# Propertifi - Backend Improvement & Deployment Plan

This document provides a detailed plan for implementing the backend features outlined in improvement.md (v2) for the Propertifi Laravel application. It includes prompts suitable for AI coding assistants like Google's Gemini or Anthropic's Claude.

**Assumptions:**

* The project is based on the Laravel framework (version 8+ inferred).
* Database is MySQL or similar relational DB.
* AI Agent has access to the project's codebase context.
* Existing Admin panel uses Blade views and potentially some JS frameworks.
* A separate Next.js frontend is being developed concurrently.

**Overarching Goals:** Implement new features (Owner Dashboard, Resources), enhance core logic (Lead Matching), improve code quality, provide necessary APIs for the frontend, and prepare for scalable deployment.

## **PHASE 1: Foundational Enhancements & Core Features (Backend)**

### **Step 1.1: Enhance Lead Model & API Validation (Align with Frontend Form)**

* **Goal:** Update backend to accept potentially new lead fields from the enhanced frontend form and strengthen validation.
* **Key Files:** app/Models/Leads.php, database/migrations/...create\_leads\_table.php (if modification needed), app/Http/Requests/StoreLeadRequest.php (New or existing), app/Http/Controllers/Api/HomePageController.php.
* **Prompt 1.1.1 (Backend - Model/Migration):**  
  Review the `Leads` model and its migration. Based on the planned multi-step form (including conditional questions for property details like 'number\_of\_units', 'square\_footage', 'occupancy\_status', 'services\_needed'), determine if new columns need to be added to the `leads` table.  
  - If new columns are needed, generate a new migration using `php artisan make:migration add\_property\_details\_to\_leads\_table --table=leads` to add them (e.g., `num\_units` (integer, nullable), `sq\_footage` (integer, nullable), `occupancy` (string or enum ['occupied', 'vacant'], nullable), `services\_needed` (JSON or TEXT, nullable)).  
  - Update the `$fillable` array in the `Leads` model to include any new attributes.  
  - Run the migration: `php artisan migrate`.
* **Prompt 1.1.2 (Backend - Validation):**  
  Create or update the Form Request class `StoreLeadRequest` (`php artisan make:request StoreLeadRequest` if new) used by the `storeLead` method in `Api/HomePageController`.  
  - Add validation rules for any new fields from Prompt 1.1.1 (e.g., `num\_units` => `nullable|integer|min:1`, `services\_needed` => `nullable|array`).  
  - Implement conditional validation if necessary (e.g., `num\_units` => `required\_if:propertyType,multi-family|nullable|integer|min:1`). Define `propertyType` based on how it's submitted or inferred from questions.  
  - Strengthen existing rules (e.g., `zip` => `required|digits:5`, `state` => `required|string|size:2`).  
  - Ensure validation for dynamic question responses is handled, likely within the `HomePageController` after `StoreLeadRequest` passes. The controller should verify that submitted `question\_id`s exist and answers match expected formats (this might involve fetching questions and looping through responses). Add this response validation logic to the `storeLead` method.
* **Prompt 1.1.3 (Backend - Controller Update):**  
  Review and update the `storeLead` method in `app/Http/Controllers/Api/HomePageController.php`:  
  - Ensure it uses the `StoreLeadRequest` for validation.  
  - Correctly save new fields (like `num\_units`, `services\_needed`) to the `Leads` model instance.  
  - Refine or implement the logic for saving question `Responses`. Ensure each response is associated with the newly created `Lead` ID. Consider wrapping lead and response creation in a database transaction (`DB::transaction(...)`) for atomicity.  
  - Return a JSON response with the created lead ID and a success message on success (Status Code 201). FormRequest handles 422 errors automatically. Implement try-catch for other potential errors (e.g., database errors) and return a 500 error response.

### **Step 1.2: Add Question Grouping/Ordering Logic**

* **Goal:** Allow admins to define steps and order for questions in the multi-step form.
* **Key Files:** database/migrations/...create\_questions\_table.php, app/Models/Questions.php, app/Http/Controllers/Admin/QuestionsController.php, Admin Views (resources/views/admin/questions/).
* **Prompt 1.2.1 (Backend - Migration & Model):**  
  Generate a new migration `php artisan make:migration add\_step\_order\_to\_questions\_table --table=questions` to modify the `questions` table:  
  - Add an integer column named `step` with a default value of 1. Add an index to this column.  
  - Add an integer column named `order\_in\_step` with a default value of 0.  
  - Update the `$fillable` array in the `Questions` model to include `step` and `order\_in\_step`.  
  - Run the migration.
* **Prompt 1.2.2 (Backend - Admin UI):**  
  Update the `Admin/QuestionsController` and its views (`resources/views/admin/questions/\_form.blade.php`, `create.blade.php`, `edit.blade.php`):  
  - Add numeric input fields for 'Step' and 'Order within Step' in the form partial (`\_form.blade.php`).  
  - Ensure the `store` and `update` methods in `QuestionsController` handle saving these new `step` and `order\_in\_step` fields. Add them to the `create()` and `update()` calls.  
  - Update the `index.blade.php` view: Display the 'Step' and 'Order' columns in the table. Add links to the table headers to allow sorting by these columns (pass sort parameters in the request and update the controller's index query accordingly).
* **Prompt 1.2.3 (Backend - API Endpoint Update):**  
  Modify the API endpoint in `Api/HomePageController` that fetches questions for the lead form:  
  - Update the Eloquent query to order the questions using `orderBy('step', 'asc')->orderBy('order\_in\_step', 'asc')`.  
  - Ensure the `step` and `order\_in\_step` attributes are included in the JSON response for each question.

