ChatGPT

Plan for a Cannabis Product Ordering Dashboard Deployment

Architecture Overview

The solution will use a **static web application architecture enhanced with dynamic data loading** to minimize maintenance. The core idea is to treat the product menu as data (in Excel form) that can be updated easily, while the application code (HTML/JS/CSS) remains static and version-controlled. Key components of the architecture:

- **Front-End (Client-Side)**: A responsive HTML/CSS/JavaScript application that runs in the browser. This front-end will dynamically load and display product data from the Excel file, and provide an order form for clients. Using a static front-end (no continually running server for page rendering) improves reliability and scalability (static files can be served quickly to many users) and simplifies deployment.

- **Data Source (Excel "Live Menu")**: The product menu is maintained as an Excel spreadsheet (with a fixed, known structure). This Excel file resides in the GitHub repository (e.g., in a `/data` folder). Whenever the Excel is updated (new file uploaded to the repo), the site will reflect the changes. We will ensure the **file name and format remain consistent** for ease of automation (e.g., always `LiveMenu.xlsx` in a known location).

- **Data Parsing**: On each deployment (or on-demand in the browser), the Excel file is parsed and converted into a data structure the front-end can use (like JSON or JavaScript objects). For a low-maintenance approach, we recommend performing this **parsing on the client side** using a lightweight JS library. Modern libraries (for example, `read-excel-file` or SheetJS) can **read small to medium** `.xlsx` **files directly in the browser or in Node and output JSON** [1] . By parsing in the browser, we avoid maintaining server infrastructure for data processing. Alternatively, a build-step script (using Node.js or Python in a CI workflow) could convert the Excel to JSON automatically during deployment – this reduces client-side work, at the cost of a slightly more complex build pipeline. Both approaches eliminate the need to manually edit code when the Excel data changes.

- **Back-End (Order Submission Handling)**: To handle form submissions (product orders), a small server-side component is needed, since static sites alone cannot send emails or securely store data. The simplest low-maintenance solution is to use a **serverless function or a lightweight server script** on the hosting environment to process orders. Given IONOS's capabilities, a practical choice is a small **PHP script** (or a Python/Node script if the hosting supports it) that runs when the form is submitted:

- The script will accept the order details (likely via an HTTP POST from the form).
- It will format the order information (client details + selected products and quantities) into an email message.

- It will send an email notification to the designated recipient (e.g., the business's ordering inbox) via SMTP or an email API.
- Optionally, it will log the order to a server-side file or database for record-keeping.

Using a simple server-side script for emailing ensures that email server credentials/API keys remain secret (not exposed in client-side code) and avoids cross-origin issues. This approach is considered **best for control and security** in production, as noted by developers [2] [3] – it gives full control over email format and avoids third-party dependencies, unlike purely client-side solutions.

- **Auto-Refresh Mechanism**: Whenever a new Excel file is uploaded to GitHub, the continuous deployment pipeline will redeploy the site, thereby **automatically pulling in the updated Excel data**. The front-end will either fetch and parse the new Excel at runtime, or use a freshly generated data file (if using a build-step conversion). This means *simply updating the Excel and pushing to GitHub will refresh the dashboard content*. As a fallback, we can include a manual "Refresh Data" trigger on the page – for example, a button that forces reloading the data (useful if the page has been open for a long time or if an admin wants to verify the latest menu is loaded without waiting for a full redeploy). In practice, with automatic deployment, such manual refresh is rarely needed, but it can be added for reassurance.

- **Separation of Concerns**: The architecture separates content from presentation:

- *Content* (product info) lives in the Excel file (and its parsed JSON form).
- *Presentation and Application Logic* live in the static HTML/JS.
- *Order transmission* is handled by a small back-end utility.

This separation makes maintenance easier: non-technical staff can update the menu via Excel, and developers only update the code for structural or feature changes.

