ChatGPT

# Indoor Navigation & Product Finder MVP – Overview & Architecture

[1] [2] Large retail stores like Walmart can span 180,000+ sq ft with 100,000+ items, making it challenging for shoppers to find products [2]. Top retailers (Walmart, Home Depot, Kroger) now provide digital *store maps* and in-store navigation to create faster, easier shopping experiences [3]. This MVP platform addresses this need by combining **customer-facing** features (in-store entry via QR/BLE, a live floor map with "blue dot" positioning, product search with route guidance, and real-time offers) with **admin-facing** tools (floor plan upload, drag-and-drop placement of product markers/offers, and analytics dashboards).

**Architecture:** The solution uses a **React** front-end (with **Tailwind CSS** for a responsive UI and **React DnD** for drag-drop interactions) and a **Node/Express** backend serving RESTful JSON APIs. Data is primarily stored in JSON format for floor layouts and product/offer mappings, with a lightweight database (or JSON files) for persistent data (e.g. analytics logs, user sessions). The diagram below outlines the system:

- **Customer Web App:** Runs on a user's device (mobile browser as a PWA), scanning a store QR to load the correct floor map and optionally listening for BLE beacon signals for indoor positioning.
- **Admin Web Portal:** Allows store admins to manage maps and content.
- **API Server:** Handles requests (map data, search queries, routes) and reads/writes JSON or DB as needed.
- **Storage:** Floor plan images and metadata are stored on the server (or cloud storage), with product locations and offers saved as JSON config. A database is used where complex querying or persistence is required (e.g., storing usage analytics).

**Technical Stack:** We will use **React** (for modular UI components and state management) and **Tailwind CSS** (utility-first styling for consistent design). **React Router** will separate customer vs. admin views. The admin map editor uses **React DnD** hooks (`useDrag`, `useDrop`) to enable dragging icons onto the floor map [4] [5]. The backend uses **Node.js** with **Express** to implement API endpoints that serve JSON data to the front-end. Pathfinding can utilize a simple algorithm (e.g., Dijkstra/A* on a predefined graph of store nodes) implemented server-side or in-browser. We aim to keep config in JSON (e.g., a `store_map.json` defining aisles, node graph, product coordinates), and only use a database for user accounts or aggregated analytics. This architecture ensures a clear separation of concerns: real-time map interactions on the front-end, with the backend focusing on data retrieval and business logic.

# Customer Entry (QR Code & BLE Onboarding)

**Visual Concept:** When customers arrive in-store, they scan a **QR code** at the entrance or enable Bluetooth to initialize the app. Scanning the QR (via the phone's camera or an in-app scanner icon) takes them to the store's map page in the web app. If BLE beacons are deployed, the app can leverage them to show the user's approximate starting position (*"blue dot"*) on the map [6]. The entry screen (on first use) prompts for camera and location permissions, showing a simple **"Scan to Start"** interface with a camera viewfinder for QR codes or a toggle for "Enable Indoor Positioning" via BLE.

**Frontend Behavior:** On load, the PWA checks if a store ID is present (from a QR code URL or stored selection). If not, it might present a QR scanner (using the **MediaDevices** API to access the camera and a JS QR decoding library like **jsQR**) or allow manual store selection. For BLE, since *precise web-based indoor positioning is limited* [7], the MVP uses BLE only if supported (via Web Bluetooth or location services) – for example, detecting a beacon at the entrance to identify the store or section. In practice, due to browser limitations on background BLE, the QR approach is primary for MVP [7]. Once the QR code (which encodes store ID and optionally entry point) is scanned, the app requests the floor map data for that store via API and navigates to the **Floor Map** view.

**Backend Logic:** The server provides an endpoint like `GET /api/stores/{id}/map` to fetch the floor map image URL and associated data (dimensions, scale, POIs). If the QR encodes a specific location (e.g., "entrance A"), that can be sent to initialize the user's position. There's no sensitive data exchange – the QR just identifies the store/position, and the app then pulls public map data. The backend may also log the scan event for analytics (store visit count, timestamp).

**Storage Strategy:** Store identifiers and floor map configs are stored in JSON (for quick retrieval). For example, a JSON file per store could list beacon IDs or QR code locations mapped to coordinates. Persistent storage isn't heavily needed for scanning – scanning simply triggers loading of static map data – but we may keep a **JSON of beacon layouts** (if BLE is used) mapping beacon UUIDs to x,y coordinates to triangulate the user. (Advanced: For BLE RSSI triangulation, an external service or native app might be needed beyond MVP scope.)

**Libraries/Tools:** The QR scanning uses a JS library (e.g., **Instascan** or **ZXing** via wasm) for decoding camera input in the browser. The BLE integration (if any) would rely on the emerging **Web Bluetooth API** to scan for advertisement packets from known beacons – given that accurate web indoor location is not fully available yet [7], this MVP may limit to detecting a single beacon (e.g., near entrance) to set the initial position, rather than continuous tracking.

