# Garden Planner & Management App – Product Requirements Document (PRD)

## Introduction

This PRD outlines the vision and requirements for a **personal garden planning and management web app**. The user is a homeowner with a nearly one-acre property containing lawns, flower beds, shrubs, and mature trees. The goal is to provide a modern web application that helps the user **map their garden**, **catalog plants**, and **schedule garden tasks**. Leveraging Google Maps imagery as a base, the app will allow the user to visualize their garden layout and record detailed information about each plant and area. Over time, intelligent features (like optimized mowing routes and, later, machine learning enhancements) will be added to improve efficiency and guidance in garden maintenance. This document captures the core features and technical approach required to fulfill this vision.

## Product Vision and Goals

* **Centralized Garden Map:** Create an interactive digital map of the user's property (approximately 1 acre) using Google Maps or similar satellite imagery as a starting point. This map will serve as a canvas for planning and tracking everything in the garden.
* **Plant and Asset Management:** Allow the user to catalog all plants, shrubs, trees, and garden features by placing markers or drawn areas on the map. Each item can have details like species, planting date, care instructions, etc., pulled from authoritative plant databases.
* **Task Planning and Reminders:** Enable the user to schedule and track maintenance tasks (mowing, watering, pruning, fertilizing, etc.), with reminders and logs. This helps ensure no important garden task is forgotten, effectively acting as a personal gardening assistant.
* **Efficiency and Optimization:** Provide tools to improve gardening efficiency, such as suggesting the most efficient route to mow the lawn with a ride-on mower. An algorithm will generate an optimal mowing pattern that covers the lawn area with minimal overlapping or backtracking.
* **Seamless User Experience:** Build a fluent and modern UI that is intuitive and visually appealing. The interface should use clear visuals (including shaded map overlays to distinguish different zones) and responsive design so it’s usable on desktop or tablet via web browser.
* **Scalability and Future Intelligence:** Lay the groundwork for future intelligent features (e.g. machine learning for plant care recommendations or more advanced route optimizations). The initial version focuses on core functionality with simple algorithms, but the architecture should accommodate adding smarter features later.

## User Personas and Use Cases

**Primary User:** The homeowner (garden enthusiast) who wants a tailored tool for managing their own garden. They are tech-savvy enough to use web apps and are passionate about optimizing their garden care routine. Key use cases include: - *Mapping the Garden:* As a user, I want to see an overhead map of my property with boundaries, so I can mark where each lawn, bed, tree, and shrub is located. - *Recording Plant Details:* As a user, I want to catalog each plant (trees, shrubs, flowers) by placing a pin or outline on the map and attaching details (name, species, planting date, notes). I’d like the app to fetch species information (like care needs, growth habit) from an online plant database for accuracy. - *Planning Maintenance Tasks:* As a user, I want to schedule recurring tasks (mowing, watering, fertilizing) and one-time tasks (planting new bulbs, trimming a tree) and have the app remind me when these tasks are due. I want to log when tasks are completed so I have a history of garden maintenance[[1]](https://www.reddit.com/r/selfhosted/comments/169o20h/introducing_plantit_a_selfhosted_open_source/#:~:text=only%20one%20in%20charge%20of,since%20last%20action%20on%20them). - *Efficient Mowing Route:* As a user, I want the app to suggest a route for mowing my lawn with my ride-on mower that covers the area efficiently without unnecessary overlap. This would save me time and fuel by minimizing redundant passes[[2]](https://executivelawncare.net/2025/05/12/best-mowing-pattern-for-riding-mower/#:~:text=1.%20). - *Visualizing Garden Plans:* As a user, I want to sketch out future changes (e.g., a new flower bed or tree planting) on the map to see how they fit into the current layout. The app should let me draw or shade areas to represent these plans. - *Task Reminders:* As a user, I want to receive reminders or see alerts for tasks due (for example, “Time to water the rose bed” or “Lawn mowing due this week”) so that I don’t forget important garden chores.

Secondary (Future) use cases might include sharing the garden map with family or a gardener, but for now the application is intended as a single-user personal tool.