### **Step 1.3: Implement Conditional Question Logic (Backend Support)**

* **Goal:** Enable questions to be shown based on answers to previous questions.
* **Key Files:** database/migrations/...create\_questions\_table.php, app/Models/Questions.php, app/Http/Controllers/Admin/QuestionsController.php, Admin Views, app/Http/Controllers/Api/HomePageController.php.
* **Prompt 1.3.1 (Backend - Migration & Model):**  
  Generate a new migration `php artisan make:migration add\_dependency\_to\_questions\_table --table=questions` to modify the `questions` table:  
  - Add a nullable `unsignedBigInteger` column `depends\_on\_question\_id`. Optionally add a foreign key constraint: `$table->foreign('depends\_on\_question\_id')->references('id')->on('questions')->onDelete('set null');`.  
  - Add a nullable string column `depends\_on\_answer`.  
  - Update the `$fillable` array in the `Questions` model to include these fields.  
  - Optionally, define the relationship in the `Questions` model: `public function parentQuestion() { return $this->belongsTo(Question::class, 'depends\_on\_question\_id'); }`.  
  - Run the migration.
* **Prompt 1.3.2 (Backend - Admin UI):**  
  Update the `Admin/QuestionsController` create/edit form (`\_form.blade.php`):  
  - Add a 'Depends on Question' dropdown (`<select name="depends\_on\_question\_id">`). Populate it with all other questions (`Question::where('id', '!=', $currentQuestionId)->pluck('question\_text', 'id')`) plus a "None" option (value=""). Use JavaScript to potentially disable the 'Required Answer' field if 'None' is selected.  
  - Add a 'Required Answer' text input (`<input type="text" name="depends\_on\_answer">`). Explain that this should match the value expected (e.g., 'Yes', 'Option A').  
  - Update the `store` and `update` methods in the controller to correctly save `depends\_on\_question\_id` (handle the "None" case by saving null) and `depends\_on\_answer`.
* **Prompt 1.3.3 (Backend - API Endpoint Update):**  
  Modify the API endpoint in `Api/HomePageController` that fetches questions:  
  - Ensure the query fetches \*all\* relevant questions for the form (likely all active questions).  
  - Include the `depends\_on\_question\_id` and `depends\_on\_answer` attributes (even if null) in the JSON response for each question. The frontend requires this data to implement the show/hide logic. Maintain the sorting by step and order.

### **Step 1.4: Create State Law Content Model & Admin CRUD**

* **Goal:** Build the backend structure for managing structured legal content.
* **New Files:** database/migrations/...create\_state\_law\_contents\_table.php, app/Models/StateLawContent.php, app/Http/Controllers/Admin/StateLawController.php, Views under resources/views/admin/state-laws/.
* **Prompt 1.4.1 (Backend - Migration & Model):**  
  Generate a migration `php artisan make:migration create\_state\_law\_contents\_table` and corresponding Eloquent model `php artisan make:model StateLawContent`. In the migration, define the schema:  
  - `id` (primary key)  
  - `state\_code` (string, 2 chars, index)  
  - `topic\_slug` (string, index) - e.g., 'security-deposits', 'eviction'  
  - `title` (string)  
  - `summary` (text, nullable)  
  - `content` (longText)  
  - `official\_link` (string, nullable)  
  - `last\_updated` (date, nullable)  
  - `status` (string, default: 'draft', index) // Consider enum if DB supports: ['draft', 'published']  
  - `timestamps()`  
  - Add a composite unique constraint: `$table->unique(['state\_code', 'topic\_slug']);`  
  In the `StateLawContent` model, define `$fillable` attributes for mass assignment.  
  Run the migration.
* **Prompt 1.4.2 (Backend - Admin CRUD Controller):**  
  Generate a resourceful controller `php artisan make:controller Admin/StateLawController --resource --model=StateLawContent`.  
  - Implement standard CRUD methods (index, create, store, edit, update, destroy) using route model binding for `StateLawContent`.  
  - Apply admin authentication middleware (`middleware('auth:admin')` or similar).  
  - In the `index` method, retrieve records with pagination (`paginate(20)`), allow filtering by `state\_code` and `status` via request parameters.  
  - Generate Blade views (`index.blade.php`, `create.blade.php`, `edit.blade.php`, `\_form.blade.php`) within `resources/views/admin/state-laws/`. Use the existing admin layout.  
  - The `\_form.blade.php` should include:  
   - Select dropdown for `state\_code` (populate with US states array/helper).  
   - Text input for `topic\_slug`.  
   - Text input for `title`.  
   - Textarea for `summary`.  
   - Rich text editor instance (like CKEditor/TinyMCE, assumed available in admin layout) for `content`.  
   - URL input for `official\_link`.  
   - Date input (HTML5 or JS datepicker) for `last\_updated`.  
   - Radio buttons/Select for `status` ('draft', 'published').  
  - Add the resource route to `routes/admin.php`: `Route::resource('state-laws', StateLawController::class);`.

### **Step 1.5: Create Public Routes/Controller for State Laws**

* **Goal:** Make the structured legal content accessible via the web (for SEO and users).
* **New Files:** app/Http/Controllers/PublicLawController.php, Views resources/views/public/laws/index.blade.php, resources/views/public/laws/show.blade.php.
* **Prompt 1.5.1 (Backend - Controller & Routes):**  
  Create a controller `php artisan make:controller PublicLawController`.  
  - Add an `index` method: Fetch distinct `state\_code`s from `state\_law\_contents` where `status` is 'published'. Create an array mapping state codes to full names and slugs (e.g., ['CA' => ['name' => 'California', 'slug' => 'california']]). Pass this array to the view.  
  - Add a `show($state\_slug)` method: Find the state code corresponding to the `$state\_slug`. Fetch all `StateLawContent` records where `state\_code` matches and `status` is 'published'. Order them logically (e.g., by a predefined topic order, or alphabetically by title). Group the results by `topic\_slug` (or fetch distinct topics first, then content for each). Pass the state name and grouped/ordered content to the view. If state slug is invalid or no content exists, abort(404).  
  - Add routes in `routes/web.php`:  
   - `Route::get('/laws', [PublicLawController::class, 'index'])->name('public.laws.index');`  
   - `Route::get('/laws/{state\_slug}', [PublicLawController::class, 'show'])->name('public.laws.show');`
* **Prompt 1.5.2 (Backend - Views):**  
  Generate basic Blade views for the public law pages (create a simple public layout if one doesn't exist: `resources/views/layout/public.blade.php`):  
  - `resources/views/public/laws/index.blade.php`: Extend the public layout. Iterate through the states array passed from the controller. Display each state name, linking to `route('public.laws.show', ['state\_slug' => $state['slug']])`. Consider adding a simple US map SVG with links.   
  - `resources/views/public/laws/show.blade.php`: Extend the public layout. Display `<h1>{{ $stateName }} Landlord-Tenant Laws</h1>`. Iterate through the topics/content passed from the controller. For each topic, display `<h2>{{ $content->title }}</h2>`, `{!! $content->content !!}` (ensure content is sanitized if needed, though admin input should be trusted), display `official\_link` and `last\_updated` date. Add intra-page navigation links for topics. Add relevant CTAs. \*Crucially, apply Tailwind CSS classes here to match the visual style of the Next.js frontend.\*