# Floor Map with Live Position ("Blue Dot")

**Visual Layout:** Once onboarded, the user sees an interactive **store floor map**. The map is typically a top-down layout of aisles and sections as an image or SVG. A "You Are Here" indicator (the **blue dot**) shows the user's current position on the map [6], updating in real-time if possible. For example, a blue circle or arrow is overlaid at the entrance initially and moves if the user changes position (in a full implementation). The UI includes basic map controls like zoom and pan, and a legend for key areas (entrances, restrooms, etc.). The image below conceptually illustrates a floor map with a position marker (blue dot) inside a store aisle:

*Example of a mobile store map view with current position indicator and item highlights (conceptual UI).*

**Frontend Implementation:** The floor map is rendered using an interactive library – e.g., **Leaflet** in non-geographic mode or an HTML canvas. We use Leaflet with a simple CRS (Coordinate Reference System) such that pixel coordinates correspond to map coordinates [8] [9]. The floor plan image (uploaded by admin) is placed as a static overlay at a defined scale. On top of this, we overlay markers for the user's position and other points of interest. The **blue dot** representing the user is a dynamic marker: initially placed at the entry point (from QR or a default). If continuous positioning is available (e.g., periodic BLE scans or device sensor

fusion), the front-end updates this marker's coordinates via state updates (e.g., using a React state for userLocation and re-rendering marker position).

The map supports **panning/zooming** so that users can explore. We set Leaflet to treat the image's pixel width/height as the coordinate bounds (using `L.CRS.Simple` and `L.imageOverlay` for the floorplan) [8] [9]. This allows adding markers via pixel coordinates of the image. For example, if a user's estimated position is (x=50, y=150) in image pixels, we add a marker at that coordinate on the map.

**Backend & Position Updates:** If using BLE or other sensors, the app might call an endpoint like `GET /api/stores/{id}/position?beacons=[list]` to get an estimated position from the server (if the calculation is server-side). However, MVP can perform a simpler approach: when the user scans different QR codes in the store (placed in various aisles), the app updates the position to that QR's known location. Continuous *live* tracking (blue dot moving as user walks) may require a native app or advanced web API; as Pointr notes, *there's currently no highly accurate web-based indoor location available* without specialized support [7]. So our MVP might simulate "live" by letting the user manually update position via QR scans or by assuming the user starts at entry (the blue dot remains at entry unless they scan another code or search for a product).

**Storage:** The floor map image (PNG/JPG) is stored on the server (e.g., in a public folder or cloud bucket) and delivered via URL to the app. A JSON config defines key anchor points: entrances, aisle centers, etc., each with coordinates. These are used to place the blue dot initially. If BLE were used, we'd store beacon coordinates in a JSON and maybe calibration data (but this is beyond MVP accuracy requirements).

**Libraries: Leaflet** (with React-Leaflet or directly) is a strong choice for map display due to its support for image overlays and markers in an intuitive way [10] [9]. Tailwind can style the container and any custom controls. The blue dot marker can be a Leaflet `L.circleMarker` or a custom icon. We ensure cross-device support (touch for mobile, click for desktop simulation) so users can interact with the map easily.

# Search & Route Guidance

**Visual Layout:** A **search bar** allows customers to find products by name or category. When a user searches "Kashi cereal", for example, the app returns a result and highlights the product's location on the map with a marker or pin. The UI might display the product's aisle info and an option to **"Show Route"**. Upon selection, a route line is drawn from the user's current position (blue dot) to the product's location, navigating through aisles. The map shifts or zooms to fit the route. An example of a route drawn on a floor map is shown below:

*Illustration of a route (blue path) drawn on an indoor map from the user's position to a product location.*

Additionally, turn-by-turn textual directions (e.g., "Go 2 aisles down, then turn right") could be displayed, but the MVP may simply show the path visually. The route updates in real-time if the user moves (in a full implementation), keeping the blue dot moving along the path [6].

**Frontend Behavior:** The search bar (React component) auto-suggests as the user types, querying an endpoint like `GET /api/stores/{id}/products?q=keyword`. When an item is selected, the front-end retrieves the item's map coordinates (either included in the search result or via a separate call). It then

drops a **product marker** on that spot (e.g., a pin icon or a highlighted aisle segment). If the user taps "Navigate" or similar, the app computes or fetches the route.

For path drawing, the app uses the store's graph of walkable paths (aisles intersections, etc.). This graph can be precomputed by admin or a simple grid. We implement a pathfinding algorithm in JS (e.g., Dijkstra or A* over the graph nodes). The resulting path is an array of waypoints (x,y pairs) that we draw as a polyline on the map canvas (Leaflet `L.polyline` or an SVG path). The front-end then focuses the map on the route. If the user's position changes (via a new QR scan or a theoretical live update), the path can be recalculated from the new position to the destination.