## Key Features and Functional Requirements

### 1. Interactive Garden Mapping

* **Base Map Integration:** The app will use Google Maps (satellite view) as the base layer to get an accurate aerial view of the property and its boundaries. The user can center the map on their property and set a fixed zoom level covering the entire acre. We will integrate the Google Maps JavaScript API (or a similar mapping library) to display the map in the app[[3]](https://medium.com/@nilima.sust/how-to-build-an-interactive-google-maps-store-locator-with-javascript-node-js-and-mongodb-ef7161c0420c#:~:text=This%20comprehensive%20blog%20post%20will,customization%20store%20locator%20from%20scratch). An API key will be required for Google Maps usage.
* **Property Boundary and Zones:** The user can outline the property boundary and different zones (lawn areas, flower beds, shrubbery clusters, etc.) on top of the map. Drawing tools (polygons, circles, lines) allow the user to trace garden beds or lawn sections. These drawn shapes can be filled with semi-transparent shading or color to distinguish areas (for example, shading the lawn area in green and flower beds in another color for visual clarity).
* **Markers for Plants and Trees:** The user can drop pins/markers on the map for each significant plant or tree. Clicking a marker opens a detail view for that plant. Markers might use custom icons (tree icon for trees, flower icon for flowering plants, etc.) to differentiate plant types at a glance.
* **Map Layers and Labels:** The map should support a layer of user-added content (the shapes and markers described above). The user can toggle the visibility of certain layers (e.g., hide the maintenance task markers or show/hide labels on plants). Labels could be the plant name or ID, shown next to markers or on zones.
* **Measurement Tool:** (Optional) Provide a way to measure distances or areas on the map (e.g., how large a lawn area is, or how long a hedge is) for planning purposes. This could use the map’s scale to calculate real-world dimensions.

### 2. Plant Catalog & Information Integration

* **Plant Detail Records:** Each plant or tree added to the map has an associated record stored in the database. At minimum, it includes: common name, scientific name, date planted, last fertilized date, last pruned date, and any custom notes. The user can add photos of the plant (optional) to track its growth or health over time.
* **External Plant Database Lookup:** To enrich the plant records, the app will integrate with an external plant database API. For example, using an API like Trefle or APIFarmer to retrieve species information (preferred soil, sun requirements, mature size, etc.) based on the plant’s name. These databases contain tens of thousands of plant species and their characteristics[[4]](https://apifarmer.com/plant-database-api/#:~:text=Plant%20Database%20API%20gives%20you,to%20meet%20almost%20anyone%E2%80%99s%20needs), ensuring the user has accurate data on their plants. When the user inputs a plant name, the app can fetch details such as ideal watering frequency, growth habits, and other care tips.
* **Search and Add Plants:** The user can search for plant names when adding a new plant. The app should provide autocomplete suggestions (possibly from the plant API) to ensure correct naming. Once selected, the plant’s information is fetched and stored. The user can override or edit any details if needed.
* **Plant List View:** In addition to the map view, provide a list or catalog view of all plants in the garden. This list can be filtered or sorted (e.g., by plant type, by next task due, alphabetical, etc.). Each list entry links back to the map location for easy reference.
* **Plant Care Reminders:** Using the species info and user’s input, the app could suggest care schedules (for example, if a plant typically needs weekly watering or quarterly fertilization, the app can prompt the user to create a recurring task for it). This will be more fully utilized in the Task Scheduling feature.

### 3. Task Scheduling & Reminders

* **Task Creation:** Users can create tasks associated with the garden or specific plants. Tasks have a title, description, associated plant/area (or general task not tied to a specific plant), due date or recurrence rule (e.g., every week, every spring, one-time on a specific date), and priority. For example: “Mow the lawn” every 2 weeks, “Prune rose bushes” every March, or “Water the apple tree” every week in summer.
* **Reminders/Notifications:** On login or at a scheduled time, the app will display due or upcoming tasks. Since this is a web app (and for personal use), we can implement email reminders or push notifications (using the browser’s notification API) for due tasks. The user should be alerted when something needs attention. A history log of when tasks were completed is useful – e.g., the user marks a task as done, and the completion date is recorded.
* **Task View on Calendar:** Provide a calendar or timeline view where the user can see upcoming tasks and past completed tasks. This helps in planning (for instance, seeing that fertilizing is due next week, or that the lawn was mowed 10 days ago).
* **Integration with Map:** Tasks related to specific locations or plants should be visible on the map. For example, if there’s a task “Prune Oak Tree”, the oak tree marker could have a small indicator icon or colored highlight when the task is due. Clicking it could show the task details. General tasks (like overall lawn mowing) could be linked to a zone (the lawn area shape could change color when mowing is due).
* **Logging and History:** Each time a task is completed, the user can mark it done in the app. The completion updates the plant’s record if applicable (e.g., updating “last watered” date for that plant). The app keeps a history of actions, so the user can review what was done when – useful for noting patterns (like how frequently they actually mow, or when a plant was last fed). This feature echoes the functionality of some gardening apps which log activities and notify users of time since last action[[1]](https://www.reddit.com/r/selfhosted/comments/169o20h/introducing_plantit_a_selfhosted_open_source/#:~:text=only%20one%20in%20charge%20of,since%20last%20action%20on%20them).

