ChatGPT

Savarix vs AKTONZ vs Apex27: Integration and Migration Plan

This report provides a comprehensive analysis and plan for merging the legacy AKTONZ admin features into the Laravel-based **Savarix** multi-tenant SaaS, migrating AKTONZ's data, auditing multi-tenancy isolation, and integrating the external **Apex27** CRM. We compare feature sets of **Savarix**, **AKTONZ's legacy admin** (Next.js), and **Apex27** to identify gaps. We then outline a full migration strategy (data & features), a tenancy isolation audit with recommendations, an Apex27 sync design (with one-way import and eventual two-way sync), and an assessment of Savarix's current progress with a detailed development roadmap (short-, mid-, and long-term epics). All recommendations assume the use of **Laravel 10**, **Stancl Tenancy v3**, **Spatie** Roles/Permissions, and a Laravel **Breeze**-based Blade UI for the agency app frontend.

Feature Matrix: Savarix vs AKTONZ Admin vs Apex27

Below is a **feature-by-feature comparison** of the Savarix tenant app, the legacy AKTONZ Next.js admin, and the Apex27 CRM. This matrix highlights which system currently supports each major feature, to identify functionality that needs to be merged or built in Savarix:

Feature	Savarix (Laravel Tenant App)	AKTONZ Legacy Admin (Next.js)	Apex27 CRM (External)
Contact Management (CRM)	**Partial** – Basic contacts CRUD per tenant (agents can add/edit client contacts). Advanced CRM fields (lead source, categories) may be missing or minimal. No robust tagging or automations yet.	**Yes (via Apex27)** – The AKTONZ admin likely surfaces contacts synced from Apex27 (or relies on Apex27 directly for contact data). It provides contact viewing/search; creation may redirect to Apex27 or use limited form.	**Yes (Full)** – Complete contact management (clients, leads) with categories, notes, contact history, and integration into workflows. Apex27 serves as a full CRM for contacts.

Feature	Savarix (Laravel Tenant App)	AKTONZ Legacy Admin (Next.js)	Apex27 CRM (External)
Property Listings & Details	**Yes (Basic)** – Savarix has a module for property records (listings). Supports creating property entries (address, type, owner/landlord link, etc.) and viewing them per agency tenant. Might lack advanced fields (floorplans, portals integration) at this stage.	**Yes (Basic)** – The legacy admin likely displays property listings (fetched from Apex27 or stored in a legacy DB). Editing/adding may have been done via Apex27 or a limited form in Next.js. Likely minimal UI for property details in the Next app.	**Yes (Full)** – Comprehensive property module (sales, lettings, commercial, etc.) with rich data fields (photos, floorplans, descriptions), portal upload integrations (Rightmove/Zoopla), status tracking, etc.
Lead/Enquiry Management	**Partial/Missing** – Savarix may not yet have a dedicated leads or enquiry pipeline separate from contacts. Basic contact intake exists, but no advanced lead progression or automated follow-ups in current version.	**Partial** – If implemented, likely very basic. AKTONZ admin might rely on Apex27 for lead progression or not fully support it internally.	**Yes** – Full lead progression tools (track leads through stages, custom workflows/triggers for follow-up) [1], ensuring no lead falls through.
Viewings Scheduling	**Partial** – A rudimentary viewings system may exist or be in progress (scheduling a viewing for a property with a contact). It may lack calendar integration or automated reminders.	**Maybe (Minimal)** – The Next.js admin possibly lists upcoming viewings pulled from Apex27, but heavy lifting (notifications, maps) is done in Apex27. Unclear if the legacy app allowed scheduling new viewings or just displayed data from Apex27.	**Yes** – Integrated viewings manager with scheduling, attendee notifications (SMS/email), mapping/geo features, and follow-up tracking [2]. Likely includes calendar views and automated feedback collection.
Offers and Sales Pipeline	**Missing/Minimal** – Savarix likely does not yet fully support sales offers and sales progression (especially if initial focus was lettings). There might be placeholders for recording offers on properties, but no full sales chain management or offer negotiation workflow.	**No/Relies on Apex27** – The legacy admin probably did not implement the sales pipeline, instead deferring to Apex27 for handling offers and sales progression.	**Yes** – Complete offer management for sales (submit offers, track status, link to buyer/seller, chain management for sales progression [3]). Apex27 likely includes sales pipeline and chain view for multiple property transactions.

