ChatGPT

Development Plan and Task Breakdown

Directory Pages

Create a new `directory_pages` model/table to hold custom directory pages with SEO metadata and configuration. Each directory page should store a custom title, meta keywords, meta description, OG image, and fields for *subdomain* and *subdirectory* (allowing independent values and a 301 redirect between them). Add a boolean "location-agnostic" flag to indicate whether to show a map of listings. Directory pages must be linked to categories and locations (postal codes or cities) via join tables, so a single page can cover multiple categories or locations as needed. (This allows, for example, grouping "Jobs" and "Real Estate" on one page if desired.) SEO best practices emphasize optimizing category or directory pages for search visibility [1], so ensure each page's metadata fields can be edited.

- **Database:** Add a `directory_pages` table with fields: `title`, `meta_keywords`, `meta_description`, `og_image_url`, `subdomain`, `subdirectory`, `location_agnostic` (bool), etc. Create join tables `directory_page_categories` (links `directory_pages` to `categories`) and `directory_page_locations` (links to `postal_codes` or `cities`). Each join table allows many-to-many links.
- **Admin UI:** Build a form to create/edit directory pages. Include multi-select inputs for categories and for location filters (postal codes or cities). Include fields for title/meta tags, subdomain and subdirectory, and the location-agnostic checkbox.
- **Routing:** Implement logic so that if a directory has a subdomain, requests to it 301-redirect to the corresponding subdirectory (or vice versa, per site config). Ensure generated URLs use the chosen domain/subdirectory.
- **Front-End Rendering:** When displaying a directory page, query all listings matching its categories and locations. If not location-agnostic, show a map (e.g. using Leaflet or Google Maps) with pins for each listing's address. Add a pin icon next to each listing's title in the list; clicking the icon recenters the map on that listing's location. If location-agnostic, omit the map.
- **SEO Integration:** Use each directory page's custom title and meta fields to populate the HTML `<title>`, meta tags, and Open Graph tags on the page. By giving directory pages unique optimized metadata, we improve their search visibility [1].

Listings and Address Data Structure

Redesign listings and location tables for flexibility. Each **Listing** can belong to multiple categories (many-to-many via a join table) and can have multiple physical addresses. Create a separate `physical_addresses` table linked to `listings` (one-to-many). In `physical_addresses`, store street, suite/unit, a foreign key to `cities` (or `state`/`country` if needed), and a foreign key to `postal_codes`. Each address row will also hold `latitude` and `longitude`. If a listing has multiple addresses, each gets its own entry in `physical_addresses`; if an address spans multiple units, represent each unit as a separate address row tied to the same listing.

- **Database:**
- Keep a `categories` table. Create a `listing_categories` join table linking `listings` and `categories` to support multi-category listings.
- Create `physical_addresses` table with columns: `id`, `listing_id` (FK), `street_address`, `suite`, `city_id`, `state_id`, `country_id`, `postal_code_id`, `latitude`, `longitude`. Enforce that each address belongs to exactly one listing.
- Modify `listings` table: remove single-address fields and instead have an association to `physical_addresses`. Also add a default `city_id` and `postal_code_id` on `listings` (nullable) so a listing can optionally reference a city or postal code directly if no specific address.
- **Location Hierarchy Tables:** Preload global location data. Import countries, states/provinces, and cities from the [dr5hn Countries-States-Cities database] [2]. Each city row should have `state_id` (nullable) and `country_id`. Each state/province has `country_id`. Handle countries without states by allowing `state_id` to be null or by creating a dummy state entry. For postal codes, import GeoNames postal code files [3]. The `postal_codes` table should have columns: `postal_code`, `city_id`, possibly `state_id`, `country_id`, and center `latitude`, `longitude`. Nearly 100 countries are supported in GeoNames files [3].