### 4. Lawn Mowing Route Optimization

* **Efficient Mowing Feature:** One standout feature is an intelligent lawn mowing guide. Given the user’s lawn area (which can be defined as one or more polygons on the map) and obstacles (trees, flower beds, etc.), the app will calculate an efficient route for mowing with a ride-on mower. The aim is to minimize wasted time, fuel, and overlapping paths by finding a pattern that covers the area systematically[[2]](https://executivelawncare.net/2025/05/12/best-mowing-pattern-for-riding-mower/#:~:text=1.%20).
* **Route Algorithm:** Initially, a simple algorithm will be used. This could be a back-and-forth striping pattern or a spiral pattern depending on the shape of the lawn. The algorithm should avoid obstacles by treating them as “no-mow zones” on the map. For example, it could generate a series of parallel lines across the lawn area and indicate turns at the edges, similar to how one would manually do stripe patterns or a spiral cut. The user might input the width of their mower deck, so the app can space the strips correctly and estimate how many passes are needed.
* **Interactive Route Map:** The mowing route can be displayed on the map as an overlay (lines or arrows showing the path). This gives a visual guide for the user to follow on their real mower. The route should show where to start, direction of travel, and how to proceed step by step (e.g., numbered waypoints or a continuous path line). Optionally, allow the user to add a starting point (like which corner of the lawn they prefer to start).
* **Adjustments and Learning:** Since this is a relatively simple task computationally, we anticipate being able to refine the algorithm with user feedback. In future iterations (beyond MVP), the app could utilize more advanced pathfinding or even machine learning to account for irregular shapes or to improve efficiency further. For now, even a fixed recommended pattern (such as a U-shaped or spiral mowing pattern) can drastically reduce overlapping and turning time during mowing.
* **Benefit to User:** This feature transforms the user’s mowing routine from a trial-and-error process to a guided experience. By minimizing unnecessary turns and overlaps, the user saves time and fuel and achieves a more professional, even cut on the lawn[[2]](https://executivelawncare.net/2025/05/12/best-mowing-pattern-for-riding-mower/#:~:text=1.%20). (It also adds a fun, game-like element – following the optimal path each time to beat your “best mowing time”.)

### 5. User Interface & Experience

* **Fluent Design Aesthetics:** The UI will follow a modern design (inspired by Fluent UI principles) with clean typography, intuitive icons, and subtle use of color and shadows. Using Fluent UI libraries or similar will give a polished look with consistent components (buttons, forms, dialogs) and smooth transitions. For example, use **shading and elevation** effects to highlight active elements or selected map regions, providing a tactile feel to the interface (e.g., a selected plant marker might glow or pop out with a shadow).
* **Responsive Web App:** The application is web-based and will run in modern browsers. It should be responsive to different screen sizes, especially desktop and tablet (mobile phone use is secondary, but a simplified view could be considered). The layout should allow the map to take a large portion of the screen, with sidebars or panels for lists and details.
* **Navigation & Layout:** A likely layout is a main map view with an overlay menu or sidebar. For instance, a sidebar that can slide out to show the list of plants or upcoming tasks, and hide to give full map view. Key UI components:
* **Toolbar/Menu:** for actions like adding a plant, toggling layers (plants, zones, tasks), switching to calendar view, etc.
* **Sidebar Panels:** for plant list or task list or settings.
* **Detail Pop-ups:** clicking a plant or task opens a modal or side panel with detailed information and actions (edit, delete, mark task done).
* **Shading for Zones:** As noted, when the user draws an area on the map (like a flower bed or lawn section), the UI will fill it with a translucent color. The color and a legend (or hover label) will identify what that zone is. This shading helps visually separate different areas of the garden on the map.
* **Performance & Smooth Interaction:** Even with many markers or shapes (potentially dozens of plants, plus a grid or path for mowing), the app should remain smooth. This implies using Canvas or WebGL layers for the map if needed, and optimizing how often re-renders happen. The map interactions (zoom, pan) should feel fluid and the overlays should adjust accordingly.
* **No GPS Requirement:** The user does not need live GPS tracking. All interactions are manual or based on the static map. This simplifies the interface – for example, there’s no “current location” dot needed. The app is focused on planning rather than realtime tracking.