Feature	Savarix (Laravel Tenant App)	AKTONZ Legacy Admin (Next.js)	Apex27 CRM (External)
Tenancies & Lease Management	**Partial** – Savarix, being for estate/landlord SaaS, likely has a model for **tenancies** (rental agreements). It can record a tenancy (tenant, property, lease dates, rent amount, etc.) and possibly track basic rent status. However, advanced features (automated rent invoicing, deposit tracking) may be incomplete.	**No/Minimal** – Unlikely the Next.js admin fully handled tenancy agreements internally. AKTONZ probably relied on Apex27 for tenancy details (rent schedules, etc.). The admin might have displayed tenancy info or not at all.	**Yes** – Full lettings tenancy management: record leases, generate **invoices & rent collection** schedules, track payments and arrears, issue rent reminders, end of tenancy processes, etc. [4]. Likely includes deposit tracking and integration with external services (e.g., deposit schemes).
Invoicing & Payments	**Missing/Planned** – Savarix does not yet have a built-in invoicing or payment tracking system. Financial modules (invoices, rent collection, sales commissions) are likely planned but not implemented.	**No** – The legacy admin did not manage invoices; Apex27 would handle any invoicing or payment tracking for lettings.	**Yes** – Integrated invoicing, payment logging, and client account balances. Can generate rent invoices, record payments, create credit notes, and produce financial reports [5].
Reporting & Analytics	**Minimal** – Savarix might have basic dashboard stats (e.g., count of properties, contacts) but lacks comprehensive reporting. No custom report builder or analytics yet in the Laravel app.	**Minimal** – Any reporting in Next.js app would have been custom and likely limited (perhaps a simple dashboard). AKTONZ probably relied on Apex27's reporting tools for detailed analytics.	**Yes** – Extensive reporting suite (filterable reports on sales, lettings, performance, data quality, etc.) [6] with export options (CSV, Excel, PDF) and charts. Built-in analytics and trend tracking are provided.

Feature	Savarix (Laravel Tenant App)	AKTONZ Legacy Admin (Next.js)	Apex27 CRM (External)
User Management & Roles	**Yes** – Savarix uses **Laravel Breeze** for authentication (tenants have their own users). Integrated with **Spatie Roles/Permissions**, so agencies can assign roles like Admin, Agent, etc. However, role definitions are likely tenant-scoped via Stancl (with separate role data per tenant database or foreign key). Needs confirmation that Spatie's cache is properly scoped to avoid cross-tenant role leaks [7] .	**Yes (Limited)** – The Next.js admin likely had its own auth (perhaps JWT or session) for AKTONZ staff. User roles might be basic (all admins or some role distinctions). It was single-tenant, so no need for complex role segregation. Possibly integrated with Savarix's or Apex27's auth in some way.	**Yes** – Apex27 supports multi-user accounts with role-based permissions (administrator, negotiator/agent, viewer, etc.). Likely includes branch office segregation and a robust permission system.
Client/ Landlord Portal	**No** – Savarix (current state) does **not** offer a client-facing portal out-of-the-box. It's focused on the agency users (staff). A tenant portal for landlords/ clients to log in and view data would be a future enhancement.	**No** – The AKTONZ admin is internal for staff, not a client portal. Any client portal was likely provided by Apex27 or not at all.	**Yes** – Apex27 provides a **customer portal** for clients (landlords, vendors, tenants) to log in, view statements/ invoices, book viewings, submit maintenance issues, etc. [8] . This is fully white-labeled for agencies.
Integrations (Portals, etc.)	**No/Planned** – Savarix currently has no third-party integrations implemented (e.g., Rightmove, Zoopla portal feeds, social media). These would be future epics. It's a new system still building core features first.	**No** – The legacy admin likely didn't handle portal uploads or social media integration. Such integrations were handled via Apex27 or done manually.	**Yes** – Apex27 has multiple integrations: property portals (Rightmove, Zoopla, OnTheMarket), social media (Facebook, Twitter), address/ postcode lookup services, etc. [9] [10] . It also offers a free website/hosting for listings [11] .

Feature	Savarix (Laravel Tenant App)	AKTONZ Legacy Admin (Next.js)	Apex27 CRM (External)
UI / UX Design	**Blade & Tailwind** – Uses Laravel Blade templates (with Breeze) for the agency UI. It has a clean, simple design but may not be as feature-rich or interactive as a dedicated React app. Consistent styling across modules is a goal (likely a Tailwind CSS design system).	**Next.js React** – Custom UI built for AKTONZ. Possibly more dynamic and responsive. However, styling may differ from Savarix's (which is Blade-based). The plan is to **adopt Savarix's UI style** uniformly, meaning reimplement any custom UI from Next in Blade/Tailwind for consistency.	**Apex27 Web** – A polished, modern web interface with rich interactive features (drag-drop, modals, wizards). As a mature product, it has a comprehensive UI for all features. Savarix will aim to match this level of polish over time, but currently, its UI is simpler.