### **Step 1.6: Create Document Template Model & Admin CRUD (incl. File Upload)**

* **Goal:** Backend structure for managing downloadable document templates.
* **New Files:** database/migrations/...create\_document\_templates\_table.php, app/Models/DocumentTemplate.php, app/Http/Controllers/Admin/DocumentTemplateController.php, Views under resources/views/admin/document-templates/.
* **Prompt 1.6.1 (Backend - Migration & Model):**  
  Generate migration `php artisan make:migration create\_document\_templates\_table` and model `php artisan make:model DocumentTemplate`. Define the schema:  
  - `id`  
  - `title` (string)  
  - `description` (text, nullable)  
  - `state\_code` (string, 2 chars, default 'US', index) // 'US' for federal/generic  
  - `category` (string, index) // e.g., 'Lease Agreement', 'Eviction Notice'  
  - `file\_path` (string) // Stores path relative to a storage disk  
  - `original\_filename` (string, nullable)  
  - `mime\_type` (string, nullable)  
  - `is\_free` (boolean, default true)  
  - `requires\_signup` (boolean, default false) // Requires owner login later  
  - `status` (string, default: 'draft', index) // e.g., ['draft', 'published']  
  - `download\_count` (unsignedInteger, default 0)  
  - `timestamps()`  
  In the model, set `$fillable` attributes.  
  Run the migration.
* **Prompt 1.6.2 (Backend - Filesystem Disk):**  
  Configure a private filesystem disk for storing templates securely. In `config/filesystems.php`, add a new disk configuration under `disks`:  
  'templates': [  
   'driver' => 'local',  
   'root' => storage\_path('app/templates'), // Store outside /public  
   'visibility' => 'private',  
  ],  
  Ensure the `storage/app/templates` directory exists and is writable by the web server. Add `storage/app/templates/\*` to `.gitignore` if not already covered.
* **Prompt 1.6.3 (Backend - Admin CRUD Controller & File Handling):**  
  Generate `Admin/DocumentTemplateController` (`php artisan make:controller Admin/DocumentTemplateController --resource --model=DocumentTemplate`).  
  - Implement CRUD methods with admin auth middleware.  
  - \*\*Store Method:\*\*  
   - Validate request data including the uploaded file (`template\_file => required|file|mimes:pdf,doc,docx|max:5120` // 5MB example).  
   - If validation passes, store the file: `$path = $request->file('template\_file')->store('', 'templates');` // Store on 'templates' disk.  
   - Create the `DocumentTemplate` record, saving `$path` to `file\_path`, and also store `$request->file('template\_file')->getClientOriginalName()` and `$request->file('template\_file')->getMimeType()`.  
  - \*\*Update Method:\*\*  
   - Validate request data (`template\_file` is `nullable` here).  
   - If a new file is uploaded: validate it, find the existing template, delete the old file (`Storage::disk('templates')->delete($template->file\_path)`), store the new file, and update `file\_path`, `original\_filename`, `mime\_type` along with other fields.  
   - If no new file, just update other fields.  
  - \*\*Destroy Method:\*\*  
   - Find the template.  
   - Delete the associated file: `Storage::disk('templates')->delete($template->file\_path);`  
   - Delete the database record.  
  - Implement `index` view with pagination/filtering.  
  - Create form views (`\_form`, `create`, `edit`) including `<input type="file" name="template\_file">`, state dropdown, category input, status select, and toggles for free/signup. Ensure forms have `enctype="multipart/form-data"`. Display current filename in edit view.  
  - Add resource route to `routes/admin.php`: `Route::resource('document-templates', DocumentTemplateController::class);`.

### **Step 1.7: Create Public Routes/Controller for Templates (incl. Download Logic)**

* **Goal:** Allow public users to browse and securely download templates, respecting access rules.
* **New Files:** app/Http/Controllers/PublicTemplateController.php, View resources/views/public/templates/index.blade.php.
* **Prompt 1.7.1 (Backend - Controller & Routes):**  
  Create `PublicTemplateController` (`php artisan make:controller PublicTemplateController`).  
  - Add `index` method: Fetch `DocumentTemplate` records where `status` is 'published'. Implement filtering by `state\_code` and `category` based on request parameters (`$request->input('state')`, `$request->input('category')`). Paginate results. Fetch distinct states and categories with published templates to populate filters. Pass data to the view.  
  - Add `download(DocumentTemplate $template)` method (using route model binding):  
   - Check if `$template->status !== 'published'`. If not, `abort(404)`.  
   - Check if `$template->requires\_signup` is true.  
   - If true, check if an owner is authenticated: `Auth::guard('owner')->check()`. (Requires Owner Auth setup from Phase 2).  
   - If signup required and user not logged in, redirect to owner login route (`route('owner.login')`) or `abort(403, 'Login required')`.  
   - If checks pass: Increment download count (`$template->increment('download\_count');`). Return the file download response: `return Storage::disk('templates')->download($template->file\_path, $template->original\_filename);`. Handle potential `FileNotFoundException`.  
  - Add routes in `routes/web.php`:  
   - `Route::get('/templates', [PublicTemplateController::class, 'index'])->name('public.templates.index');`  
   - `Route::get('/templates/download/{template}', [PublicTemplateController::class, 'download'])->name('public.templates.download')->middleware(['auth.optional:owner']);` // Use optional auth if needed, refine middleware later.
* **Prompt 1.7.2 (Backend - View):**  
  Generate `resources/views/public/templates/index.blade.php` view extending a public layout.  
  - Display filter dropdowns/forms for State and Category, populated from controller data. Forms should submit GET requests to filter.  
  - Iterate through the paginated `$templates`. Display `title`, `description`, `state\_code`, `category`.  
  - Add a "Download" button linking to `route('public.templates.download', $template)`. Add visual cues (e.g., lock icon) if `requires\_signup` is true.  
  - Include pagination links (`$templates->links()`).  
  - Apply Tailwind CSS to match the Next.js frontend styling.