### 6. Data Management and Offline Use

* **User Accounts:** Since this app is “just for me” (the user), initially it may be single-user with a simple login. However, to protect the data and allow access from any device via web, a login system with authentication is needed. A single user account (or ability to create one account) can be implemented. In future, if sharing is needed, user roles could be expanded.
* **Data Storage:** All data (plant records, tasks, map drawings, etc.) will be stored in a MongoDB database. The schema might include collections for Users, Plants, Tasks, and perhaps one for MapAreas (shapes/polygons drawn). Using MongoDB is suitable given the app will store document-like records (each plant with various attributes, each task with details, etc.). MongoDB’s geospatial capabilities could be tapped if we store coordinates for markers (e.g., to query what plants are in a certain area, though in a personal app this is minor).
* **Cloud vs Local:** The app will be online-accessible, meaning the backend (Node.js server and the database) will likely be hosted (could be on a cloud platform or a home server with port forwarding). Online access is fine and expected. There is **no requirement for offline functionality**, so the app can assume an internet connection for loading the map tiles and making any API calls (Google Maps, plant info API).
* **Backup & Export:** As a personal data store of one’s garden, it would be good to allow exporting the data (e.g., export all plant data to CSV or JSON, or even export the map as an image/PDF). This ensures the user “owns” their data and can have a backup outside the app. Regular backups on the server side are also advisable.
* **Security & Privacy:** Even though this is a personal app, basic security practices apply. User authentication should be secure (store hashed passwords, use HTTPS). The Google Maps API key and any plant API keys should be kept server-side or secured so they aren’t exposed publicly. As the data is personal (and not highly sensitive, mostly garden info), privacy concerns are minimal, but the app should not share data without user’s consent.

## Technical Requirements

### Tech Stack

* **Backend:** Node.js (with Express or a similar framework) will serve as the backend API. It will handle user authentication, serve the frontend (if using server-side rendering or just serve static files for an SPA), and provide endpoints for data (e.g., CRUD operations for plants and tasks). Business logic like generating the mowing route algorithm will also run on the server (could also be done client-side, but server might be better for heavy computation if needed).
* **Database:** MongoDB will store the application data. A document-oriented approach fits well: each plant entry as a document, each task as a document, etc. MongoDB’s geospatial fields could store coordinates of plant markers or polygon vertices if needed. We will use Mongoose (ODM) for schema definitions and data access in Node.
* **Frontend:** A single-page application (SPA) using a modern framework (e.g., React, Angular, or Vue). Given the mention of Fluent UI, using **React** with Fluent UI component library is a strong choice (Microsoft’s Fluent UI library provides React components with the Fluent design aesthetic). React can manage the state of the app (plants, tasks, map data) and interact with the Google Maps API through a library or direct JS API calls.
* **Mapping Library:** The Google Maps JavaScript API will be integrated for the map. Alternatively, we could consider open source options like Leaflet or Mapbox with Google or satellite tiles, but Google Maps is explicitly mentioned. We will use it for map imagery and possibly its drawing tools. The app will fetch map tiles and display markers and custom overlays for shapes. Google’s API will also allow reverse-geocoding if needed (not critical here) and measuring distances.
* **External APIs:** Integration with a plant information API (such as APIFarmer Plant API or Trefle). We will need to call this API when the user searches for or adds a plant, to fetch details. This requires handling API keys and possibly caching the results in our database (to avoid hitting rate limits repeatedly for the same plant species).
* **DevOps and Deployment:** In a personal project context, the app could be hosted on a service like Heroku, Vercel, or a personal server. Node.js and MongoDB could be hosted via services like MongoDB Atlas for convenience[[5]](https://medium.com/@nilima.sust/how-to-build-an-interactive-google-maps-store-locator-with-javascript-node-js-and-mongodb-ef7161c0420c#:~:text=Section%204%3A%20Creating%20the%20MongoDB,Database). We’ll ensure the environment is properly configured with the necessary API keys and connection strings.