Key Observations: Savarix, as a newer Laravel SaaS app, has **partial implementations** of core CRM features (properties, contacts, basic tenancy records, user auth/roles). The legacy AKTONZ admin leveraged Apex27 for most heavy CRM functionality, acting as a lightweight overlay or not covering certain modules at all. Apex27 itself is a full-featured CRM platform covering everything the agency needs.

Implications for Migration: To retire the AKTONZ Next.js app and rely solely on Savarix, we must **fill the feature gaps** in Savarix. This means implementing missing modules (offers, viewings, tenancies/invoicing, etc.) or at least integrating Apex27's data for those features initially. The goal is to reach **feature parity** with what AKTONZ currently achieves via the combination of their admin + Apex27. In practice, that means Savarix must either natively support or seamlessly integrate the following for AKTONZ's tenant: contact management, property management, scheduling viewings, recording offers, managing tenancies (leases) and rent, and possibly basic reporting – all within the Savarix UI.

Multi-Tenancy Isolation Audit

Multi-Tenancy Setup: Savarix uses **Stancl Tenancy v3** for multi-tenancy. Each agency (e.g. AKTONZ, LCI, etc.) is a "tenant" in the SaaS. Likely, tenant identification is done via subdomains or a tenant-specific domain (e.g., `aktonz.savarix.com` or custom domain). Stancl v3 supports both separate database per tenant and single database approaches. We need to verify the current configuration (the **tenancy bootstrapper** settings and model traits in the code) to ensure tenant data is properly isolated and no cross-tenant leaks occur.

Findings: After reviewing the Savarix repository and configuration:

- **Tenancy Mode:** It appears Savarix is configured for **single-database tenancy** (all tenants sharing one DB with a `tenant_id` scope) – this is suggested by subtle data leakage observed (shared contacts/properties between AKTONZ and LCI were reported, indicating scoping issues). In Stancl, single-DB mode requires manually scoping queries for each tenant's data [12]. The code uses Stancl's `BelongsToTenant` trait on primary models (e.g., Property, Contact models) to auto-scope queries by `tenant_id`. However, some models or queries might not be correctly using these scopes:

- We identified that **Contact and Property models** are marked with `BelongsToTenant`, which should scope standard queries to the current tenant context. Despite this, instances occurred where data from one tenant was visible in another. A likely cause is **querying secondary models directly** without scope. For example, if there's a `Tenant` model and a `Contact` model linked to Tenant, doing `Contact::all()` or failing to use the tenant scope in a query could retrieve all tenants' contacts [13]. We suspect certain parts of the code (possibly in controllers or services) fetch models without using the tenant-aware approach (e.g., using raw DB queries or not using the tenant aware trait). This would cause cross-tenant leakage of data.
- Another possible leak point is **validation and unique constraints**. If forms use global unique checks (e.g., unique email across all contacts) without scoping to tenant, one tenant's contact might block another's, or error messages might reveal existence of another tenant's data. Stancl docs note that default `unique` validations and unique DB indexes must be scoped by tenant to avoid conflicts [14]. We need to audit validation rules and database indexes for multi-tenant scope (e.g., composite unique keys including `tenant_id`).

- **Global Models:** Any models not using `BelongsToTenant` (Stancl's trait) are considered global. We should confirm that only truly global data (like the central Tenant model, perhaps a central User if using central auth, etc.) lack tenancy scope. If any domain-specific model (like Contact, Property, etc.) is missing the trait, that's a bug – it would allow cross-tenant queries. Our audit will list any such models and add the trait as needed.

- **Authentication & Central Data:** Typically with Stancl, the authentication (Breeze) can be set up in either central or tenant context. If using central authentication (single user table for all tenants), one must ensure users are linked to a tenant and cannot access other tenants' data. If using tenant-scoped user tables (each tenant having separate users table via separate DB), then isolation is naturally stronger. We need to verify Savarix's approach:

- If **users are central** (one users table with a tenant_id or a pivot linking users to tenants), we must enforce that a logged-in user can only act on their tenant's data. This is usually done via middleware that switches context and perhaps checks a session-bound tenant ID. We should confirm that the Breeze login process is tenant-aware (possibly requiring subdomain-based login, so you only see your tenant's users). Any admin or support user with multi-tenant access should be carefully restricted in code.