Technologies and Tools

This section details the recommended technologies for each part of the system, chosen for ease of use, compatibility with IONOS hosting, and low maintenance overhead:

- **Front-End Framework:** Use simple **HTML5, CSS3, and JavaScript** for the client-facing app. You can enhance maintainability by using a component-based front-end framework *if needed* (such as React or Vue), but given the relatively straightforward functionality (displaying a list and a form), a lightweight approach using plain JS or a minimal library (like jQuery or Vanilla JS) is sufficient. This avoids the complexity of a heavy framework and builds. For styling and responsiveness, consider a CSS framework like **Bootstrap** or **Tailwind CSS**, which will allow quick responsive design without needing to write extensive CSS. The goal is a **responsive UI** that works on desktop, tablet, and mobile (e.g., using Bootstrap's grid or Tailwind's utility classes to ensure product tables/cards wrap appropriately on smaller screens).

- **Excel Parsing:** To read the Excel file in the browser, incorporate a JS library such as **SheetJS (xlsx)** or `read-excel-file`. These libraries can fetch an Excel file (either via an AJAX call or by referencing a file URL) and parse its contents into a JavaScript object/array. For example, `read-excel-file` can load an Excel given a URL or file blob and produce an array of rows that you can transform into JSON

objects (with keys like Product Name, Category, Price, etc.) [4] [5]. This will be used at runtime to populate the dashboard. If a build-step approach is chosen instead, use **Node.js** with a library like `xlsx` (SheetJS's Node version) or **Python** with `pandas/openpyxl` to parse the Excel. In a Node build script, for instance, `xlsx` can read the Excel and you can output a `menu.json` file that gets deployed with the site; in Python, `pandas` can convert sheets to JSON or CSV which you then include. The best-fit for low maintenance is the *client-side parse* for simplicity (no custom build logic or servers needed), assuming the Excel remains reasonably sized.

- **Data Format and Structure:** Based on the provided Excel structure, it contains product listings with fields like *Product Name*, *Category*, *THC %*, *Price*, *Case Size*, etc., and possibly a second sheet with extended descriptions. The solution will merge or use these fields to present the menu. For example:

- You might create a JSON structure like:

```
{
  "Tinctures": [
      { "name": "Jive Daytime Tincture", "THC": "500mg", "price": 22.5,
"description": "Start your day with ...", "caseSize": 20, ... },
      ...
  ],
  "Pre-rolls": [ ... ],
  ...
}
```

where each category (from the Category column) becomes a key, and the value is an array of products in that category with their details. This will make it easy to iterate and display grouped by category on the page. The parsing code (either in JS or during build) will transform the raw rows into a structured JSON like above.

- **Front-End Dynamic Behavior:** Use JavaScript in the page to:

- Fetch and parse the Excel (or load the pre-generated JSON).
- Dynamically inject product listings into the HTML (e.g., using DOM manipulation or template literals). For maintainability, you could use a simple templating approach or a micro-library, but plain JS loops are fine given the data structure.
- Implement the **order form logic**: allow the user to input a quantity for each product (e.g., an `<input type="number">` next to each product, or "+/-" buttons). You might also include an "Add to Order" button for each product or simply have a single form that lists all products with quantity fields.
- Tally the selected items (optional: before submission, create a summary of items ordered).
- Collect the client's contact info from form fields (name, dispensary, email).

- On clicking **Submit Order**, gather all this data (perhaps constructing an order object or summarizing in text).

- **Form Submission Method:** The client-side will send the order data to the back-end for processing. There are two primary ways to do this:

- **Form POST Request**: Use a traditional HTML `<form>` element that posts to a URL (e.g., an `order.php` or `order-handler.php` on the server). The form would be populated with hidden inputs (for the order details in text form) or better, just rely on named inputs for each item quantity and the contact fields. On submit, the PHP script iterates through the inputs to build the order. This method is simple and leverages built-in form behavior.