Country/State/City/Postal Structure

Design a hierarchical location schema:
- **Country** table: basic country info (name, iso2, etc). Add a column or flag indicating if the country uses states/provinces or not, and if it uses postal codes or not.
- **State/Province** table: columns `id`, `name`, `code`, `country_id`. States link to countries. If a country has no states (as many small countries do), either leave states empty or create a placeholder "state" for consistency.
- **City** table: columns `id`, `name`, `state_id` (nullable), `country_id`. If a country has no states, `state_id` can be null and `country_id` used directly.
- **PostalCode** table: columns `id`, `postal_code`, `city_id`, `state_id` (nullable), `country_id`, `latitude`, `longitude`. Each code links to one city. For countries with no postal codes, this table will be empty.
- **Listing Location Fields:** Each listing (or address) should have foreign keys `postal_code_id` and `city_id`. If a listing has a postal code, use that (and implicitly its city); otherwise, set `postal_code_id` null and use the city alone. This accommodates "city-only" locations for postal-less countries as requested.

Populate these tables via one-time scripts/migrations: clone or download the [countries-states-cities database on GitHub] and load the PSQL data [2]. Download GeoNames postal code `.zip` files and import relevant fields into `postal_codes` [3].

Geocoding Integration

On saving or updating a listing with an address, automatically geocode its location to get latitude/longitude. First call the [geocode.maps.co API] using the `GEOCODEMAPS_API_KEY`; if that fails or returns no result, fall back to the Google Geocoding API with `GOOGLEMAPS_API_KEY`. (We avoid MapQuest as it requires billing.) Store the resulting lat/lng in the `physical_addresses` record (or in the listing if using a single address). Google's Geocoding API "converts addresses into geographic coordinates (latitude and longitude)" [4], and geocode.maps.co offers a free tier for forward geocoding [5] [6].

- **Implementation:**
- In the backend, hook into the listing/address creation or update logic. If the address fields change and an address is present, asynchronously request `https://geocode.maps.co/search?q=ADDRESS&api_key=...`. Parse the JSON response for `lat` and `lon`. If no result, call Google's Geocoding (e.g. via their REST API) as fallback [4].
- Populate `latitude` and `longitude` in the `physical_addresses` table (and optionally copy to `listings` if having a main coordinate).
- Ensure the environment variables `GEOCODEMAPS_API_KEY` and `GOOGLEMAPS_API_KEY` are loaded (per the Makefile/SOPS configuration mentioned).
- Handle errors/logging: if both services fail, record the listing's geocoding status so it can be fixed manually.

API Endpoints

Expose CRUD endpoints for the new entities. Specifically, build REST APIs (or GraphQL) for:

- **Listings:** Create, read, update, delete a listing. Support multi-part listing data including multiple addresses and category tags. When creating/updating, allow an array of category IDs and an array of address objects.
- **Physical Addresses:** Endpoints to add/edit/delete a listing's addresses (if not handled inline in Listings API).
- **Categories:** CRUD for categories (name, slug, parent category if needed).
- **Directory Pages:** CRUD for directory pages. Include endpoints to associate categories and locations to a directory page.
- **Lists (User Collections)** – Phase 2: endpoints to create/edit/delete a user's named list, and to add or remove listings from a list.
- **Votes and Reviews (Phase 2):** endpoints to submit a vote (up/down) or a review for a listing.
- **Reports (Phase 2):** endpoint to file a "report dead listing" on a listing.

Do not expose endpoints for static location tables (country/state/city/postal) since these are preloaded. Use appropriate authentication/permissions on endpoints (e.g. only admins can CRUD categories or directories). Document each API route and expected JSON schema.

Crawler and Data Ingestion

Refactor and extend the data collection system to handle various input formats. Implement a script (or set of scripts) with modes for input type: HTML, CSV/JSON, or raw text. The goal is a two-stage pipeline:

1. **Data Extraction Script:**
2. *Inputs:* Accept a file path or URL and a specified format flag (`--format html|json|csv|text`).
3. *HTML/CSV/JSON:* If the input is HTML, parse it with BeautifulSoup (or similar) to extract listing fields (title, description, URL, address, etc). If CSV/JSON, parse each row/object to the same field structure. Implement site-specific parsers as needed.