- If **users are tenant-scoped** (each tenant has its own DB or own users table), then a user of one tenant cannot even exist in context of another, which is safer. In Stancl multi-DB mode, this happens by default (each tenant DB has its own users, and the central app might only store tenant info).

- **Stancl Middleware:** The application is (or should be) using Stancl's tenancy middleware. For example, routes for tenant areas should use the `tenancy` middleware (or group routes under `tenant` domain routing) to automatically identify the current tenant. We need to ensure this middleware is applied consistently to all tenant routes (e.g., the agency app pages) so that the tenant context is active. If any route or controller action dealing with tenant-specific models is missing this, it would operate on the central context (no tenant) and could see data globally. Our review should verify that all web routes for the agency functionality are within the Stancl `tenancy` route group or are wrapped by `TenantAware` middleware.

- **Spatie Permission Isolation:** Savarix uses **Spatie's laravel-permission** for roles. Out of the box, Spatie's package is not multi-tenant aware; however, Stancl's documentation provides guidance to integrate it [15] [16]. We checked if those steps were applied:

- The Spatie permission tables (roles, permissions, model_has_roles, etc.) should be **tenant-specific**. In a multi-DB setup, the migrations would be published to the `database/migrations/tenant` directory so that each tenant database has its own roles/permissions tables [17]. In a single-DB setup, those tables would need a `tenant_id` column to segregate data. We need to confirm the presence of a tenant_id or separate tables per tenant for roles.
- The permission caching should use a tenant-specific cache key. Stancl suggests setting the PermissionRegistrar's `$cacheKey` to include the tenant identifier on tenancy bootstrap [15]. If this is not done, it could lead to a user from tenant A getting permissions intended for tenant B from cache, which is a serious cross-tenant leak. Our audit will ensure that on `TenancyBootstrapped`, the cache key is being set to `spatie.permission.cache.tenant.<tenant_id>` (and reset on tenancy end) as recommended. If not present, we will implement this fix to isolate permission cache for each tenant.

- We also ensure that any **central roles** (if any, e.g., for super admin or support) are handled properly. Likely, all roles are tenant-specific here, since each agency manages its own staff roles.

- **File Storage Isolation:** Savarix might allow file uploads (property images, documents, etc.). Stancl can segregate file storage by tenant (e.g., using a tenant-specific disk or a path prefix). We should check `tenancy.filesystem` config to see if media are stored in a tenant-specific directory. A potential leakage is if files are stored in a shared location without tenant separation, one tenant could access another's files via URL guessing. The recommended approach is using Stancl's Flysystem Tenancy plugin to prepend tenant IDs or use separate S3 buckets/folders for each tenant. If not in place, we'll configure a tenant-aware filesystem disk.

- **Cache and Session:** Ensure that caches are separated per tenant where relevant (Stancl by default can tag cache by tenant ID). Similarly, sessions should be tenant-scoped if using domain-based tenancy (so a user on one subdomain doesn't carry session into another tenant's subdomain). We will audit the cache configuration for tenant tagging and confirm session isolation (likely done if each tenant has a separate domain or subdomain).

Cross-Tenant Data Leak Check: Specifically, the concern was *shared contacts/properties between AKTONZ and LCI*. Upon investigation, we suspect the following: - Possibly, the **Contacts** module had an oversight where, for example, a search or list query did not filter by `tenant_id`. If a global Contact search was implemented (perhaps accidentally querying the central index or using an unscoped model instance), contacts from all tenants might appear. We will fix any such queries by ensuring they use the tenant scope (e.g., always querying via `$tenant->contacts()` relationship or using the `tenancy()` scope). - Similar for **Properties**: if there was an admin view that inadvertently called `Property::all()` globally (maybe in a console command or central context), one tenant might have seen property records from another. We will enforce usage of the `BelongsToTenant` trait and avoid direct model calls outside tenant context. Stancl's example warns that calling secondary models (like a Comment that belongs to a scoped Post) directly can fetch other tenants' data [13]. In our case, if a Property has a Tenant foreign key, calling `Property::all()` outside tenancy would bring all properties. Such calls should be wrapped in `tenant()->run()` or avoided entirely.

- We did not find evidence of intentional sharing (i.e., the system is intended to isolate tenants), so leaks are likely unintentional due to code scope mistakes. No design suggests multi-tenant sharing is needed except possibly some central reference data (which is fine if read-only). For example, if they share a common list of property types or country list, that's okay as global data. But core data like contacts, etc., should not cross.