- **AJAX (Fetch API)**: Use JavaScript `fetch()` to send the order as a JSON payload to an API endpoint (for example, a Node or Python-based endpoint, or a PHP script that reads JSON from `php://input`). This can provide a smoother user experience (the page can display a confirmation message without full reload). If using IONOS Deploy Now with a static site + PHP, approach #1 (form POST) might be easiest to configure, but #2 is also possible if the script is set up to handle JSON requests.

- **Back-End Technologies**: We recommend using **PHP** for the back-end on IONOS, as their Deploy Now service explicitly supports PHP deployments [6] and it doesn't require managing a separate server process. A simple PHP script can use `mail()` or an email library (like PHPMailer) to send emails. If the project maintainers prefer Node.js or Python, those could be used *but* note that IONOS's static deployment doesn't support running Node servers at runtime [7]. In case of Node/Python, you would need an IONOS **Managed Server** or a separate service (increasing complexity). For low maintenance, PHP is a good choice here since it can run on cheap shared hosting and is triggered only on form submission. The PHP script can be kept very small (just reading inputs, sending mail, and writing a log).

- **Email Notification Service:** Set up an email system to send the order details to the business (and optionally a confirmation to the client). Two options:

- **Use SMTP with a Server-Side Script**: Configure PHP (or Python) to use an SMTP server (e.g., Gmail SMTP, IONOS's email SMTP if included, or a dedicated service like SendGrid SMTP). For example, using PHPMailer in PHP, you can connect via SMTP with authentication and send a nicely formatted email (HTML email listing the order items and client info). This is secure and reliable, as credentials are kept on the server side and not exposed. Make sure to store these credentials securely (in a config file or environment variable if possible).
- **Use an Email API**: Alternatively, use a cloud email API (SendGrid, Mailgun, etc.). This could be done server-side (preferred, to keep API keys secret) or client-side. Client-side API calls are *not recommended in production* due to security and CORS concerns [8], so we lean towards server-side. If using an API, the server script will make a HTTP POST to the service with the email data instead of SMTP.

The email should include all relevant order info: product names, quantities, and the client's contact details. Using a templated format (e.g., an HTML table listing each item) makes it easy for the business to read. Also, set a clear subject line (like "New Order from [Dispensary Name]") for filtering. **Tip:** It's good practice

to also send a confirmation email to the client's email address, confirming their order was received, as this enhances user experience.

- **Order Logging:** For a light, secure logging method, consider:
- **Server-Side Log File**: The back-end script can append each order (in CSV or JSON format) to a file located in a non-public directory on the server (to protect sensitive data). For example, a CSV line might include timestamp, client name, client email, and a summary of order items. This file serves as a simple database of orders. Ensure proper file permissions and regularly rotate or back up the log.
- **Email Logs**: As a simple fallback, you can rely on emails as the record (each received order email is saved). To improve this, you might BCC an internal address on every order email that is specifically meant for logging (e.g., an archive mailbox).
- **Cloud Spreadsheet or Database**: As an optional enhancement, the script could also write the order into a Google Sheets via API or a small MySQL database if one is available on IONOS. However, these add complexity. For a production-ready but low-volume scenario, a flat file log or just the email trail is sufficient.

Regardless of method, **do not store any sensitive personal data unnecessarily**. Only store what's needed for order fulfillment (in this case, the contact info and order details). Ensure the log file (if used) is not web-accessible.

- **Security and Validation:** Apply basic security best practices:
- **Input Validation**: On the client side, use HTML5 form validation (e.g., `required` attributes, `type="email"` for email format) and/or additional JS checks to ensure the form is filled out correctly (no negative quantities, email is present, etc.). On the server side, **always validate and sanitize inputs** as well, since client-side checks can be bypassed. For example, if using PHP, use functions like `filter_var($email, FILTER_VALIDATE_EMAIL)` for email, and sanitize text fields to prevent header injection or malicious content.
- **Spam Protection**: Public forms can attract spam bots. Implement a simple anti-spam measure such as a honeypot field (an invisible field that real users won't fill, but bots might) or Google reCAPTCHA v2/v3. This will prevent abuse of the order form (which could otherwise be used to send spam emails to you).
- **HTTPS**: Ensure the deployed site uses HTTPS, which IONOS Deploy Now will provide with TLS/SSL automatically for custom domains [9] [10]. This is important to protect the form data in transit (especially the client's email/contact).