## **PHASE 2: User Dashboards & Platform Maturity (Backend)**

### **Step 2.1: Implement Property Owner Authentication**

* **Goal:** Create a separate login/registration system for property owners.
* **Key Files:** config/auth.php, app/Models/Owner.php (New), app/Http/Controllers/OwnerAuth/ (New Folder), Routes (routes/owner.php - New), Views (resources/views/owner/auth/).
* **Prompt 2.1.1 (Backend - Model & Migration):**  
  Generate a model `Owner` with migration: `php artisan make:model Owner -m`.  
  - In the migration, define columns similar to the `users` table: `id`, `name`, `email` (unique), `email\_verified\_at` (nullable), `password`, `remember\_token`, `timestamps`.  
  - In the `Owner` model, implement the `Illuminate\Contracts\Auth\MustVerifyEmail` interface and use `Illuminate\Foundation\Auth\User as Authenticatable`, `Notifiable` traits. Set `$fillable` attributes.
* **Prompt 2.1.2 (Backend - Auth Configuration):**  
  Configure a new authentication guard and provider for owners in `config/auth.php`:  
  - Under `guards`, add: `'owner' => ['driver' => 'session', 'provider' => 'owners']`.  
  - Under `providers`, add: `'owners' => ['driver' => 'eloquent', 'model' => App\Models\Owner::class]`.  
  - Under `passwords`, add: `'owners' => ['provider' => 'owners', 'table' => 'password\_resets', 'expire' => 60, 'throttle' => 60]`. (Use existing password\_resets table or create a new one).
* **Prompt 2.1.3 (Backend - Auth Controllers & Routes):**  
  Generate authentication controllers specifically for owners (e.g., using `laravel/ui` scaffolding adjusted for the owner guard, or manually create controllers):  
  - Create controllers under `app/Http/Controllers/OwnerAuth/` for Login, Register, ForgotPassword, ResetPassword, Verification.  
  - Adapt the logic within these controllers to use the `owner` guard (e.g., `Auth::guard('owner')->attempt(...)`, `Auth::guard('owner')->login(...)`, `$request->user('owner')`).  
  - Create a new route file `routes/owner.php`. Define routes for login, register, logout, password reset, and email verification, pointing to the new OwnerAuth controllers. Group these routes and apply appropriate middleware (e.g., `web`).  
  - Include this route file in `app/Providers/RouteServiceProvider.php` under the `web` middleware group, perhaps with a prefix like `/owner`.
* **Prompt 2.1.4 (Backend - Auth Views):**  
  Create Blade views for owner authentication under `resources/views/owner/auth/` (login, register, verify, passwords/email, passwords/reset). Adapt standard Laravel auth views to use the correct routes (e.g., `route('owner.login')`) and styling to match the public frontend.
* **Integration Point:** Consider how/when owners register. Option 1: Separate registration page. Option 2: Automatically create an inactive owner account after lead submission and send verification email. Modify Api/HomePageController@storeLead for Option 2.

### **Step 2.2: Implement Property Owner Dashboard (Basic)**

* **Goal:** Provide a logged-in area for owners to view their submitted leads.
* **New Files:** app/Http/Controllers/OwnerDashboardController.php, Views (resources/views/owner/dashboard.blade.php), Routes (routes/owner.php).
* **Prompt 2.2.1 (Backend - Controller & Route):**  
  Create `OwnerDashboardController` (`php artisan make:controller OwnerDashboardController`).  
  - Add an `index` method. Inside, get the authenticated owner (`Auth::guard('owner')->user()`). Fetch associated leads (requires linking `Leads` to `Owners` - add `owner\_id` to `leads` table via migration and define relationships: `Owner` hasMany `Lead`, `Lead` belongsTo `Owner`). Pass the owner's leads to the view.  
  - Add a route in `routes/owner.php`, protected by owner auth middleware (e.g., `auth:owner`): `Route::get('/dashboard', [OwnerDashboardController::class, 'index'])->name('owner.dashboard');`. Ensure the `Authenticate` middleware redirects correctly for the `owner` guard.
* **Prompt 2.2.2 (Backend - View):**  
  Create `resources/views/owner/dashboard.blade.php`.  
  - Extend a simple owner-specific layout (create `layout/owner.blade.php`).  
  - Display a welcome message.  
  - List the leads submitted by the owner (e.g., Property Address, Date Submitted, Status - initially just 'Submitted').  
  - Add navigation links (e.g., Submit New Lead, Resources, Profile).  
  - Apply styling consistent with the public frontend.
* **Prompt 2.2.3 (Backend - Link Lead to Owner):**  
  Generate a migration to add a nullable `owner\_id` (foreign key to `owners` table) to the `leads` table. Run migration.  
  Update `Api/HomePageController@storeLead`: If an owner is logged in during submission (`Auth::guard('owner')->check()`), associate the lead with their ID. If not logged in, consider logic from Step 2.1 Integration Point (e.g., create owner account). Update the `Owner` and `Lead` models with `hasMany`/`belongsTo` relationships.