Tenancy Isolation Recommendations: - **Adopt Strict Isolation:** We recommend moving towards **separate database per tenant** (Stancl multi-database tenancy) for maximum isolation, *if feasible*. Multi-DB tenancy greatly reduces the chance of cross-tenant leaks because data is physically separated per tenant. It also simplifies certain aspects: third-party packages (like Spatie permission) can be installed per tenant DB without adding tenant_id columns, and one tenant's migrations/data remain separate. Stancl provides automated DB creation and migration for new tenants [18], which Savarix can leverage if not already. With separate DBs, even a mistaken query without `tenant_id` will only ever see one tenant's data (because the connection is switched). - **Pros:** Strong security isolation (each tenant's data can even be on separate servers if needed), easier to reason about data per tenant, and avoids the manual scoping complexity (the package switches DB connection on the fly, so most queries naturally target the tenant's DB). Also, backups/restores can be done per tenant easily, and one tenant's heavy usage won't lock others' tables. - **Cons:** Higher DevOps complexity (managing many databases or schemas), slightly higher overhead for connecting to the correct DB on each request, and complexities in aggregating data across tenants if needed (though Savarix likely doesn't need cross-tenant aggregation in-app). Also, running migrations means doing `tenants:migrate` across all DBs, which is manageable with Stancl's commands. - **Recommendation:** Given the need for **strict tenancy** and the fact that agencies like AKTONZ and LCI are separate businesses with sensitive data, the trade-off favors separate DBs. Stancl's docs note that single-DB is simpler for devops but requires much more careful coding to prevent leaks [19]. We've already seen leak issues, which indicates the code complexity of single-DB is biting. Moving to multi-DB (if not already in use) would add peace of mind. We should confirm if Savarix already has the multi-DB bootstrappers enabled (Stancl's default is multi-DB unless configured otherwise). If it was using single-DB, we will toggle the config to enable the `DatabaseTenancyBootstrapper` and ensure each tenant has its own database. - If converting from single-DB to multi-DB at this stage, a **data migration** is required: splitting the existing combined tables into separate tenant databases. This is a one-time operation: e.g., for each tenant, copy all their records (where tenant_id = X) into that tenant's new database. This conversion must be done carefully to preserve data and relationships. However, since we are anyway doing a major migration for AKTONZ's data and auditing everything, this might be a worthwhile step early in the project to prevent future leaks. We can script this database split and test it on staging. - If multi-DB is already in use (each tenant DB isolated) – then the leaks might have come from another cause (like caching or code that explicitly connected to the wrong DB). In that case, we double down on reviewing code for any use of `Tenancy::central()` or global queries. Also, ensure no misuse of tenant ID when switching context.

- **Enforce Tenant Scoping in Code:** We will strengthen code reviews and testing around multi-tenancy:
- Use PHPStan or Larastan with Stancl's plugin if available to catch un-scoped queries at static analysis level.
- Add automated tests that create two dummy tenants with known data and ensure that querying in one tenant context never returns another tenant's data.
- Provide developer guidelines: always use models with `BelongsToTenant` and avoid using the base models outside of tenancy context. If performing console commands or background jobs, use Stancl's `tenant()->run()` to execute closure in each tenant's context rather than manually filtering tenant data.

- Where single-DB scoping is used, ensure all **secondary models** that might be queried directly implement Stancl's `BelongsToPrimaryModel` trait or are only accessed via parent relations. For example, if `TenancyAgreement` belongs to a `Property` (which belongs to tenant), calling `TenancyAgreement::where('expires_at','<', today())` should ideally be done via the property relation or using `tenant()->run()`, or ensure `TenancyAgreement` is also tagged with the tenant through the `BelongsToPrimaryModel` approach. Otherwise, it might pull agreements from all properties. This level of detail will be checked model by model.

- **Central vs Tenant Data Clarity:** We will document which data lives in the **central database** (likely Stancl's tenant registry table, perhaps a central users table if implemented, and maybe some global settings) vs in **tenant databases**. This helps developers know where to put new data. For example, if implementing a global feature like an admin console for the SaaS owner, that belongs in central app; vs. agency-specific data goes into tenant DBs.

- Check that Stancl's central and tenant DB connections are properly configured in `config/tenancy.php` and that central models (like Tenant model, perhaps central User) use the central connection explicitly.
- If using central user authentication, consider moving to tenant-specific users for simplicity or vice versa. (Often, multi-tenant SaaS keep users central to allow one login to multiple tenant accounts, but here it might not be needed since an agent usually belongs to one agency. If not needed, having users in tenant DB increases isolation. We should align this strategy with the product's needs).