- **Authentication (if needed)**: Currently, the site is client-facing and presumably open to any dispensary clients. If in the future you want to restrict it (so only certain clients access the menu), consider simple auth measures (even a basic password or code, or more robust login). But for now, it's likely a public ordering page.

- **Maintainability Considerations**: The chosen technologies emphasize low maintenance:

- No running application server to monitor (if using static + PHP, the PHP only runs on request).
- Automatic deployment means no manual file transfers to hosting.
- Updates to the menu don't require code changes, just replacing the Excel file.
- The solution leverages widely used tech (HTML/JS, a bit of PHP) which most web developers can readily understand, ensuring long-term support.

- All code and data are in GitHub, providing version control. Changes to the menu or code can be tracked, and if an issue arises after an update, it's easy to revert to a previous version of the site or menu file.

Deployment Workflow (Step-by-Step)

This section outlines how to implement and deploy the dashboard to production, using GitHub for code management and IONOS for hosting. We assume you have an IONOS hosting account (with Deploy Now or a similar feature) and a GitHub repository ready for the project.

1. Set Up the GitHub Repository: - Organize the repository with a clear structure. For example: - `/index.html`, `/styles.css`, `/app.js` (for main HTML, CSS, JS files) - `/data/LiveMenu.xlsx` (the Excel menu file, possibly with a generic name like this) - `/scripts/` (for any build or parsing scripts, if used) - `/server/order.php` (the server-side form handler, if using PHP; this could also live at root or a `/api` subfolder). - Any other assets (images, etc.) and README documentation. - Commit and push this structure to GitHub. This will be the codebase that IONOS will deploy. Using GitHub ensures all changes (including new Excel files) are tracked.

2. Connect GitHub to IONOS (Continuous Deployment): - Log in to the IONOS hosting control panel and locate the **Deploy Now** (or similar) feature, which integrates with GitHub. IONOS allows you to *connect a GitHub repository and automate deployments* [11] . - In Deploy Now, create a new project and select the GitHub repository and branch (likely `main` or `master`) for deployment. During setup, specify that this is a static site or a PHP project (since we have PHP for form handling). IONOS will detect common static site frameworks or provide an option to configure build commands. - If our project is plain HTML/JS + optional PHP, there is **no build command needed** (we're not using webpack or static site generators in this plan). We can indicate the output directory as the repository root (since our `index.html` is at root, for example). - If you chose to use a build script to generate JSON from Excel, you might need to set up a build step. For instance, you could add a custom GitHub Actions workflow or use a package.json script. In IONOS Deploy Now, you can specify something like `npm install` and `npm run build` as build steps. The build script (written in Node) would read `data/LiveMenu.xlsx` and output `data/menu.json`, which is then included in the deployed files. Ensure the action is configured to add that file to the publish directory. - Once configured, trigger the first deployment. The Deploy Now service will pull the code from GitHub, run any build steps, and deploy to a provided URL. You should get a temporary IONOS URL (and later you can attach your custom domain). Verify that the site loads at this URL.

3. Configure Environment (if needed): - If your back-end email script requires configuration (SMTP credentials, API keys, etc.), now is the time to set those up. IONOS's Deploy Now allows setting environment variables through the interface, or you might need to include a config file in the project (if using PHP, perhaps a separate config PHP or a section at the top of the script). For example, define constants for SMTP server, username, password, recipient email, etc. **Do not hardcode sensitive passwords** directly in version control; use environment variables or keep them in a secure file not committed to Git. - In a PHP environment on IONOS, you may also need to enable certain extensions (like enabling PHPMailer's requirements or ensuring the mail function is not disabled). In most shared hosting, the `mail()` function works out of the box, but if using SMTP, ensure the server can reach that SMTP host (it should, typically on standard ports).