### **Step 2.3: Implement Owner Bookmarking Feature (Backend)**

* **Goal:** Allow logged-in owners to save links to legal content and templates.
* **New Files:** database/migrations/...create\_owner\_bookmarks\_table.php, app/Models/OwnerBookmark.php, app/Http/Controllers/OwnerBookmarkController.php, Routes (routes/owner.php).
* **Prompt 2.3.1 (Backend - Migration & Model):**  
  Generate migration and model `OwnerBookmark`. Define schema:  
  - `id`  
  - `owner\_id` (foreign key to `owners.id`, cascade on delete)  
  - `bookmarkable\_id` (unsignedBigInteger)  
  - `bookmarkable\_type` (string) // For polymorphic relation (StateLawContent, DocumentTemplate)  
  - `timestamps()`  
  - Add unique composite index: `$table->unique(['owner\_id', 'bookmarkable\_id', 'bookmarkable\_type']);`  
  In the model, define `$fillable` (`owner\_id`, `bookmarkable\_id`, `bookmarkable\_type`). Define the polymorphic relationship `public function bookmarkable() { return $this->morphTo(); }`. In the `Owner` model, define `public function bookmarks() { return $this->hasMany(OwnerBookmark::class); }`.  
  Run migration.
* **Prompt 2.3.2 (Backend - Controller & Routes):**  
  Create `OwnerBookmarkController`.  
  - Add `store(Request $request)` method: Validate input (`bookmarkable\_id`, `bookmarkable\_type`). Get authenticated owner ID. Create or update bookmark using `updateOrCreate`. Return JSON success response.  
  - Add `destroy(Request $request)` method: Validate input. Find and delete the bookmark for the authenticated owner. Return JSON success response.  
  - Add routes in `routes/owner.php` protected by `auth:owner` middleware:  
   - `Route::post('/bookmarks', [OwnerBookmarkController::class, 'store'])->name('owner.bookmarks.store');`  
   - `Route::delete('/bookmarks', [OwnerBookmarkController::class, 'destroy'])->name('owner.bookmarks.destroy');`
* **Prompt 2.3.3 (Backend - Display Bookmarks):**  
  Update `OwnerDashboardController@index` method: Eager load bookmarks with their related models: `$owner->load(['bookmarks.bookmarkable'])`. Pass bookmarks to the `owner/dashboard.blade.php` view.  
  Update the dashboard view to display saved bookmarks, linking to the original content (e.g., `route('public.laws.show', ...)`, `route('public.templates.download', ...)`).  
  \*Note: Frontend integration (AJAX calls from law/template pages to store/destroy bookmarks) is also required.\*

### **Step 2.4: Implement Owner Saved Calculations (Backend)**

* **Goal:** Allow logged-in owners to save results from the calculators.
* **New Files:** database/migrations/...create\_saved\_calculations\_table.php, app/Models/SavedCalculation.php, app/Http/Controllers/SavedCalculationController.php, Routes (routes/owner.php).
* **Prompt 2.4.1 (Backend - Migration & Model):**  
  Generate migration and model `SavedCalculation`. Define schema:  
  - `id`  
  - `owner\_id` (foreign key to `owners.id`, cascade on delete)  
  - `calculator\_type` (string, index) // e.g., 'rental-roi', 'rent-vs-buy'  
  - `title` (string, nullable) // User-defined name for the calculation  
  - `inputs\_json` (JSON)  
  - `results\_json` (JSON)  
  - `timestamps()`  
  In the model, define `$fillable` and cast `inputs\_json`, `results\_json` to `array`. Define relationship in `Owner` model: `public function savedCalculations() { return $this->hasMany(SavedCalculation::class); }`.  
  Run migration.
* **Prompt 2.4.2 (Backend - Controller & Routes):**  
  Create `SavedCalculationController`.  
  - Add `store(Request $request)` method: Validate input (`calculator\_type`, `title`, `inputs\_json`, `results\_json`). Get authenticated owner ID. Create `SavedCalculation`. Return JSON success response.  
  - Add `destroy(SavedCalculation $calculation)` method (route model binding): Check if calculation belongs to auth owner. If yes, delete it. Return JSON success/error response.  
  - Add routes in `routes/owner.php` protected by `auth:owner`:  
   - `Route::post('/saved-calculations', [SavedCalculationController::class, 'store'])->name('owner.calculations.store');`  
   - `Route::delete('/saved-calculations/{calculation}', [SavedCalculationController::class, 'destroy'])->name('owner.calculations.destroy');`
* **Prompt 2.4.3 (Backend - Display Saved Calculations):**  
  Update `OwnerDashboardController@index`: Eager load `savedCalculations`. Pass them to the `owner/dashboard.blade.php` view.  
  Update the dashboard view to display a list of saved calculations (title, type, date). Add delete buttons triggering AJAX calls to the destroy route. \*Note: Frontend integration (AJAX calls from calculator pages) is needed to save calculations.\*

### **Step 2.5: Enhance Property Manager (User) Dashboard (Backend Logic)**

* **Goal:** Provide necessary data for the improved PM dashboard. (Assuming PMs log in via existing users table/admin guard, or a separate PM guard needs setup similar to Owner).
* **New Files:** app/Http/Controllers/PmDashboardController.php, Routes (potentially in routes/admin.php with middleware, or routes/pm.php).
* **Assumption:** Property Managers use the existing User model and potentially log in via the 'admin' guard or a specific role. Adjust guard/middleware names as needed.
* **Prompt 2.5.1 (Backend - Controller & Route):**  
  Create `PmDashboardController` (`php artisan make:controller PmDashboardController`).  
  - Add an `index` method. Get authenticated PM user (`Auth::user()` or `Auth::guard('pm')->user()`).  
  - Fetch necessary data:  
   - Recent leads assigned to PM: `$user->userLeads()->with('lead')->latest()->take(10)->get();` (Requires `userLeads` relationship on `User` model: `hasMany(UserLead::class)`).  
   - Lead statistics (requires `status` column on `user\_leads` - see Step 2.6): Count total leads, count by status (New, Contacted, Won).  
   - Credit balance (if applicable): `$user->credits` (assuming a `credits` attribute or relationship).  
   - Subscription/Tier info: `$user->tier` (assuming a relationship).  
  - Pass this data to the PM dashboard view.  
  - Add a route (e.g., in `routes/admin.php` or a new `routes/pm.php`) protected by appropriate auth middleware (e.g., `auth:admin`, potentially checking for PM role): `Route::get('/dashboard', [PmDashboardController::class, 'index'])->name('pm.dashboard');`. \*Adjust route path/name if `/admin/dashboard` is already used.\*
* **Prompt 2.5.2 (Backend - View):**  
  Create or modify the PM dashboard view (e.g., `resources/views/pm/dashboard.blade.php` or adapt `admin/dashboard.blade.php` based on user role).  
  - Extend an appropriate layout (admin or dedicated PM layout).  
  - Display the stats fetched from the controller (Total Leads, Leads by Status, Credits, Tier).  
  - Display the list of recent leads in a table.  
  - Apply styling consistent with the admin panel or target design.