**Backend Logic:** The server provides search results and possibly routing. A `/api/search?query={name}` returns matching products with their location data (like aisle or coordinates). Each product in the database has an assigned aisle or coordinate (entered via the admin interface). For routing, one approach is to do it on the client to reduce server load. Alternatively, an endpoint `GET /api/route?from={nodeId}&to={nodeId}` could return a list of nodes or points to draw, if the server knows the store graph. For MVP simplicity, the floor's walkable areas can be represented as a grid or a network defined in JSON (e.g., nodes at aisle intersections). The **route guidance** uses this network – for example, "Node A (near user) to Node D (target aisle)" – computing shortest path. We ensure that each product location is associated with the nearest node in the graph (administrators will tag product locations relative to nodes). The **Walmart app** similarly drops a pin at the item's location and uses aisle info for guidance [11], and our MVP mirrors that concept.

**Storage:** The product catalog (or a subset for the MVP) is stored in a JSON or database. It contains product names, IDs, and location references (aisle number or precise coordinates). For quick development, we might maintain a JSON file like `products.json` listing items and their positions. The routing graph can be a JSON file (`mapGraph.json`) enumerating nodes (with coordinates) and edges (walkable connections). These JSON files are loaded by the server (or even by the client as static data) for route calculations [12] [13]. JSON is favored for easy editing, but if the product list is large, a lightweight DB or indexed search (SQLite or Elasticlunr.js on the front-end) might be used for quick text queries.

**Libraries/Tools: Fuse.js** (a lightweight fuzzy search library) can run in the browser for product search suggestions. Pathfinding can use a library or custom code – e.g., implementing Dijkstra since the graph is small (just a store). If we choose to offload route computation, Node can use a library like **graphlib** or just custom code to find shortest paths. We also ensure to handle edge cases (if user is already near the product or invalid queries). The UI for results uses Tailwind components (dropdown of suggestions, etc.) and the map updates are done via Leaflet as described. This provides a smooth experience: **searching an item drops a pin and shows an easy route to it** [11] [1].

# Offers & Promotions Overlay

**Visual Layout:** The platform can display **special offers** or promotional zones as an overlay on the map. For example, if certain aisles have discounts (e.g., a *"Buy 2 get 1 free"* area for snacks), those zones are highlighted on the map with an icon or shaded region. When the user is near an offer, or if they tap an "Offers" toggle, markers (like a % or sale tag symbol) appear at those locations. The UI could also show a list of current deals; tapping a deal focuses the map on that location. In our design, an offer might appear as a

colored icon on the map that can be tapped to see details (e.g., *"2 for $7 on Cereals in Aisle 5"*). This engages users with location-based promotions [14].

For instance, consider an overlay where aisle 5 is highlighted in yellow to indicate a promotion zone; a small popup might read "Cereal Sale – Aisle 5". The concept is akin to Walmart's Black Friday color-coded maps that show deal locations on the store map [1]. Offers can also be time-sensitive: the admin can schedule them to appear only during certain hours (the front-end can fetch offers live).

**Frontend Implementation:** The offers overlay is built on the same map component. A button or switch ("Show Offers") toggles the visibility of offer markers. These markers could be icons (e.g., a star or sale tag) placed at admin-defined coordinates. If the offer pertains to an entire section (e.g., an aisle), we could highlight the aisle region – for MVP, a simpler approach is to drop an icon at the aisle midpoint or use a semi-transparent rectangle overlay. The front-end fetches current offers via an API (`GET /api/stores/{id}/offers`) when the user activates the feature (or at app load). It then iterates through the offers list and adds each marker to the map.

We ensure these markers have interactivity: clicking one could open a small tooltip with the offer details (title, description, expiry). The user's current position (blue dot) can be used to filter relevant offers (e.g., show only if within X distance). In future, as Oriient notes, *real-time navigation enables location-based proximity marketing*, delivering **relevant offers at the most opportune time/place** [14]. In MVP, we simulate this by allowing manual toggle or highlighting all deals.

**Backend Logic:** The server maintains a list of active offers for the store. Each offer entry includes a location (which could be linked to a product or a zone). For example: `{ "id": 101, "title": "Cereal Sale", "description": "2 for $7 on select cereals", "coords": [120, 80], "radius": 10, "validTill": "2025-12-31" }`. The `coords` might mark the center of the promo area; `radius` (or an aisle ID) could indicate coverage. The API may simply dump all offers in JSON. Business logic such as automatically showing an offer when user is nearby can be handled client-side by comparing the distance between userLocation and offer coords (if we implement continuous positioning). If proximity detection is desired on backend (for precision), it could take user location input and filter offers, but given limited live positioning in web, the front-end approach is fine.