4. Testing the Deployment: - After deployment, test the site's functionality: - Does the product menu display correctly? (The front-end should load the Excel or JSON. Check the browser console for any errors in fetching or parsing. If issues, you may need to adjust path references or parsing logic). - Test responsiveness by viewing on a mobile device or using browser dev tools. - Fill in the form with sample data and submit an order. Verify that: - A confirmation message appears on the page (if you implemented one). - The email is received at the intended inbox. Check spam folder as well and mark it safe if needed. - The content of the email is correctly formatted (all items and quantities present, contact info included). - If applicable, check that the order was logged (e.g., see if the log file updated or the database entry was created). - Fix any issues discovered (iterate by updating the code on GitHub and pushing; the continuous deployment will redeploy the updates).

5. Setting Up the Custom Domain (if applicable): - If you have a domain (like `yourcompany.com` or a subdomain for the menu), point it to the IONOS deployment. In Deploy Now, you can add a custom domain in the settings. You'll likely need to create a CNAME or A record in your DNS pointing to IONOS (their interface will provide details). Once connected, IONOS will provision an SSL certificate automatically for HTTPS [9]. - Confirm the site is accessible via the custom domain and still secure (https).

6. Updating the Excel Menu (Content Updates): - When the product list changes (new products, price changes, etc.), the process is simple: - Update your Excel file (make sure to follow the same structure: e.g., same sheet names, column layout, etc., so the parsing logic continues to work). For example, if the new file is provided by someone, just ensure it's saved with the same name and format. - In the GitHub repo, replace the old Excel file with the new one. If using GitHub's web interface, you can upload the file to the `data/` folder and commit the change (the file name being identical will overwrite the old version; that's fine). If using Git locally, put the new file and `git push`. - The push triggers the IONOS deployment pipeline. The new Excel will be deployed. Depending on implementation: - If client-side parsing: the next user who loads or refreshes the page will get the new Excel data. (Be mindful of browser caching: you might configure the server to send proper cache headers or always fetch with a cache-busting query parameter to ensure the latest file is fetched.) - If build-time JSON: the GitHub Action or build step will generate a new `menu.json` from the Excel and deploy it. The front-end will then fetch this updated JSON for all new page loads. - **Single-click manual trigger**: In case you want a manual refresh trigger, you could integrate a small admin interface or simply instruct an admin user to click a "Refresh" button on the page which calls the data loading function again. However, given the continuous deployment, the content will update automatically on each Excel commit, so usually just reloading the page is enough. If an immediate refresh is needed (say, the page is displayed on a screen in a store and someone updated the Excel), clicking a refresh button in the UI could re-fetch the data without a full page reload. Implement this by having your data-loading JS function callable on demand (e.g., `loadMenuData()` that you can call again to refetch the Excel and update DOM).

7. Ongoing Maintenance: - With this setup, day-to-day updates are primarily **menu updates via Excel**. No code changes are required for content updates. Ensure everyone who will update the menu knows how to commit to GitHub (or have a developer set up an easier interface for them, if needed). - Periodically, update the dependencies (JS libraries, etc.) if using any, to keep up with security patches. Because the site is simple, dependency management is minimal (possibly just the Excel parsing library and maybe Bootstrap – both can be included via CDN for ease, or pinned in your repo). - Monitor the order emails/logs. If order volume grows, consider if the process still holds (Excel might become cumbersome with hundreds of products – at that point a database-driven solution might be considered; similarly, dozens of orders per day might warrant a more robust order management system). The plan as given is optimized for low-to-moderate scale with minimal overhead.

User Experience Design

Designing a smooth user experience is crucial for a client-facing ordering dashboard. Below is how the app will function for end users (dispensary clients placing orders):

- **Landing Page / Menu View:** When a user visits the dashboard, they are presented with a clean, branded interface (use the company's branding and possibly a header or logo). The product menu is displayed in a structured format:
- Products are **categorized** (e.g., *Tinctures*, *Pre-rolls*, etc.). The UI can present categories as separate sections with a heading, making it easy to scan. Within each category, each product is listed in a row or card format.
- For each product, show key details from the Excel: **Product name**, perhaps a short description (if provided), **THC/CBD content** (if relevant), **price per unit**, and possibly **case size** or bulk pricing if relevant to the client. You can highlight new or popular items if needed using simple badges (could be another column in Excel to indicate such status).

