, image, columns, etc.
          blocks: ['column1', 'column2', 'column3', 'text', 'link', 'image',
'video']
        }
      },
      // Configure asset manager for image upload
      assetManager: {
        upload: '/api/upload/image',  // your Express endpoint for image upload
        uploadName: 'files',
        autoAdd: true
      },
      // Configure storage manager for saving/loading page designs
      storageManager: {
        type: 'remote',
        autosave: true,
        autoload: true,
        stepsBeforeSave: 1,
        options: {
          remote: {
            // REST endpoints for GET/PUT (or POST) to save the page JSON
            urlLoad: `/api/pages/${pageId}`,
            urlStore: `/api/pages/${pageId}`,
            fetchOptions: (opts: any) =>
              opts.method === 'POST' ? { method: 'PUT' } : {},
            // Wrap/unwrap JSON so we store { data: {...} }
            onStore: data => ({ id: pageId, data }),
            onLoad: result => result.data
          }
        }
      }
    });

    return () => editor.destroy(); // cleanup on unmount
  }, [pageId]);

  return (
    <div className="w-full h-full bg-white border rounded">
      <div ref={editorRef} className="h-full" />
    </div>
  );
};

export default PageBuilder;
```

Explanation: - We import GrapesJS and its CSS. - The `#ref` container (`editorRef`) is where the editor UI will appear. - `gjs-preset-webpage` plugin adds basic building blocks (columns, images, text). - **Asset Manager:** `upload: '/api/upload/image'` tells GrapesJS to send uploaded images to that endpoint. The default UI will allow dragging files onto the asset modal. - **Storage Manager:** Setting `type: 'remote'` makes GrapesJS call your endpoints to save/load. Here we use PUT on `/api/pages/:id`. The `onStore/onLoad` handlers adapt your JSON shape. With `autosave: true`, changes are saved automatically (or you can disable that and add a manual Save button that calls `editor.store()`). - Tailwind classes (e.g. `"bg-white border rounded"`) can style the container.

3. Backend API Endpoints (Express + Drizzle)

On the server side, create API routes to handle saving and loading the page design JSON. Example using Express and Drizzle ORM:

```
// server.ts (or app.ts)
import express from 'express';
import session from 'express-session';
import connectPgSimple from 'connect-pg-simple';
import passport from 'passport';
import bcrypt from 'bcrypt';
import { drizzle } from 'drizzle-orm';
import { pgTable, serial, jsonb } from 'drizzle-orm/pg-core';

const app = express();
app.use(express.json());

// Database setup (Neon/Postgres via Drizzle)
const db = drizzle(/* your PG connection */);

// Define a table to store pages
const pages = pgTable('pages', {
  id: serial('id').primaryKey(),
  slug: serial('slug'),     // or use text if slug not numeric
  data: jsonb('data').notNull(),  // stores GrapesJS project data JSON
});

// GET page data
app.get('/api/pages/:id', async (req, res) => {
  const id = Number(req.params.id);
  const page = await db.select().from(pages).where(pages.id.eq(id)).all();
  if (!page[0]) return res.status(404).send({ error: 'Not found' });
  res.json({ id, data: page[0].data });
});

// PUT (or POST) to save/update page data
app.put('/api/pages/:id', async (req, res) => {
```

```
  const id = Number(req.params.id);
  const { data } = req.body;  // GrapesJS will send { id, data }
  if (!data) return res.status(400).send({ error: 'No data provided' });

  // Insert or update
  await db.insert(pages).values({ id, data }).onConflictDoUpdate({
    target: [pages.id],
    set: { data: data }
  });
  res.sendStatus(200);
});

// Image upload endpoint used by Asset Manager
app.post('/api/upload/image', /* use multer or similar to handle multipart */
(req, res) => {
  // Save file to server or S3, then return { data: [ { src: 'url-of-uploaded-
image' } ] }
  // GrapesJS expects JSON { data: [url, ...] }
});

// ... (other routes, auth setup, etc.)
```

Explanation: - We use an Express route `/api/pages/:id` for GET and PUT. This matches our GrapesJS `urlLoad` / `urlStore`. - When GrapesJS loads a page, it expects JSON with a `data` field containing the editor state [5]. We comply by returning `{ data: <projectJson> }`. - On save, GrapesJS will `PUT` JSON `{ id, data }`. We extract `data` and store it (using Drizzle ORM in this example). - The image upload endpoint should accept file uploads (e.g. using `multer`) and respond with JSON `{ data: ['https://yourcdn/path/image.jpg'] }`, so GrapesJS can add the uploaded image to its asset list [4]. - **Security:** Protect these routes with authentication (Passport + sessions) so only admins can save designs.

4. Advanced Table Support

To allow table creation with fine-grained control, add a block or plugin. For example, you can register a custom "Table" block in GrapesJS:

```
useEffect(() => {
  // ... after initializing editor
  editor.BlockManager.add('custom-table', {
    label: 'Table',
    category: 'Basic',
    content: {
      type: 'table',
      components: [
        {
          type: 'table-row',
```

```
          components: [{ type: 'table-cell', components: 'Table cell text' }]
        }
      ]
    }
  });
}, [pageId]);
```

This will let users drag a 1×1 table. They can then use the style panel to add rows/columns (by editing HTML or with plugins). For full features (merged cells, styling, borders), use a ready plugin like `grapesjs-table` or the GrapesJS Studio SDK's table plugin. These plugins let users specify number of rows/columns, cell background colors, hide lines, etc. (They leverage GrapesJS's custom component and style system.)

5. Rendering the Saved Page on Frontend

Once designs are saved, your public product page component can fetch and render them dynamically. In React with Wouter, you might have a route like `/product/:id`. In that component, fetch the saved page data and inject it:

```
import React, { useEffect, useState } from 'react';
import { useLocation } from 'wouter';

const ProductPage: React.FC = () => {
  const [_, params] = useLocation();
  const pageId = params.id;
  const [htmlContent, setHtmlContent] = useState<string>('');

  useEffect(() => {
    fetch(`/api/pages/${pageId}`)
      .then(res => res.json())
      .then(({ data }) => {
        // data is the GrapesJS project JSON.
        // We can extract HTML and CSS:
        const parser = new DOMParser();
        // GrapesJS returns JSON of components; we can reconstruct HTML:
        // Simplest: send HTML/CSS from server. Or use GrapesJS in JS:
        // For demo, assume server stored HTML/CSS too:
        const { html, css } = data;
        setHtmlContent(`<style>${css}</style>${html}`);
      });
  }, [pageId]);

  return <div dangerouslySetInnerHTML={{ __html: htmlContent }} />;
};
```

Alternatively, you can let the server side (Express) compile `editor.getHtml()` and `editor.getCss()` and store them in the database along with the JSON, so the frontend just injects raw HTML/CSS. GrapesJS

provides `editor.getHtml()` and `editor.getCss()` to retrieve the current layout's HTML and CSS [6]. This ensures the live site always shows the latest design without rebuilding the React app.

Complete Integration Example

Putting it all together:

- **Frontend (Admin Panel):** A React page with the `PageBuilder` component above. Style the container with Tailwind (e.g. `className="h-screen px-4"`). Provide a UI (button) to manually trigger save (`editor.store()`) if not autosaving.
- **Backend (CMS/API):** Express routes for `/api/pages/:id` and `/api/upload/image` as shown. Use Drizzle to persist JSON. Ensure CORS and JSON handling are enabled.
- **Authentication:** Use `passport-local` and `express-session` so only logged-in admins can access the editor routes. (This follows your chosen stack.)
- **On Production:** Host the React app and Express server on your infrastructure (self-hosted). GrapesJS runs entirely in the browser, so it meets the "self hosted" requirement with no external SaaS.

By using GrapesJS with this setup, you get a fully flexible "Word-like" editor for any page layout. Users can place text, images, tables, etc., side by side, resize them, and style them visually. All content and design is saved as structured JSON (and/or HTML/CSS), which your React front end then loads and renders for visitors. This satisfies **all** listed advanced requirements (free layout, image upload/resize, complex tables) without manual coding of each page.

Sources: The solution above leverages the open-source GrapesJS framework [1] [2], its Asset Manager for image upload [4], the Canvas module for resizing components [3], and its StorageManager for remote saving/loading of page data [5]. Output HTML/CSS is obtained via `editor.getHtml()`/`getCss()` [6] so pages can be rendered dynamically. This approach fully implements a custom page builder in your React+Express stack.

---

[1] [2] Integration of grapesjs react - effectively and efficiently
https://esketchers.com/integrating-grapesjs-react/

[3] Canvas | GrapesJS
https://grapesjs.com/docs/modules/Canvas.html

[4] Asset Manager | GrapesJS
https://grapesjs.com/docs/modules/Assets.html

[5] Storage Manager | GrapesJS
https://grapesjs.com/docs/modules/Storage.html

[6] django - Get HTML and CSS from grapes.js newsletter editor in Javascript - Stack Overflow
https://stackoverflow.com/questions/53984919/get-html-and-css-from-grapes-js-newsletter-editor-in-javascript