**Storage:** Offers are relatively few and can be stored in a JSON file or a database table if they need to be frequently updated by admins. A JSON structure is flexible – the admin portal can edit this JSON (or through an API endpoint like `POST /api/offers` to add one). For persistence and queries (e.g., "show all current offers"), a lightweight DB (SQLite or Mongo) can store offers especially if scheduling (validTill, etc.) or analytics (how many viewed) is needed. However, *favoring JSON*, we might store them in a file that the backend reads and updates atomically for MVP simplicity.

**Libraries/Tools:** No special library beyond the map library is needed for overlay – we use the mapping tool (Leaflet or React DnD canvas) to draw shapes or icons for offers. We might use a small icon image (e.g., sale tag SVG) for markers. For date handling (offer expiry), a utility like **Day.js** could be used. Tailwind can style an "Offers List" modal or tooltips. Optionally, using **Web Notifications** or toasts could alert a user of a nearby deal (if implementing proximity alerts). This feature aligns with the idea that *location-based offers throughout the journey can boost engagement* [14] – our MVP lays the groundwork by mapping deals onto the floor plan for easy discovery by shoppers.

# Admin Portal – Floor Plan Upload & Setup

**Visual Layout:** The admin interface for floor plans allows authorized users to **upload a floor plan image** and calibrate it. The UI might have an **"Upload Floor Plan"** button which opens a file picker for an image (JPEG, PNG, SVG etc., as supported [15]). After upload, the floor plan is displayed on screen, and the admin can define the scale or mark key reference points (like the store's dimensions or known coordinates). There may be simple inputs for metadata (store name, floor name, etc.). For MVP, we assume the floor plan image is already correctly scaled, so the admin mainly needs to upload and perhaps enter the image's pixel dimensions or real-world scale (for reference).

Once uploaded, the image is stored and a preview is shown in the admin portal. The admin can then proceed to add markers for products or zones in subsequent steps. For example, the admin clicks "Save Floor Plan" and the system stores it [16]. The UI flow might consist of: 1. Select image file (drag-and-drop or file input). 2. Preview image with basic zoom/pan. 3. Save -> this floor plan is now the base map for customer view.

**Frontend Implementation:** Using React, we create a component for floor plan management. The file input can be managed via an `<input type="file" />` element. We may use a library for image preview or simply use the FileReader API to show the chosen image before uploading. For styling, Tailwind helps position the upload form (e.g., a centered box with dashed border for drag-and-drop). Admins can drag a file onto it or click to browse. Once a file is chosen, the image appears in an `<img>` tag or canvas for preview.

We also capture some data: e.g., floor name, or set default map scale. If needed, the admin can input the real dimensions (e.g., "Store length = 100m") which could be stored to translate pixel distances to real distance (optional for MVP route distance estimation). The interface will also include a **submit** button to upload.

**Backend API & Logic:** We provide an endpoint like `POST /api/admin/floorplan` to handle the file upload. The backend (Express) can use middleware like **multer** to accept image files. On upload, the server saves the file (e.g., to `public/floorplans/{storeId}.png`). We also create or update a JSON record for that store's map (storing the file path/URL and any metadata). The endpoint returns a success response with the stored filename or ID. If the admin is updating an existing map, the old image might be replaced (the API can handle both create and update).

We enforce format checks (only allow images of certain types; Appspace docs note JPG, SVG, BMP, PNG, GIF are acceptable [15]). Optionally, we might generate multiple resolutions or thumbnails, but not necessary if we use client-side scaling.

After successful upload, the admin front-end can call `GET /api/admin/floorplan` to fetch the new map info and render it for further editing steps (tagging). The system **stores** the floor plan info persistently – likely in a JSON file or a small database record. Using JSON, we could have a master `stores.json` where each store entry has a `floorPlanImage: "path/to/image.png"`. If multiple floors or stores exist, data structure should accommodate that (maybe keyed by store ID).

**Storage Strategy:** Favor JSON or file storage here – images are binary so they will reside as files on disk or cloud (not JSON). But the reference to them and calibration data can be in JSON. For example:

```
{
  "storeId": "Walmart_001",
  "floorPlan": {
     "image": "floorplans/Walmart_001_floor1.png",
     "width_px": 1200,
     "height_px": 800,
     "scale_m_per_px": 0.1
  },
  "productMarkers": [ ... ],
  "zones": [ ... ]
}
```

This JSON can be updated after upload (we populate the `image` path and maybe the image's pixel size). Persistent storage of this JSON could be in a file or a NoSQL collection. If using a relational DB, we'd store a record with a file path to the image on disk or cloud URL.

**Libraries/Tools:** On the client side, we might use **Dropzone.js** or a React component for drag-drop file upload to enhance UX (or HTML5 DnD events to highlight drop area). On backend, **Multer** as mentioned for file handling. No heavy library is needed for simply moving the file to storage. We will likely also incorporate authentication for admin routes (a simple login system or even a static password for MVP). The admin portal itself can be a protected route in React; we won't delve deep into auth here but in practice an admin login and session would be required to use these upload/edit APIs.