- If there are image assets (the Excel had a "Digital assets" column likely linking to images), consider showing a small thumbnail image of the product for visual appeal. This can be optional, but images often enhance user engagement. Just ensure images are optimized for web (small file sizes).

- **Responsive Design:** The layout should be responsive:

- On **desktop/tablet**, you might show a table or grid of products. For example, a table with columns: Product, Description, THC%, Price, Quantity (input).
- On **mobile**, a single-column list or collapsible sections for categories would work better. Use media queries or a responsive framework to adjust the layout. Ensure form inputs are easily tappable and that the font is legible on smaller screens.

- Test the UI on common devices to ensure no elements are cut off and that horizontal scrolling isn't needed.

- **Ordering Interaction:** Each product listing includes an input for the client to specify the quantity they want to order. Some UX considerations:

- Use an `<input type="number" min="0">` for quantity to allow easy increment/decrement (browsers show little arrows). Set reasonable defaults (maybe 0) and perhaps a max if appropriate (or leave it open if any amount).
- If the list is long, you might implement an "Add to Order" button on each row. Pressing it could highlight that the item is selected. But since we're collecting all orders in one form, it might be simpler to have clients enter all desired quantities and press a single **Submit Order** at the bottom.

- For long menus, consider also a "Back to top" link or floating summary of how many items selected.

- **Client Information Form:** At the top or bottom of the product list (or even as a fixed sidebar on wide screens), present input fields for **Dispensary Name**, **Contact Name**, **Email**, and any other required info (phone number maybe, if needed for confirmation). These are required fields to ensure the business knows who is ordering.

- Validate email format and other fields (HTML5 validation and additional JS for phone formatting if needed).

- If the client is usually known (like a repeat customer login scenario), you might consider allowing them to save their info or pre-fill it, but that's an enhancement. Initially, just a fresh form each time is fine.

- **Submitting the Order:**

- Use a prominent **Submit Order** button. Make it clear (e.g., "Submit Order" or "Place Order").
- When clicked, give immediate feedback. For example, change the button to a "Please wait..." state to show the form is being processed. If using a standard form post, the user will see the next page (or the same page reloaded) after submission. If using AJAX, you can display a loader animation.
- After successful submission, show a **confirmation message** on the page. This could either be on a new page (if using form post, you can have the PHP script return a simple HTML thank-you message, possibly including a summary of what was ordered), or as an in-page pop-up/alert (if using AJAX, you can dynamically replace the form with a "Thank you, your order has been placed!" message). This confirmation should reassure the user that their order went through. You might say something like "Thank you for your order! We've emailed a confirmation to you and our team will be in touch if any further information is needed."

- If you implemented sending a confirmation email to the client, they should receive that almost immediately. Many businesses also use this email to provide a copy of the order details to the client for their records.

- **Error Handling:** Ensure that user errors are communicated clearly:

- If the user tries to submit without required fields, the form should not submit and instead highlight the missing info (HTML5 will do this by default for `required` fields).

- If there's a problem in submission (e.g., network issue, or the email API fails), inform the user. For example, if using AJAX, catch the error and display "Sorry, there was an issue submitting your order. Please try again or contact us at [phone/email]." If using a PHP form post, you might redirect to an error page or the same form with an error message. Proper error handling improves trust.

- **Accessibility & UX Enhancements:**

- Label all form inputs clearly (use `<label>` tags tied to inputs, so that clicking the label focuses the input, and screen readers can identify them).
- Consider table accessibility if using tables (add proper table headers).
- Tab order should flow logically (so one can tab through the form fields in order).
- Use contrasting colors for text and background to ensure readability.
- If many products, you could add a search/filter box to let the client quickly find a product by name. This is not strictly required, but can be a nice feature if the menu grows.