### Integration and Data Flow

* **Google Maps Integration:** We’ll obtain a Google Maps API key and include the Maps JS script in the frontend. The app initializes the map centered on the user’s address or coordinates. Custom overlays (for drawn zones) might use Google Maps API’s own drawing library or we can capture clicks and draw on an HTML canvas overlay. Markers for plants will likely be Google Maps Marker objects, stored by their coordinates. The user’s drawing of polygons can be done via the Google Maps Drawing Manager (which provides UI for drawing shapes) or via a custom UI. We need to persist any drawn shapes (polygon vertices) to the database so they remain on the map in future sessions.
* **Plant Info API:** When adding a plant, after the user selects a species (either from a dropdown or after typing the name), the backend Node server will call the external Plant API to retrieve info such as scientific name, ideal conditions, etc. This info is returned and shown to the user and saved in MongoDB in the plant’s document. To avoid making the user wait on each load, we might fetch data once and cache it. With 100,000+ species info available via API[[4]](https://apifarmer.com/plant-database-api/#:~:text=Plant%20Database%20API%20gives%20you,to%20meet%20almost%20anyone%E2%80%99s%20needs), we have a broad coverage for whatever the user has in their garden.
* **Task Scheduling:** The backend can use a scheduler (like Node-cron or Agenda) to check for due tasks and send emails or push notifications. However, since it's single-user, a simpler approach is to generate reminders when the user opens the app (the frontend could highlight overdue tasks). For push notifications, we could use the browser’s Push API if the user permits, or send an email using an email service.
* **Algorithm for Mowing Path:** This could be implemented either in the frontend (e.g., a JavaScript function that takes the polygon of the lawn and computes path lines) or in the backend (where we might do more complex computation without blocking UI). A simple approach: divide the lawn polygon into parallel stripes the width of the mower deck, using a heuristic to minimize turns. The computed path (sequence of GPS coordinates or grid lines) is then sent to the frontend to display. Since no live GPS is used, we won’t track the mower in real-time – the path is just a static plan for the user to follow.
* **Scalability:** Though the app is for personal use, using a solid tech stack means it could be expanded to more users or larger gardens. Node and MongoDB are capable of handling large data if needed, and the Google Maps interface can handle many markers (though one acre likely won’t have more than a few hundred points of interest, which is fine). If ever needed, the architecture could be expanded to multi-user with minimal changes (just adding user IDs to data records, etc.).

## Non-Functional Requirements and Constraints

* **Platform:** Web application, tested on modern browsers (Chrome, Firefox, Safari, Edge). No native mobile app for now, and no offline mode requirement. The user must have internet access to use the app (for map tiles and any API calls).
* **Performance:** The app should load the initial map and data within a few seconds on a typical home internet connection. Interactions like adding a marker or drawing a zone should feel immediate. Heavy operations (like computing a complex route) should be done in a way that doesn’t freeze the UI – possibly showing a “calculating route” message if it takes more than a second.
* **Usability:** Since this is for a single user (likely the app creator themselves), the UI can be optimized to their needs. However, general good usability practices apply: the app should be intuitive, with help tips or documentation for complex features (like how to draw zones or interpret the mowing route).
* **Reliability:** Data (plant logs, tasks) should be stored reliably. Using a managed DB (like MongoDB Atlas) helps ensure uptime. Regular backups or export capability protects against data loss. The Node backend should handle errors gracefully (e.g., if the plant API is down, show an error message but allow the user to input details manually).
* **Privacy/Security:** The map data will include the location of the user’s home/garden. This is sensitive info, so the app must ensure it’s not publicly accessible. The application should enforce login before showing any data. All communication should be over HTTPS to protect data in transit (especially if the app is hosted online). Since it’s personal, compliance with regulations like GDPR is not a major concern, but the general principle of user control over data applies.
* **Cost Considerations:** Google Maps API usage is free up to a limit (there is a monthly free credit), and for a single user mapping one property, it should stay well within free tier. Similarly, some plant info APIs are free for low volumes or have generous limits for non-commercial use (e.g., Trefle had a free tier). We should track API usage and ensure the app doesn’t silently incur large costs. Caching data like plant info can reduce external API calls. If usage grows (unlikely for personal use), we might consider open-source data to avoid fees.