# Admin Portal – Map Tagging & Product Placement (Drag-and-Drop)

**Visual Layout:** After uploading a floor plan, admins can enter a **"Edit Map"** mode where they place markers for products, categories, or important points. The interface shows the floor plan image (often scaled to fit in the viewport) and a palette of marker icons (for example, a pin for products, a star for promotional zones, perhaps different icons for different product categories or store amenities). The admin can **drag and drop** these icons onto the map to mark locations. For instance, an admin searching for "Cereal" can drag a cereal-box icon onto Aisle 5 where cereals are located. They can also simply drag a generic product marker and then fill in details (product name/ID, etc.) in a form.

During dragging, the marker might appear semi-transparent and snap to the nearest valid position (we may implement simple grid snapping or just free placement). Once dropped, the marker's coordinates on the image are recorded. The admin can click the marker to edit its properties (e.g., assign a product or category, or label it). They can also reposition or delete markers as needed.

**Frontend Implementation:** We utilize **React DnD** for this feature. The application wraps the relevant components in a `DndProvider` (likely using the HTML5 backend for desktop use, and it supports touch

events as well) [17]. We define **drag sources** for the palette icons (using `useDrag`) and **drop targets** for the map area (using `useDrop`) [4] [5]. The map image container is a drop target that accepts certain item types (e.g., "PRODUCT_MARKER"). When a drop occurs, we calculate the drop coordinates relative to the image. For example, React DnD provides the pointer offset; we adjust for the map container's position and scale to get image pixel coordinates.

As markers are placed, they become part of the state (e.g., an array of marker objects in a React state or context). We render each marker as an absolutely positioned `<div>` or an SVG element on top of the map. Markers could be draggable post-placement too – implementing `useDrag` on placed markers allows admins to reposition them by dragging again (the drop target remains the map). We also implement a way to select a marker (e.g., clicking highlights it and opens a sidebar or popup form to input details like "Product: Kashi Cereal, SKU: 12345, Aisle: 5").

For bulk placement, an admin might prefer to see a list of products and drag them onto map. We can facilitate a **search bar in admin mode**: type a product name to filter the list, then drag it from the list to the map. Alternatively, if the admin just places unlabeled pins, they can later click and assign a product from a dropdown.

**Backend API & Logic:** Each placement or edit can be sent to backend via APIs. For example: - `POST /api/admin/markers` with body `{type: "product", productId: 12345, x: 100, y: 250}` to add a marker. - `PUT /api/admin/markers/{markerId}` to update position or info. - `DELETE /api/admin/markers/{markerId}` to remove one.

However, to minimize friction, we might allow the admin to do multiple placements and then hit "Save Changes" which sends the full markers list in one request. The backend will update the store's JSON config accordingly (persisting all marker positions and their links to products). Using JSON, the store entry might have a `productMarkers` array that we replace entirely with the new set on save.

**Storage:** Product placement data ties a product (or category) to a coordinate. We store these in the store's JSON config. For example:

```
"productMarkers": [
   { "id": "prod_12345", "name": "Kashi Cereal", "x": 854, "y": 322 },
   { "id": "prod_67890", "name": "Organic Oats", "x": 860, "y": 500 }
]
```

We may use product IDs to refer to an external product database if one exists. If not, the admin-entered name is stored directly. For categories or general tags, we might have entries like `{ "id": "category_dairy", "label": "Dairy Section", "icon": "milk.png", "x": ..., "y": ... }`. These JSON entries are loaded by the customer app to render the map pins and to search (the search could match product names or categories from this dataset).

If a more robust persistent store is needed (especially if many markers), a simple database table could store each marker with fields for store, coordinates, and linked product ID. But JSON is perfectly adequate for an MVP with moderate data size, and it's human-editable if needed.

**Libraries/Tools: React DnD** (already integrated) handles the heavy lifting of drag/drop interactions in the UI, making it straightforward to capture drop coordinates and move items around [4]. We may use an icon library or custom SVGs for marker icons. Tailwind can help style the edit sidebar or marker tooltips. If needed, for the map display in admin mode, we might reuse the same Leaflet/canvas component but augmented with drag/drop – or simply position markers in a relative/absolute div since precise geospatial calcs aren't needed (pixel positioning is enough).

Overall, this feature empowers the admin to **visually map out where each product or point of interest is on the floor plan**. It aligns with how Walmart's teams maintain maps: placing products in correct locations and keeping info updated [18]. Our system provides a user-friendly way to do that with drag-and-drop instead of raw coordinates.