- If the menu data includes any time-sensitive info (like expiration dates or limited stock), highlight those. Since it's a live menu, presumably it's always updated to what's available.

- **Scalability of UX:** As the number of products or categories grows, the UI should handle it:

- If there are dozens of categories, maybe a collapsible accordion for categories could be used (so user can collapse sections they aren't interested in).
- If each category has many items, ensure performance of the rendering remains good (the client-side parsing of Excel for, say, a few hundred items is fine; if it became thousands, one might need to paginate or revisit using a database).
- The design should still function with larger data sets by leveraging the above techniques (search, collapse, etc.).

By focusing on a user-friendly interface, dispensary clients will find it easy to place orders, and the business will receive structured information ready to process. The combination of an automatically updated menu and a straightforward order form provides a **smooth experience similar to an e-commerce cart but in a simplified B2B order context**. All of this is achieved with minimal ongoing technical work, thanks to the automated integration between the Excel data source, GitHub version control, and IONOS deployment pipeline.

Best Practices for Maintainability and Scalability

Finally, here are some recommended best practices and considerations to ensure the system remains maintainable and can grow with your needs:

- **Code Organization & Documentation:** Keep your code clean and well-commented. For example, clearly comment the section of JavaScript that parses the Excel or JSON ("// parse products by category"), and the portion that sends the form. If someone else needs to work on the project later, this will be invaluable. Document in the README how the Excel data maps to the UI, so future updates to the Excel structure can be handled.

- **Excel Template Management:** Since the solution relies on a fixed Excel structure, maintain that structure rigorously. It may help to create a **data dictionary** or template description (e.g., "Column A: Product Name, Column B: Strain, Column C: THC %, … Sheet2 Column F: Short Description, etc."). If someone unfamiliar tries to change the Excel (like adding a new column), the documentation should warn that the front-end parsing code might need updating. In short, *changes to the data format should be coordinated with code changes*. Minor content edits (adding rows, etc.) require no code changes.

- **Scaling Data**: In the future, if the number of products grows significantly or if multiple menus (for different regions or clients) are needed, consider migrating the data to a more robust solution (like a database or at least a structured JSON maintained in repo). The current Excel approach is perfect for ease of use up to a moderate size. If it becomes slow for clients to download (say the Excel becomes a few MB), you might then switch to generating a JSON file server-side, which is typically smaller than the raw Excel and faster to parse. That is a relatively small change given the architecture (just add a build step or small API).

- **Scaling Users**: A static site can handle large numbers of concurrent users easily, since it's just serving files (and maybe performing a bit of JS work in their browsers). The main load considerations would be:

- The Excel parsing in JS (which is client-side, so it scales with users naturally – each user's browser does the work). If using build-time JSON, then there's virtually no load on user side beyond a simple fetch of JSON.

- The email submission burst if many orders come in at once. The email script execution is very short, so even dozens of simultaneous submissions should be fine on a typical server. If expecting very heavy use, you might implement queuing or more robust mail handling, but likely unnecessary.

- **Continuous Integration (CI):** Use GitHub not just for deployment but also for basic CI checks. For instance, you could add a GitHub Action that lints your JS code or validates the JSON output. Also, if using a build script for Excel, write tests for it (e.g., a small test Excel to ensure the parser outputs correct JSON). This prevents deploying a broken site if someone unintentionally messed up the parsing script or if the Excel had an unexpected format.

- **Continuous Deployment Settings:** In IONOS Deploy Now, take advantage of features like **staging deployments** if available. For example, you could set up a staging branch that deploys to a staging URL. That way, you can test new features or major changes on a staging site without affecting the production site. IONOS allows multiple deployments and even database integration if needed [12]. For a simple static site, branching for staging is easy – just push to the staging branch to see changes, and merge to main for production.