### **Step 2.6: Implement PM Lead Status Management (Backend)**

* **Goal:** Allow PMs to update the status of leads assigned to them.
* **Key Files:** database/migrations/...create\_user\_leads\_table.php, app/Models/UserLeads.php, app/Http/Controllers/UserLeadController.php (New), Routes.
* **Prompt 2.6.1 (Backend - Migration & Model):**  
  Generate a migration to modify the `user\_leads` table: `php artisan make:migration add\_status\_notes\_to\_user\_leads\_table --table=user\_leads`.  
  - Add a string column `status` with a default value (e.g., 'New'). Add an index. Define allowed statuses (e.g., 'New', 'Contacted', 'Not a Fit', 'Client Won'). Consider using an enum if DB supports.  
  - Add a text column `notes` (nullable).  
  - Update `$fillable` in `UserLeads` model.  
  - Run migration.
* **Prompt 2.6.2 (Backend - Controller & Route):**  
  Create `UserLeadController` (`php artisan make:controller UserLeadController`).  
  - Add an `updateStatus(Request $request, UserLead $userLead)` method (using route model binding).  
   - Authorize that the `$userLead` belongs to the authenticated PM (`if ($userLead->user\_id !== Auth::id()) { abort(403); }`).  
   - Validate the incoming status (`$request->validate(['status' => 'required|in:New,Contacted,Not a Fit,Client Won'])`).  
   - Update the status: `$userLead->update(['status' => $request->status]);`.  
   - Return JSON success response.  
  - Add a route (e.g., in `routes/admin.php` or `routes/pm.php`) protected by PM auth middleware: `Route::patch('/user-leads/{userLead}/status', [UserLeadController::class, 'updateStatus'])->name('pm.userLeads.updateStatus');`.
* **Frontend Integration:** The PM dashboard/lead list view will need AJAX calls (e.g., on dropdown change) to this endpoint to update the status. Update PmDashboardController to fetch leads with their statuses.

### **Step 2.7: Implement PM Lead Notes (Backend)**

* **Goal:** Allow PMs to add private notes to leads assigned to them.
* **Key Files:** app/Http/Controllers/UserLeadController.php, Routes.
* **Prompt 2.7.1 (Backend - Controller & Route):**  
  Add a method `updateNotes(Request $request, UserLead $userLead)` to `UserLeadController`.  
  - Authorize that the `$userLead` belongs to the authenticated PM.  
  - Validate the notes (`$request->validate(['notes' => 'nullable|string|max:5000'])`).  
  - Update the notes: `$userLead->update(['notes' => $request->notes]);`.  
  - Return JSON success response.  
  - Add a route (e.g., `routes/pm.php`) protected by PM auth: `Route::patch('/user-leads/{userLead}/notes', [UserLeadController::class, 'updateNotes'])->name('pm.userLeads.updateNotes');`.
* **Frontend Integration:** The PM lead view will need a textarea and an AJAX call (e.g., on blur or save button click) to this endpoint.

## **PHASE 3: Advanced Logic & Verification (Backend)**

### **Step 3.1: Enhance PM Preference Model & UI (Backend)**

* **Goal:** Allow PMs to set more detailed preferences for lead matching.
* **Key Files:** database/migrations/...create\_user\_preferences\_table.php, app/Models/UserPreferences.php, app/Http/Controllers/Admin/PreferencesController.php (or dedicated PM controller), Admin/PM Views.
* **Prompt 3.1.1 (Backend - Migration & Model):**  
  Generate a migration to modify the `user\_preferences` table: `php artisan make:migration add\_details\_to\_user\_preferences\_table --table=user\_preferences`.  
  - Add columns based on desired criteria:  
   - `property\_types` (JSON, nullable) - Store array like ['single-family', 'multi-family'].  
   - `min\_units` (integer, nullable).  
   - `max\_units` (integer, nullable).  
   - `required\_services` (JSON, nullable) - Store array like ['leasing', 'maintenance'].  
   - \*(Add other relevant criteria fields)\*.  
  - Update `$fillable` and `$casts` (for JSON columns) in the `UserPreferences` model.  
  - Run migration.
* **Prompt 3.1.2 (Backend - Admin/PM UI Update):**  
  Update the controller and view where PMs manage preferences (likely `Admin/PreferencesController` and its view, or create dedicated PM preference routes/controller/view).  
  - Add form fields for the new criteria (e.g., checkboxes for property types/services, number inputs for units).  
  - Ensure the controller saves these preferences correctly (casting arrays to JSON before saving). Handle updating existing preferences (`updateOrCreate` based on `user\_id`).

### **Step 3.2: Refine Lead Matching Algorithm (LeadDistribute.php)**

* **Goal:** Improve lead relevance by using enhanced preferences.
* **Key Files:** app/Console/Commands/LeadDistribute.php.
* **Prompt 3.2.1 (Backend - Algorithm Update):**  
  Modify the `handle()` method in `app/Console/Commands/LeadDistribute.php`:  
  - Fetch unprocessed leads (`Leads` where status indicates distribution needed).  
  - For each lead:  
   - Identify the target zip code from the lead data.  
   - Query potential `Users` (PMs) who have this zip code in their `UserZipcodes`.  
   - \*\*Refine Query:\*\* Further filter these potential PMs based on the new `UserPreferences` matching the lead's details (e.g., `property\_type` from lead questions/data vs `user\_preferences.property\_types`, `lead.num\_units` vs `user\_preferences.min\_units`/`max\_units`, required services). Use `whereJsonContains` for JSON fields or appropriate clauses. Eager load preferences: `User::with('preferences')...`.  
   - Implement logic to handle multiple matches (e.g., round-robin, tier priority - see later steps).  
   - Create `UserLeads` records for matched PMs.  
   - Send notifications (Email/SMS).  
   - Update the `Lead` status to 'Distributed'.  
  - Wrap the processing of each lead in a transaction. Add robust logging and error handling.

### **Step 3.3: Implement Lead Cap/Budget Logic**

* **Goal:** Limit lead distribution based on PM subscription or credits.
* **Key Files:** app/Console/Commands/LeadDistribute.php, app/Models/User.php, app/Models/Tier.php (if exists), app/Models/Credits.php.
* **Prompt 3.3.1 (Backend - Algorithm Update):**  
  Modify the `handle()` method in `app/Console/Commands/LeadDistribute.php`, within the loop where matched PMs are found for a lead:  
  - Before creating a `UserLeads` record and sending notification for a specific PM:  
   - Check the PM's subscription/tier status (`$pm->tier->lead\_limit`? Requires relationships). Compare against leads received this month/cycle.  
   - OR Check the PM's credit balance (`$pm->credits`). Ensure balance is sufficient (e.g., > 0 or > lead\_cost).  
   - If the PM is over limit or has insufficient credits, \*skip\* assigning the lead to this PM and continue to the next potential match.  
   - If the lead is assigned and uses credits, decrement the PM's credit balance (`$pm->decrement('credits', $lead\_cost);`). Ensure this is atomic (use DB transaction).  
  - Add relevant relationships/attributes (`tier`, `credits`) to the `User` model if not present.