# Admin Portal – Offers & Zone Management

**Visual Layout:** In addition to standard products, admins need to mark **offer zones** (areas where promotions apply) and potentially define store **zones** or sections (for analytics or navigation constraints). In the admin UI, this could be an extension of the tagging feature: for example, a palette of markers might include a "Sale" icon which the admin can drop on the map wherever an offer should be. Alternatively, the admin can select an existing product marker and mark it as on sale (linking it to an offer entry).

For zone management, the admin might draw or tag sections of the map (like defining that Aisle 1 is the "Dairy Zone" or grouping a cluster of aisles into a department). MVP can simplify this by treating zones similarly to offers – essentially named regions. The UI might allow the admin to draw a rough polygon or rectangle on the map for a zone; however, implementing free-draw is complex for MVP. Instead, the admin can place a special marker as a **zone label** (e.g., a pin that represents an entire aisle or area) and assign it a category (like "Produce Section"). This serves two purposes: labeling the map for user reference and grouping analytics.

For **offers**, the admin interface might include an "Offers" panel listing current offers. Admins can add a new offer by selecting a location (perhaps by clicking on the map) and entering promo details (title, description, validity). We can combine this with drag-and-drop: drop a "sale" icon on a location, then fill in the details in a popup form.

**Frontend Implementation:** We reuse drag-and-drop for offers: the admin drags an offer icon onto the map. On drop, we capture coordinates and then prompt the admin with a modal form (using a controlled component or popover) to input the offer info (e.g., "Discount details, expiry date"). Alternatively, the admin could first fill out an offer form (with text and maybe select an icon color) and then click a "place on map" button that lets the next click on the map set the location. For simplicity, the drag approach is more consistent with product placement.

Zone marking (like entire aisle labeling) could be done by dropping a "Zone" icon at the aisle's start and naming it. If needed, multiple products can be associated with a zone by admin in a separate data view (or simply by naming conventions). Given MVP scope, we might not implement drawn polygons for zones, just point tags.

**Backend API & Logic:** Adding an offer might trigger an API call such as `POST /api/admin/offers` with details { title, description, x, y, [radius], validFrom/To }. The backend will append this to the offers list in storage. If an admin updates or removes an offer, corresponding `PUT` or `DELETE` calls are made. The logic ensures that expired offers are filtered out (either by the backend not serving them if now > validTo, or by the frontend hiding if expired).

For zones, if we treat them as special markers, the API could be similar to product markers but with type "zone". For example, `POST /api/admin/zones` { name: "Electronics Dept", x: 500, y: 300, category: "electronics" }. The backend would save it under a zones section in JSON. These zones might be used for analytics grouping or displayed on the consumer map as labels.

**Storage:** Offers are stored as discussed in the Offers Overlay section – likely JSON with coordinates and text. Zones (if implemented simply as named points) can be stored in JSON as well, or even folded into productMarkers as a separate type. For clarity, we might have:

```
"offers": [
  { "id": "offer1", "title": "Holiday Sale", "x": 250, "y": 600, "details":
"All toys 20% off", "expires": "2025-12-31" }
],
"zones": [
  { "id": "zone1", "name": "Electronics", "x": 800, "y": 200 }
]
```

This is straightforward for the app to parse and overlay. If zone polygons were needed, we could store a list of coordinates outlining the area, but MVP will avoid that complexity.

**Libraries/Tools:** No extra libraries beyond what we have are needed. We will use our form handling (possibly React Hook Form for ease of managing form state) to capture admin input for offer details. Date pickers could be plain inputs or a simple library if needed for selecting validity dates. **Tailwind** helps style the forms and modals in a consistent, clean way.

By giving admins these tools, the store maps stay up-to-date with promotional information just like Walmart's team color-codes and highlights deals on their digital maps [19]. Admins can quickly add or remove a sale marker as campaigns change, ensuring customers always see relevant offers in the app at the right location [14]. This also sets the stage for dynamic marketing – although push notifications via geofence are beyond MVP, the data is in place for such future enhancements [20].

# Admin Portal – Analytics Dashboard

**Visual Layout:** The admin site also features an **Analytics Dashboard** where store managers can view usage data: e.g., most searched products, popular routes, dwell times at certain areas, and offer engagement. The dashboard is presented as a series of widgets or charts. For example, a bar chart might show the **Top 10 searched items** this week; a heatmap overlay (in a more advanced case) might show the map with intensity where users spent the most time. For MVP, we can focus on key metrics such as: - **Search Analytics:** a list of top search queries and how often they were made. - **Navigation Usage:** number of route requests or map

views per day. - **Offer Interactions:** which offers were tapped or viewed the most. - Possibly **Entry Scans:** count of QR scans (footfall via the app).

The visual design could be a simple web page with a few charts (using a library like Chart.js) and summary numbers (KPIs). For example: *"125 searches today"*, *"Most searched: 'Milk' (20 times)"*, *"Offer X viewed 50 times"*. If multiple stores were supported, an admin could filter by store, but in MVP assume one store context.