- **Backups and Versioning:** Because everything (code and data) is in Git, you automatically have version history. It's wise to occasionally backup the Excel data outside GitHub as well, especially if it's being updated frequently (just to have a copy in Excel history – though Git preserves versions, retrieving an old Excel from Git may require some know-how). Also backup any configuration like email credentials (securely, outside the repo).

- **Monitoring and Notifications:** Set up monitoring on the site and the email system. For instance, use an Uptime Monitoring service to alert if the site goes down. Monitor the dedicated email inbox for bounces (if an order notification email ever fails to deliver, you want to catch that). IONOS likely provides logs – check them (especially the PHP error log) occasionally to see if the order script is throwing any warnings/errors.

- **Privacy and Compliance:** Since this is a cannabis-related ordering system, ensure compliance with any data privacy regulations in your region (for example, GDPR if in EU, or state laws if in US). Generally, we are only collecting business contact info, but still treat it carefully. Provide a privacy notice if appropriate, informing what is done with the data (e.g., "We use your contact information solely to process your orders."). Also, since it's B2B (dispensary orders), marketing use is probably not relevant, but don't add them to any marketing list without consent.

- **Graceful Failure Modes:** Think about what happens if something goes wrong:

- If the Excel fails to load or parse (e.g., someone accidentally saved it in an incompatible format), the front-end should detect that and display an error message like "Unable to load live menu at this time." This is better than a blank page. Implement try-catch around the parsing, and maybe keep the last working JSON as a fallback.

- If the email fails to send (due to SMTP issues), the PHP script could catch that and either try an alternate method or at least return an error page saying "Order could not be sent. Please try again later." Logging such failures is useful for admin to follow up.

- **Future Feature Thoughts:** While out of scope for the initial deployment, the architecture leaves room for enhancements:

- **Admin Interface**: In the future, an authenticated admin page could be added to upload the Excel from a web form instead of GitHub. For now, GitHub commits are fine (especially since it provides version control), but a non-technical user might prefer a web UI. This would require building a file upload feature and some auth.
- **Inventory Management**: If you ever need to show stock levels or prevent ordering beyond stock, you'd need a real-time data source or at least regularly updated data. That would be a bigger shift (likely requiring a database). But the current system could be extended by adding a "Stock" column in Excel and simply displaying it or warning if an order exceeds it.
- **Multiple Menus or Custom Pricing**: If you have different client types or regions, you could support multiple Excel files (one per region, etc.) and perhaps deploy them to different URLs or allow the user to select their region which then loads a specific file. The code could easily be parameterized for this.

In summary, this implementation plan provides a **detailed guide to building a production-ready, static web dashboard for cannabis product orders**. It leverages **GitHub for source control and content updates, IONOS for reliable hosting and continuous deployment, client-side logic for dynamic data binding, and a minimal server-side component for handling orders securely**. By following this guide, you'll achieve a system that is easy to update (just drag-and-drop a new Excel into GitHub), requires minimal server management, and offers a smooth user experience for your clients. The use of established best practices and simple technologies will ensure the dashboard is maintainable and scalable as your business grows, all while providing a professional, responsive interface for users.

Sources:

- IONOS Deploy Now Documentation – on connecting a GitHub repo for automated static site deployment [11].
- *read-excel-file* library (JavaScript) – capability to parse Excel files in browser or Node to JSON [1].
- Stack Overflow – approaches to sending emails from a static or semi-static site (server vs API vs form services) [2] [3].

---

[1] [4] [5] GitHub - catamphetamine/read-excel-file: Read *.xlsx files in a browser or Node.js. Parse to JSON with a strict schema.
https://github.com/catamphetamine/read-excel-file

[2] [3] [8] javascript - Sending email from static html page - Stack Overflow
https://stackoverflow.com/questions/23111233/sending-email-from-static-html-page

[6] [7] [9] [10] [11] [12] Deploy Static Sites via GitHub | Docs | IONOS Deploy Now
https://docs.ionos.space/docs/deploy-static-sites/