### **Step 3.4: Implement Tiered Lead Exclusivity Logic**

* **Goal:** Offer exclusive leads to higher-paying tiers for a period.
* **Key Files:** app/Console/Commands/LeadDistribute.php, database/migrations/...create\_user\_leads\_table.php, app/Models/UserLeads.php, app/Models/Tier.php (if exists).
* **Prompt 3.4.1 (Backend - Migration & Model):**  
  Generate a migration to modify the `user\_leads` table:  
  - Add a nullable timestamp column `exclusive\_until`.  
  Run migration. Update `$fillable` and `$dates` cast in `UserLeads` model.
* **Prompt 3.4.2 (Backend - Algorithm Update):**  
  Modify `LeadDistribute.php`:  
  - When querying potential PMs for a lead, prioritize PMs based on their tier (requires `$pm->tier->priority` or similar attribute).  
  - If the highest priority matching PM(s) belong to a tier that grants exclusivity (check `$pm->tier->has\_exclusivity` flag):  
   - Assign the lead ONLY to these top-tier PM(s).  
   - Set the `exclusive\_until` timestamp on the created `UserLeads` record(s) (e.g., `now()->addHours(24)`).  
   - Update the `Lead` status to something like 'Distributed - Exclusive'.  
  - Add a \*separate\* scheduled command or logic within `LeadDistribute` to run later:  
   - Find leads with status 'Distributed - Exclusive' where `exclusive\_until` has passed.  
   - Find lower-priority PMs matching these leads (excluding those who already received it).  
   - Assign the lead to them (without setting `exclusive\_until`).  
   - Update the `Lead` status to 'Distributed'.  
  - Ensure the initial query in `LeadDistribute` only processes leads that are 'New' or 'Distributed - Exclusive' (and past expiry).

### **Step 3.5: Implement "Verified" PM Program (Backend)**

* **Goal:** Allow admins to mark PMs as verified, potentially influencing lead distribution.
* **Key Files:** database/migrations/...create\_users\_table.php, app/Models/User.php, app/Http/Controllers/Admin/UsersController.php, Admin Views.
* **Prompt 3.5.1 (Backend - Migration & Model):**  
  Generate a migration to modify the `users` table: `php artisan make:migration add\_verification\_to\_users\_table --table=users`.  
  - Add a boolean column `is\_verified` (default false, index).  
  - Add a text column `verification\_notes` (nullable) for admin use.  
  - (Optional) Add a timestamp `verified\_at` (nullable).  
  - Update `$fillable` and `$casts` in the `User` model.  
  - Run migration.
* **Prompt 3.5.2 (Backend - Admin UI):**  
  Update `Admin/UsersController` and views (`edit.blade.php`, `index.blade.php`):  
  - In the edit view, add a checkbox or toggle for "Is Verified". Add a textarea for "Verification Notes".  
  - Update the `update` method to save these fields. Add authorization checks if needed (only certain admin roles can verify?).  
  - In the index view, display the verification status (e.g., a badge). Allow filtering/sorting by verification status.
* **Prompt 3.5.3 (Backend - Algorithm Update):**  
  Modify `LeadDistribute.php`:  
  - When querying potential PMs for a lead, add sorting to prioritize verified users (`orderBy('is\_verified', 'desc')`) before other criteria like tier or round-robin.
* **Frontend:** Ensure the "Verified" status is included in API responses for PM listings/details so a badge can be shown.

## **PHASE 4: Technical Debt, API Expansion & Deployment Prep (Backend)**

### **Step 4.1: Implement Unit/Integration Tests**

* **Goal:** Increase code reliability and prevent regressions.
* **Key Files:** Test files under tests/Feature/ and tests/Unit/.
* **Prompt 4.1.1 (Backend - Feature Tests):**  
  Generate PHPUnit Feature tests for critical user flows:  
  - \*\*Lead Submission:\*\* Test the `/api/home-page-lead` endpoint with valid and invalid data (using `StoreLeadRequest`). Assert correct responses (201, 422) and database state (Lead, Responses created).  
  - \*\*Owner Auth:\*\* Test registration, login, password reset endpoints for the `owner` guard.  
  - \*\*PM Auth:\*\* Test login, password reset for the `admin`/`pm` guard.  
  - \*\*Template Download:\*\* Test the `public.templates.download` route, checking access control (free vs. requires\_signup, logged in vs. out). Mock the Storage facade.  
  - \*\*Lead Distribution Command:\*\* Create a test for `LeadDistribute`. Seed necessary data (Lead, Users, Preferences, Zipcodes). Execute the command (`$this->artisan('leads:distribute')`). Assert that `UserLeads` records are created correctly based on matching rules. Assert notifications were sent (mock Mail facade).
* **Prompt 4.1.2 (Backend - Unit Tests):**  
  Generate PHPUnit Unit tests for complex logic within models or services:  
  - Test any custom logic within the `User`, `Lead`, or `UserPreferences` models.  
  - If specific helper classes or services were created (e.g., for matching), test their methods in isolation.

### **Step 4.2: API Security Review & Enhancement**

* **Goal:** Ensure backend APIs are properly secured.
* **Key Files:** routes/api.php, routes/owner.php, routes/pm.php, app/Http/Kernel.php, Controllers.
* **Prompt 4.2.1 (Backend - Review & Apply Middleware):**  
  Review all routes defined in `routes/api.php`, `routes/owner.php`, `routes/pm.php` (or PM routes in `admin.php`).  
  - Ensure appropriate authentication middleware is applied:  
   - Public API endpoints (like lead submission, public data fetching) should NOT require auth, but should have rate limiting (`throttle:api`).  
   - Owner-specific endpoints (dashboard, bookmarks, saved calculations) MUST use `auth:owner` middleware.  
   - PM-specific endpoints (dashboard, lead status updates) MUST use `auth:admin` (or `auth:pm` if a separate guard exists) and potentially role/permission checks.  
   - Admin-only endpoints MUST use `auth:admin` and role/permission checks.  
  - Review API controllers (`Api/\*`, `Owner\*`, `Pm\*`) for authorization logic (e.g., ensuring a user can only modify their own resources using policies or manual checks).  
  - Configure Laravel Sanctum if API tokens (e.g., for mobile app or SPA separate from web sessions) are needed. Ensure Sanctum middleware (`EnsureFrontendRequestsAreStateful`) is correctly configured in `app/Http/Kernel.php` if using SPA session-based auth.  
  - Review CORS configuration (`config/cors.php`) to ensure only allowed origins (Next.js frontend domain) can access the API.