4. *Raw Text:* If format is text, send the raw content to an LLM (via OpenRouter or another available endpoint) with a prompt to extract listings. The LLM should output structured data (e.g. JSON) by identifying URLs, titles, descriptions, addresses, categories, etc. As recent workflows show, LLMs can "transform HTML (unstructured data) into structured data" by following a prompt [7], which solves varied or messy sources.
5. *Output:* Write a **basic data JSON file** containing an array of extracted listings with fields (title, description, address parts, category hints, etc). Do not yet call the database.

6. *Reuse Modules:* Move any existing description-generation or text-cleanup code into shared modules so both crawler and later scripts can use them.

7. **Enrichment and Import Script:**

8. *Input:* Take the JSON file from stage 1. For each record, call an LLM (e.g. free OpenRouter models like Dolly or Mistral) to improve/standardize fields: generate a concise short description and a longer description, refine titles, and possibly classify subcategories. For example, prompt: "Given this raw listing data, output a JSON with improved title, short and long descriptions, and category IDs." LLMs have been successfully used to structure and clean scraped web data [7].
9. *Output:* Use the final JSON to insert into the database via the API. For each entry, make API calls to create the listing, its addresses, and category links. This two-step approach (parse then enrich) allows review/edit of the basic data JSON before final import.

Crawler Details

- **BeautifulSoup:** Use Requests/Selenium + BeautifulSoup for standard sites (common HTML patterns).
- **LLM Parsing:** For unstructured or inconsistent sources, send content to an LLM (prompt carefully about expected fields). Ensure output format validity (e.g. a JSON schema) and implement retries or validation if the LLM output is malformed. As one analysis notes, feeding HTML text to a well-instructed LLM can extract the desired fields, reducing brittle scraper maintenance [7].
- **Modularity:** Code should be modular: separate the parsing logic from the description-generation logic. For example, have a shared `describeListing(raw_data)` function that either script can use.
- **OpenRouter Models:** Select free models via OpenRouter (deepseek-chat or Llama, etc) for cost efficiency. Validate their output quality and handle limits.

Future User Accounts (Phase 2)

Plan for an opt-in user account system with frictionless registration (like Reddit's). Design tasks but implement after core features.

- **User Model:** Create a `users` table with: `id`, `unique_token` (randomly generated on account stub creation), `username` (nullable, unique), `email` (nullable, unique), `password_hash` (nullable), and timestamps. The `unique_token` lets users resume a session without traditional login.
- **Auto-Generated Stubs:** When a user first performs an action requiring login (e.g. saving a listing), auto-create an account stub with a random token and generated username. Present the token and

username to the user (e.g. via onscreen prompt). They should store these to log back in. Optionally offer them to set an email/username/password now or later.
- **Login Flows:** Always allow “magic link” or token-based login: user enters their email or username, then you send a one-time code or link to their email (passwordless authentication [8]). If they have set a password, also allow password login. Support sending an email with a login link or code that “completes the authentication flow” [8]. After login, give users a dashboard to optionally choose/change username, email, and to set a permanent password. Do not force all steps upfront – users can remain with just a token if they prefer.
- **Collections (“Lists”):** Allow users to create named collections of listings.
- **Database:** Create `lists` table: `id`, `user_id`, `name`, `slug`, `created_at`. Also a join table `list_items` (`list_id`, `listing_id`).
- **UI:** Add an “Add to list” icon/button on each listing. Clicking it expands a panel showing the user’s lists (each with a checkbox showing membership) and a field to create a new list. Clicking a checkbox adds/removes the listing via AJAX; creating a new list entry triggers its creation and adds the listing.
- **Sharing:** Lists should be accessible via URLs like `/users/{username}/{list_slug}`. If usernames not set, use the generated username or token. On the list page, show all listings in that collection. Provide options to download the list as CSV or KML: export listing data (including coordinates) into those formats for mapping or analysis.
- **Voting:** Enable upvote/downvote on listings.
- **Database:** `votes` table: `user_id`, `listing_id`, `vote_type` (+1 or -1). Enforce one vote per user per listing (update existing instead of duplicate).
- **UI:** Add up/down arrows or a toggle on each listing. When clicked, send an AJAX request to submit the vote. Update the listing’s score display immediately.
- **Reviews:** Allow written reviews (optionally with photos).
- **Database:** `reviews` table: `id`, `user_id`, `listing_id`, `rating` (optional, e.g. star 1–5), `comment`, `created_at`. `review_images` table: `id`, `review_id`, `image_url`.
- **UI:** On a listing page, logged-in users can submit a review form (comment, optional rating, upload images). Images are uploaded to storage and linked. Display all reviews under the listing with pagination.
- **Reporting:** Users can flag a listing as dead/nonexistent.
- **Database:** `reports` table: `id`, `user_id`, `listing_id`, `reason` (optional text), `created_at`.
- **UI:** Add a “Report Dead” button on listings. Clicking sends a report via AJAX. Provide an admin interface to review and remove or update reported listings.