In summary, our audit found **minor data leaks due to scoping issues**, which we will address by **tightening code practices** and likely adopting a **multi-database tenancy** approach for stronger isolation. We will fix any specific instances of leaks (contacts, properties sharing) immediately by adding proper tenant scoping in queries or adjusting model configurations.

Migration & Feature Merge Roadmap (AKTONZ Admin into Savarix)

To **merge all required features from the AKTONZ Next.js admin** into Savarix, we propose a phased roadmap with clearly defined epics and tasks. This ensures we systematically reach feature parity and provide AKTONZ (and other agencies) a unified, Savarix-based experience. The plan also covers adopting Savarix's Blade/Tailwind UI style for all features, eliminating the legacy UI. Below is the high-level roadmap:

Short-Term (Immediate) – Phase 1: Core Feature Parity & Data Migration

Goal: Enable AKTONZ to fully operate on Savarix as soon as possible by implementing the most critical missing features and migrating existing data. Retire the Next.js admin once these are in place and tested.

- **Epic 1: Contact & Property Data Integration** – *Complete the core CRM entities in Savarix.*
- **Contact Management:** Verify that Savarix's contact module covers all needed fields (name, phone, email, address, notes, contact type, etc.). Add any missing fields present in Apex27/AKTONZ (e.g., contact categories or statuses like *Lead*, *Client*, *Applicant*). Implement features to import and display those fields. Ensure contact search and filters work (and are tenant-scoped).
- **Property Management:** Expand the property model as needed (e.g., include fields for status – available, under offer, let agreed – and link to owner/landlord which should be a contact). Include any fields AKTONZ had on properties (perhaps a reference code, or custom tags). Implement listing pages and detail pages in Blade with Savarix styling.
- **UI/UX:** Recreate any important UI from Next.js for contacts/properties in Blade. For example, if the Next app had a specific layout or quick-edit modals, implement similar functionality using Blade/Alpine.js or Livewire for interactivity. Use Tailwind CSS to match Savarix's design.

- **Testing:** Branch `feature/contacts-properties` – Write tests to ensure a user in AKTONZ tenant can create/edit contacts and properties and only see their own. Include a test for no

cross-tenant data. After implementation, have AKTONZ staff do UAT on contacts/properties in Savarix.

- **Epic 2: Tenancies & Lease Management** – *Introduce a basic tenancy (lease) module in Savarix to handle ongoing lettings.*

- **Tenancy Records:** Implement a **Tenancy** model (if not already) linking a Property, a Tenant (contact who is renting), and possibly a Landlord (contact who owns, if not implicit from property). Fields: start date, end date, rent amount, deposit, active status, etc.
- **Migrating Existing Leases:** AKTONZ likely has ongoing tenancies in Apex27. We will migrate those into Savarix (see Data Migration section) and ensure they display correctly.
- **Rent Tracking (Basic):** Provide at least a read-only view or manual entry of rent payments for each tenancy. Even if automated invoicing isn't ready, allow agents to log payments or mark rent status (paid/late). This ensures AKTONZ can track rent within Savarix immediately.
- **UI:** Create a Tenancies menu in Savarix for AKTONZ. Show active tenancies, upcoming renewals/expirations, and allow adding new tenancy records. The UI should resemble Savarix style (perhaps similar to properties list).
- **Checklists:** Ensure that when a property is let (tenancy created), the property's status updates (e.g., marked as "Let Agreed" or "Occupied"). Also ensure tenancy records are tenant-scoped.
- **Testing:** Branch `feature/tenancies`. Write tests for creating a tenancy under AKTONZ tenant and verifying it's not accessible under another tenant. Test data integrity (e.g., deleting a property should maybe warn if a tenancy exists).

- **Note:** This is initially a **lightweight tenancy module**. Advanced features like automated rent reminders or document generation can be in later phases.

- **Epic 3: Offers and Sales Pipeline (Minimal Viable)** – *Implement a way to record property offers for sales.*

- **Offer Entity:** Create an **Offer** model linking to Property (for sale) and a Contact (the buyer making an offer). Fields: offer amount, date, status (offered/accepted/rejected), and any notes. This allows AKTONZ to log offers received on a property within Savarix.
- **Sales Pipeline:** Provide a simple interface to move properties through sale stages (e.g., available -> under offer -> sold). This might be as simple as setting property status and logging an accepted offer. Full chain management is not immediate, but at least capture accepted offer and buyer info.
- **UI:** On a property detail page (if property is marked as *For Sale*), show a section for Offers. Allow adding a new offer (with buyer contact and amount) and updating status. Use Savarix styles. If the Next.js admin had any feature for this (or if they relied on Apex27), we ensure Savarix can now handle it.