### **Step 4.3: Query Optimization**

* **Goal:** Improve database performance, especially for list views and matching.
* **Key Files:** Controllers (index methods), LeadDistribute.php.
* **Prompt 4.3.1 (Backend - Eager Loading & Indexing):**  
  Review Eloquent queries in controllers and commands that fetch multiple records or involve relationships:  
  - Identify potential N+1 query problems using tools like Laravel Debugbar or Telescope.  
  - Apply eager loading (`->with([...])`) where appropriate (e.g., loading `lead` with `userLeads` on PM dashboard, loading `preferences` and `zipcodes` when matching users in `LeadDistribute`).  
  - Analyze `WHERE` clauses and `ORDER BY` clauses in critical queries (e.g., finding PMs by zip code, filtering leads/templates/laws).  
  - Generate new migrations to add database indexes to frequently queried columns (`state\_code`, `status`, `zipcode`, foreign keys like `user\_id`, `lead\_id`, `owner\_id`).

### **Step 4.4: Create/Update Public API Endpoints for Frontend Data Needs**

* **Goal:** Provide efficient API endpoints for the Next.js frontend.
* **Key Files:** routes/api.php, app/Http/Controllers/Api/ folder.
* **Prompt 4.4.1 (Backend - API Resources/Controllers):**  
  Review the data needed by the Next.js frontend (Testimonials, Blog posts/slug, PM list/filters, PM detail/slug, States, Cities by State).  
  - Create dedicated API controllers under `app/Http/Controllers/Api/` if they don't exist (e.g., `Api/TestimonialController`, `Api/BlogController`, `Api/PropertyManagerController`, `Api/LocationController`).  
  - Implement controller methods to fetch and return the required data. Use pagination (`paginate()`) for lists. Apply filtering/sorting based on request parameters.  
  - Use Laravel API Resources (`php artisan make:resource ...`) to transform Eloquent models into consistent JSON structures, selecting only the necessary fields to minimize payload size. Eager load relationships needed for the resource.  
  - Define public routes in `routes/api.php` for these controllers, applying rate limiting (`throttle:api`).  
  - Example endpoints:  
   - `GET /api/testimonials/published`  
   - `GET /api/blogs?page=1&limit=10&category=...`  
   - `GET /api/blogs/{slug}`  
   - `GET /api/property-managers?state=CA&city=Sacramento&page=1&sort=rating&filters[services]=leasing`  
   - `GET /api/property-managers/{slug}`  
   - `GET /api/locations/states`  
   - `GET /api/locations/states/{state\_code}/cities`

### **Step 4.5: Backend Deployment Preparation**

* **Goal:** Configure the Laravel application for a production environment.
* **Key Files:** .env.example, config/\*.php, app/Providers/AppServiceProvider.php.
* **Prompt 4.5.1 (Backend - Environment & Config):**  
  Prepare the Laravel application for production deployment:  
  - \*\*Environment File:\*\* Ensure `.env.example` includes all necessary environment variables required for production (`APP\_NAME`, `APP\_ENV=production`, `APP\_KEY`, `APP\_DEBUG=false`, `APP\_URL`, Database credentials, Mail driver/credentials, Queue connection, Cache driver, Session driver, Stripe keys, Filesystem disk config, etc.).  
  - \*\*Configuration Caching:\*\* Ensure deployment script runs `php artisan config:cache` and `php artisan route:cache`.  
  - \*\*Optimization:\*\* Ensure deployment script runs `php artisan optimize` (combines config, route, view caching) and `composer install --optimize-autoloader --no-dev`.  
  - \*\*Error Reporting:\*\* Configure logging (`config/logging.php`) and error reporting services (e.g., Sentry, Flare) for production.  
  - \*\*HTTPS:\*\* In `AppServiceProvider`'s `boot` method, force HTTPS scheme if behind a load balancer/proxy: `URL::forceScheme('https');`.  
  - \*\*Queue Worker:\*\* Set up Supervisor (or similar) on the server to run `php artisan queue:work` persistently for handling jobs like lead distribution and email sending. Configure queue connection (`QUEUE\_CONNECTION`) in `.env`.  
  - \*\*Scheduler:\*\* Ensure the server's cron is set up to run `php artisan schedule:run` every minute to trigger scheduled commands like `LeadDistribute`.  
  - \*\*Storage Link:\*\* Ensure deployment script runs `php artisan storage:link`.  
  - \*\*Database Migrations:\*\* Ensure deployment script runs `php artisan migrate --force`.  
  - \*\*Permissions:\*\* Ensure `storage` and `bootstrap/cache` directories are writable by the web server user.
* **Prompt 4.5.2 (Backend - Deployment Documentation):**  
  Create a `DEPLOYMENT.md` file in the Laravel project root documenting:  
  - Server requirements (PHP version, extensions, DB, etc.).  
  - Required environment variables (list from `.env.example`).  
  - Step-by-step deployment process (e.g., git pull, composer install, npm install/build, artisan commands: migrate, cache, optimize, storage:link, queue worker setup, cron setup, web server config).  
  - Notes on configuring Nginx/Apache virtual host (pointing document root to `/public`, PHP-FPM setup).  
  - Rollback procedure.

**AI Agent Usage Notes:**

* **Context is Key:** Provide relevant existing file contents when asking for modifications.
* **Iterative Refinement:** Review generated code. Use follow-up prompts for clarification, error handling, or style adjustments.
* **Testing:** Manually test each feature after implementation. Use the generated PHPUnit tests (Step 4.1).
* **Security:** Pay close attention to authorization, validation, file handling, and API security steps. Review carefully.
* **Transactions:** Use database transactions (DB::transaction()) for operations involving multiple database writes (like creating Lead + Responses, or assigning Lead + decrementing Credits).