**Frontend Implementation:** The dashboard is a protected route in the React admin app (only accessible to logged-in admins). We utilize a chart library such as **Chart.js** or **Recharts** to create graphs. For instance, a pie chart for search categories or a line chart of daily usage. We fetch analytics data from the backend via endpoints like: - `GET /api/admin/analytics/searches?since=2025-06-01` – returns aggregated search counts. - `GET /api/admin/analytics/offers` – returns offer click/view counts. Alternatively, one `GET /api/admin/analytics` could return a JSON blob with all needed stats for the dashboard in one go.

The React components then feed this data into charts. Tailwind can be used to quickly style the layout (maybe a grid of cards, each containing a chart or metric). We ensure the dashboard is responsive so admins can even check on a tablet or mobile if needed.

**Backend Data Collection:** To produce these analytics, the backend (or even front-end) must log events. Key events to log: - When a user searches (e.g., query "milk"), log it (increment count or store timestamp). - When a route is generated or a product pin is viewed, log which product. - When an offer marker is clicked or shown, log it. - QR scan entries can also be logged.

We can implement a simple logging mechanism: for each event, append an entry to a log JSON or increment counters in memory that flush to a JSON/DB. For example, maintain a JSON like:

```
"searchCounts": { "milk": 20, "bread": 15, "cereal": 10, ... },
"offerClicks": { "offer1": 8, "offer2": 3 },
"dailyVisits": { "2025-06-22": 45, "2025-06-23": 50 }
```

This could be updated through API calls (e.g., front-end calls `POST /api/analytics/search` with {query:"milk"} whenever a search happens). The backend then updates the count. For MVP simplicity, even writing to a JSON file per day would work. However, concurrency and persistence considerations might push using a real database or at least an in-memory structure that dumps to file periodically.

For the dashboard requests, the backend aggregates data as needed. If using a DB, we'd run queries (like SQL COUNTs or NoSQL aggregation). If using JSON logs, we might load them and tally counts.

**Storage:** If traffic is low, JSON logs are fine. For more robust solution, a small database (SQLite or a cloud DB) would handle writes of each event. For example, a SQLite table `SearchLog(query text, time datetime)` where we insert a row per search. Then `SELECT query, COUNT(*) FROM SearchLog GROUP BY query` gives counts. For MVP, even an in-memory array that resets on server restart could be acceptable (though we'd lose data on restart). A middle ground: keep cumulative counts in a JSON and update it on each event (ensuring to lock file or use atomic

write to avoid corruption). This is a design decision balancing *favoring JSON* with reliability – we lean on JSON to adhere to instructions, acknowledging a DB might be needed as data scales.

**Libraries:** For charts, **Chart.js** (via react-chartjs-2) or **Recharts** are easy to integrate for visualizing data. For date handling and formatting (if showing trends by day), again Day.js or similar can help. If using a DB, the backend might use an ORM like **SQLite3** or direct queries. But likely, we'll implement minimal custom logic (reading JSON, counting) for MVP.

This analytics feature provides valuable insights [21] – e.g., seeing that many users search for "bread" could indicate either bread is hard to find or very popular, informing the store layout or signage. Also, if an offer has low clicks, the manager might move its placement or improve visibility. By summarizing *"data generated by each customer journey"*, the dashboard helps optimize operations and marketing [21]. Even with a basic MVP, presenting these metrics completes the platform's value loop: admins can improve the in-store experience based on real usage data.

# API Endpoints & Interaction Flows

Finally, we summarize the key API endpoints and how front-end and backend interact in this MVP:

**Customer-Facing API Endpoints:** - `GET /api/stores/{storeId}/map` – Returns floor map data: image URL, dimensions, and all markers (product locations, zones, etc.) that the client should display. *(Called when a user scans QR or opens the store map)*. - `GET /api/stores/{storeId}/products?query=XYZ` – Searches products (by name or keywords) and returns a list of matches (with product ID, name, and location info). *(Used for the search bar autocomplete and results)*. - `GET /api/stores/{storeId}/route?from={pointA}&to={pointB}` – (If server-side pathfinding) Returns a sequence of coordinates or node IDs for the shortest path. If pathfinding is client-side, this may not be needed; instead the client computes using data from the map config. - `GET /api/stores/{storeId}/offers` – Returns current offers (each with location coords and details) for the store. *(Used to overlay offers on the map)*. - (Optional) `POST /api/analytics/event` – Used by client to log events (search, offer click, etc.). For example, body `{ type: "search", query: "milk" }`. We might also have specialized ones like `POST /api/analytics/search` or `.../offerView` to increment counters. These calls happen in the background when users perform actions.