## Future Enhancements

While the initial version focuses on core functionality with straightforward implementations, the following enhancements are envisioned for later phases: - **Machine Learning for Care Recommendations:** In a future update, incorporate machine learning to analyze the user’s garden data and possibly external factors (weather, season) to give smart recommendations. For instance, an ML model could predict the optimal watering schedule by learning from weather forecasts and soil data, or suggest plants that would thrive in uncovered spots of the garden based on conditions and the success of existing plants. - **Visual Plant Identification:** Add a feature where the user can take a photo of a plant or weed in their garden and the app identifies it (leveraging an AI service or model). This can help in cataloging unknown plants or detecting diseases. There are existing AI plant identification APIs that could be integrated down the line. - **Enhanced Mowing and Pathfinding:** Use more advanced algorithms or even computer vision for the mowing feature. For example, if drone imagery or detailed maps are available, the app could optimize patterns around very irregular obstacles better than a simple stripe approach. Machine learning could potentially optimize the path based on prior mowing data (if the user tracks how long it took or where they had issues). - **Community and Sharing Features:** If the app ever extends beyond personal use, allow users to share their garden map or plant data with others (friends, or an online community of gardeners). This could lead to features like comparing schedules, exchanging tips for specific plants, or showing off garden layouts. - **IoT Integration:** Connect with smart garden devices – for example, automatic irrigation systems, soil moisture sensors, or robot lawn mowers. The app could use sensor data to update task recommendations (e.g., skip watering if soil moisture is high) or send commands (e.g., tell a robotic mower the planned route). - **Mobile App or PWA:** Wrap the web app as a Progressive Web App, so it can be installed on mobile devices and possibly used offline for viewing data (still would need connectivity for maps unless offline maps are saved). - **Multi-Property Support:** If the user has more than one garden or moves to a new house, allow managing multiple garden profiles within the app.

Each of these future ideas can be added incrementally, and the architecture laid out (Node/MongoDB, modular frontend) will support expansion. However, the priority for now is to deliver a robust, usable solution for the single property and user at hand.

## Conclusion

This app will serve as a comprehensive digital **garden companion** for the user, combining mapping, record-keeping, and intelligent planning tools in one place. By using a modern tech stack (Node.js, MongoDB, React with Fluent UI) and integrating reliable external services (Google Maps for visualization and plant databases for information), we ensure that the product is both technically feasible and aligned with the user's ambition. The result will be a personalized garden management platform that not only documents the garden’s current state but also helps in efficiently maintaining and improving it for the future. The inclusion of an optimized mowing route planner showcases the potential for smart features, and upcoming enhancements will continue to transform routine gardening into a more **informed, efficient, and enjoyable** experience for the user. The PRD above will guide the development team in building the app to meet these requirements and delight our garden enthusiast user.

[[1]](https://www.reddit.com/r/selfhosted/comments/169o20h/introducing_plantit_a_selfhosted_open_source/#:~:text=only%20one%20in%20charge%20of,since%20last%20action%20on%20them) Introducing Plant-it - A self-hosted, open source gardening companion app : r/selfhosted

<https://www.reddit.com/r/selfhosted/comments/169o20h/introducing_plantit_a_selfhosted_open_source/>

[[2]](https://executivelawncare.net/2025/05/12/best-mowing-pattern-for-riding-mower/#:~:text=1.%20) Best Mowing Pattern For Riding Mower - Executive Lawn Care

<https://executivelawncare.net/2025/05/12/best-mowing-pattern-for-riding-mower/>

[[3]](https://medium.com/@nilima.sust/how-to-build-an-interactive-google-maps-store-locator-with-javascript-node-js-and-mongodb-ef7161c0420c#:~:text=This%20comprehensive%20blog%20post%20will,customization%20store%20locator%20from%20scratch) [[5]](https://medium.com/@nilima.sust/how-to-build-an-interactive-google-maps-store-locator-with-javascript-node-js-and-mongodb-ef7161c0420c#:~:text=Section%204%3A%20Creating%20the%20MongoDB,Database) How to Build an Interactive Google Maps Store Locator with JavaScript, Node.js, and MongoDB | by Nilima Das | Medium

<https://medium.com/@nilima.sust/how-to-build-an-interactive-google-maps-store-locator-with-javascript-node-js-and-mongodb-ef7161c0420c>

[[4]](https://apifarmer.com/plant-database-api/#:~:text=Plant%20Database%20API%20gives%20you,to%20meet%20almost%20anyone%E2%80%99s%20needs) Plant Database API | Plant Taxonomy | Plant Characteristics & Data

<https://apifarmer.com/plant-database-api/>