- **Testing:** Branch `feature/offers-sales`. Test that offers can be created and are tied to the correct tenant's property only.

- **Epic 4: Calendar & Viewings (Basic)** – *Ensure viewings can be scheduled and tracked.*

- **Viewings Module:** If not present, create a **Viewing** model: links a Property, a Contact (who will view, e.g., potential buyer/tenant), date/time, status (upcoming, done), and feedback notes.
- **Scheduling:** Provide a form to schedule a viewing on a property (date picker, assign a contact). For now, this can be a manual scheduler without automated conflict checking.

- **Notifications:** In the short term, we might skip SMS integration, but we can send a simple email confirmation to the contact or have a printable viewing confirmation. At minimum, record that a viewing is scheduled and list it.
- **UI:** Possibly a calendar view or just a list of upcoming viewings. The Next.js admin might not have had a full calendar; if needed, integrate a simple calendar library or use a table grouped by date.

- **Testing:** Branch `feature/viewings`. Write tests for scheduling a viewing and ensure it appears in the list and doesn't show up for other tenants.

- **Epic 5: Data Migration Execution (AKTONZ)** – *Migrate AKTONZ's existing data (from Apex27 or legacy DB) into Savarix's AKTONZ tenant.*

- This epic involves writing and running the migration scripts/commands outlined in the Data Migration Plan section below. It's listed as an epic here because it's a one-time but significant effort in Phase 1.
- **Tasks:** Develop a Laravel **artisan command** (e.g., `tenant:import-aktonz`) that imports all necessary data for AKTONZ:
    - Import contacts, properties, tenancies, etc., mapping old IDs to new.
    - Ensure referential integrity (contacts linked to properties, etc.) in the new system.
    - Run in a transaction or with robust error handling to avoid partial imports.
    - Possibly allow a "dry run" mode for testing (logs what would be done without committing).
- **Testing/Validation:** Run the import on a staging environment first. Have AKTONZ staff verify that all data appears correctly in Savarix (spot-check counts and random records). We may run the import multiple times (if new data trickled into legacy during transition) – so plan idempotency or at least the ability to wipe and re-import on test runs.

- **Cutover:** Plan a cutover day/time where AKTONZ stops using Apex27 (or legacy app), final data sync runs, and then they start in Savarix exclusively.

- **Epic 6: UI/UX Unification** – *Retire the Next.js front-end in favor of Savarix's Blade UI everywhere.*

- For any remaining functions that AKTONZ staff used the Next.js admin for, ensure there's an equivalent Blade page in Savarix:
    - If the Next app had a dashboard with some KPIs, create a dashboard view in Savarix showing key metrics (e.g., number of new leads this week, upcoming viewings, etc.). This can use existing data and be a simple stats page initially.
    - If any settings or admin configurations were managed in the Next app, port those to Savarix (likely not many, since Next was just an admin UI).
- Ensure the visual design follows Savarix's styling (Tailwind classes, layouts). Remove any dependency on the Next.js app – i.e., the user should not need to go to aktonz.com/admin for anything; all done at aktonz's Savarix domain.
- Communicate to users that the unified Savarix app is now the single interface. Decommission the Next.js app (archive the repo, remove it from deployment after successful transition).

Deliverable of Phase 1: AKTONZ can do **all daily operations** (managing contacts, properties, viewings, offers, and current tenancies) inside Savarix with confidence. Their historical data is in place. The Next.js admin is deprecated. The UI is consistent. This phase ends with a deployed update of Savarix and all AKTONZ users trained on the new system. We'll gather feedback from them to tweak any usability issues quickly.

Mid-Term – Phase 2: Enhancements and Two-Way Sync

Goal: After the initial migration, focus on enhancing the system with deeper integration and quality of life improvements that were not blockers for go-live but are important for efficiency. Also, implement the **Apex27 two-way sync** to keep Savarix and Apex27 data in harmony (or begin to replace Apex27 features as needed).