**Admin-Facing API Endpoints:** - `POST /api/admin/floorplan` – Uploads a new floor plan image. (Multipart form-data). **Response:** JSON with success and perhaps the stored filename. *(Used when admin uploads or replaces a floor map)*. - `POST /api/admin/markers` – Add a new marker (product or other). **Body:** { type, x, y, [productId/name] }. **Response:** new marker ID. - `PUT /api/admin/markers/{id}` – Update marker position or data. - `DELETE /api/admin/markers/{id}` – Remove a marker. - `POST /api/admin/offers` – Add a new offer. **Body:** { title, description, x, y, [radius], validUntil }. Returns an ID. - `PUT /api/admin/offers/{id}` – Edit offer. - `DELETE /api/admin/offers/{id}` – Remove offer. - `GET /api/admin/analytics` – Get aggregated analytics data (could include multiple sections: searchCounts, usageStats, etc.) [21] for populating the dashboard.

*(Note: The admin endpoints would typically require authentication – e.g., a token or session cookie – but for MVP we might assume a simple auth layer not described in detail.)*

**Interaction Flow Highlights:** - **Store Entry Flow:** User scans QR -> Browser opens store URL (or app navigates to storeId) -> Front-end calls `/api/stores/{id}/map` -> Backend returns map + markers JSON -> Front-end displays map with markers and awaits user input. - **Search Flow:** User types query -> Front-end calls `/api/stores/{id}/products?query=q` -> Backend returns matches -> User selects item -> (Front-end optionally logs selection via analytics) -> Front-end either knows coordinates from response or calls map API to get details -> If route requested, either front-end computes using map data or calls `/api/stores/{id}/route?from=A&to=B` -> Once path obtained, front-end renders it on map. - **Offer Display Flow:** User toggles "Offers" -> Front-end calls `/api/stores/{id}/offers` -> Backend returns offers list -> Front-end adds offer markers to map. If user taps an offer marker -> maybe no extra API call (details already provided), but front-end could log it via analytics. - **Admin Editing Flow:** Admin logs in -> Navigates to Floor Plan page -> Front-end calls `/api/stores/{id}/map` to get current map and markers for editing -> Admin uploads a new image (front-end POST to `/api/admin/floorplan`) -> on success, maybe refresh map data. Admin drags markers -> on dropping each, could call `POST /api/admin/markers` or wait until "Save" -> Admin hits save, front-end sends batch of markers (perhaps using multiple calls or one bulk update endpoint) -> Backend stores the updated marker list in JSON. Similarly for offers: admin adds/edits -> calls corresponding endpoints -> backend updates offers JSON. - **Analytics Flow:** When admin opens dashboard -> Front-end calls `/api/admin/analytics` -> Backend compiles data (from logs or DB) and returns JSON stats -> Front-end renders charts. Data collection for analytics occurred via prior user actions hitting `POST /api/analytics/...` endpoints (these are fire-and-forget in the user app, not affecting UX).

Every API responds with JSON, and errors (like validation issues on upload or unauthorized access) are handled gracefully with messages. The system favors JSON for easy interchange of structured data between front and back, and uses files or simple structures over heavy databases where feasible, per MVP needs.

**Conclusion:** This comprehensive plan outlines a web-based indoor navigation and product-finding platform with a rich feature set. Shoppers get an interactive store map with real-time position and route guidance to any product, plus on-map promotions to enhance their experience [22] [1]. Administrators have intuitive tools to keep maps updated – uploading new layouts and tagging product locations via drag-and-drop – as well as insights into system usage to continually improve the service [21]. By leveraging modern web tech (React, Tailwind, Leaflet, React DnD) and structured JSON-driven data, this MVP can be implemented quickly and evolve with more advanced indoor positioning capabilities in the future. All components work in concert to make **in-store navigation** as seamless as the now-familiar GPS navigation, bringing the convenience of digital wayfinding into the retail space.

---

[1] [2] [11] [18] [19] [20] Walmart Uses Location-Based Technologies to Improve In-Store Navigation for Shoppers - Geospatial World
https://geospatialworld.net/prime/case-study/location-and-business-intelligence/walmart-uses-location-based-technologies-to-improve-in-store-navigation-for-shoppers-3/

[3] [6] [14] [21] [22] From Store Maps to Real-Time Navigation - Oriient
https://www.oriient.me/from-store-maps-to-real-time-navigation-3/

[4] [5] [17] How to implement drag and drop in React with React DnD - LogRocket Blog
https://blog.logrocket.com/drag-and-drop-react-dnd/

7 Indoor location in browsers - explained
https://www.pointr.tech/blog/dispelling-web-based-bluedot-myth

8 9 10 javascript - Creating overlay on an image using Leaflet - Stack Overflow
https://stackoverflow.com/questions/22714954/creating-overlay-on-an-image-using-leaflet

12 13 15 16 Create Buildings, Upload Floor Plan Maps, & Configure Zones in Locations - Introduction
https://docs.appspace.com/latest/introduction/configure-maps-floor-plans/