These user features are for a later phase, but planning them avoids redesigning core models later.

Points Needing Clarification

All requested features are technically feasible, but a few areas may need more detail:

- **Subdomain vs Subdirectory:** The spec allows both a subdomain and a subdirectory for a directory page. Clarify the desired behavior (e.g. should one always redirect to the other?). Implementation could simply enforce a canonical structure, but decide whether the primary path is the subdomain or the directory URL.

- **Postal-Code and City Fallback:** The approach for cities without postal codes is “city-only”. Confirm: will listings in such countries have `postal_code_id = NULL` and rely solely on `city_id`? How to query/filter these listings versus postal-coded listings? Ensure the schema allows queries like “all listings in city X” covering both cases.

- **Multiple Addresses:** If a listing has multiple addresses, should all be displayed on the listing detail page, and should the listing appear under each relevant location-based directory page? (Likely yes.) Plan how to handle multiple map pins for one listing if needed.

- **LLM Prompts and Limits:** Defining the prompts for raw text parsing and enrichment will require iteration. Also consider rate limits or token limits (e.g. breaking very long HTML into pieces).

- **Environment Setup:** Ensure the build system (Makefile/SOPS) loads the new `GEOCODEMAPS_API_KEY` and `GOOGLEMAPS_API_KEY` variables correctly. This is part of dev setup but should be tested.

If any of these points are unclear, please advise before development begins so we can adjust the design.

Sources: We’ll rely on existing data packages for location tables [2] [3], public geocoding APIs [5] [4], and recent examples of using LLMs for web data parsing [7] [8].

---

[1] 9 SEO Tips for Category Pages To Boost Your Rankings
https://www.seo.com/basics/content/category-pages/

[2] GitHub - dr5hn/countries-states-cities-database: Discover our global repository of countries, states, and cities! Get comprehensive data in JSON, SQL, PSQL, SQLSERVER, MONGODB, SQLITE, XML, YAML, and CSV formats. Access ISO2, ISO3 codes, country code, capital, native language, timezones (for countries), and more. #countries #states #cities
https://github.com/dr5hn/countries-states-cities-database

[3] GeoNames
http://download.geonames.org/export/zip/

[4] Geocoding API overview | Google for Developers
https://developers.google.com/maps/documentation/geocoding/overview

[5] [6] Free Geocoding API - Geocode Addresses & Coordinates
https://geocode.maps.co/

[7] Enhancing Web Scraping With Large Language Models: A Modern Approach | by Nacho Corcuera Platas | Medium
https://medium.com/@ignacio.cplatas/enhancing-web-scraping-with-large-language-models-a-modern-approach-6216d5bba8d5

[8] What is Magic Link Login? How it Works
https://www.pingidentity.com/en/resources/blog/post/what-is-magic-link-login.html