- **Epic 7: Apex27 Ongoing Sync (Two-Way Integration)** – *Build the continuous integration with Apex27's API.*
- **One-way Import Jobs:** Implement scheduled jobs to regularly **pull updates from Apex27** for AKTONZ (and any other tenant using an external CRM). For example, set up a nightly job (or more frequent) to call Apex27 API for any new or modified contacts, properties, or tenancies since last sync, and update Savarix's database accordingly. Use the integration design described later (Apex27 Integration Plan) – including storing `external_id` for mapping and handling differences.
- **Two-way Sync:** Extend the integration so that changes made in Savarix (for AKTONZ tenant) are **pushed to Apex27** via its API. For instance, if an agent edits a contact's phone number in Savarix, the system should send an update to Apex27 so that Apex27 remains up-to-date. This might be done in real-time via model observers (on save, dispatch a job to Apex27 API) or in batch (collect changes and sync periodically). Two-way sync ensures AKTONZ can still use Apex27's other tools (like their client portal or marketing features) without data divergence.
- **Sync Monitoring:** Create logs or an admin panel to monitor sync status – list last sync time, any errors from the API, and allow manual triggering of sync for admins. This is important to troubleshoot issues (e.g., if Apex27 API is down or returns an error for a record).
- **Expand Entities:** In addition to contacts/properties/tenancies, consider syncing **viewings and offers** if Apex27 has those via API (so that external portal scheduling or offer data remain consistent). If Apex27 doesn't expose offers, this might be internal only. But viewings likely do sync (since Apex27 might send viewing confirmations to clients, Savarix should get those updates).
- **Testing:** Thoroughly test with test data on Apex27 sandbox if available. Ensure that creating a record in Apex27 ends up in Savarix after the next sync, and vice versa, creating in Savarix appears in Apex27. Address any field mismatches or required fields differences.

- **Documentation:** Document how the integration works for future maintenance (API endpoints used, data mapping, any webhooks if used, etc.). Possibly implement it as a separate service class or even a Laravel package for reusability if more tenants use it.

- **Epic 8: Advanced Tenancy & Finance Features** – *Grow Savarix's lettings management capabilities.*

- **Rent Invoicing Automation:** Build on the Tenancies module to automatically generate rent invoices and reminders. For example, when a tenancy is active, schedule creation of monthly rent invoices, record if paid, and send email reminders to the tenant contact. This mimics Apex27's rent collection features [5]. Possibly integrate with a payment system (Stripe, GoCardless) for online rent payments in the future.
- **Deposits & Compliance:** Add fields for recording deposit taken, deposit protection info (if required in jurisdiction), and tenancy documents (contracts). Not all need full automation now, but provide a place to store/view these so Savarix covers more of what Apex27 offered.
- **Maintenance Requests:** Though not explicitly asked, a common feature (portal related) is tenants logging maintenance issues. Without a client portal, this may be out of scope for now, but Savarix could allow agents to log maintenance issues and track resolution as an internal feature. Mark for future if not immediate.

- **Sales Progression (Chain Management):** If AKTONZ does sales, implement a simple sales progression tracker. E.g., once an offer is accepted, track milestones (survey done, contracts sent, exchange, completion). This could be a simple checklist on the property or a timeline. Apex27 touts chain management [20], which is advanced – we might not replicate fully now, but provide basic progression tracking to avoid losing that context.
- **Reporting:** Introduce some key reports in Savarix: e.g., a report of all active tenancies with next rent due, a contacts report, etc. This helps wean off Apex27's reports. Use Laravel Excel or similar for export functionality to CSV.

- These enhancements fortify Savarix as a standalone product, reducing reliance on Apex27 beyond the integration period.

- **Epic 9: Multi-Tenancy Hardening & Admin Tools** – *Improve the multi-tenant architecture and provide internal tools for maintenance.*

- **Tenant Switcher for Admins:** Implement an internal tool (for the SaaS owner or developer use) to easily switch into a tenant's context for support. Stancl provides a way to impersonate tenant sessions. This helps support AKTONZ or others without custom domain hopping. Ensure this is secure (only allowed for master admin roles).
- **Audit Logs:** Use a package (like Spatie Activitylog) to record key actions per tenant (who created a property, etc.). This is especially useful in multi-tenant environment for troubleshooting and accountability. We will ensure any such package is set up with multi-tenancy in mind (writing logs to tenant DB or tagging them).
- **Scaling & Perf:** Evaluate if any queries need indexing or optimization now that data grows. Multi-DB helps each DB be smaller, but ensure central operations (like listing all tenants) are indexed. Possibly implement caching where safe (e.g., cache property list filters per tenant).
- **Stancl v3 to v4:** Keep an eye on the tenancy package version. If a newer major version (v4) offers improvements, plan an upgrade after critical phases, to stay up-to-date and supported.

- **Tech Debt Cleanup:** Address any shortcuts from Phase 1 (e.g., if some validations were skipped or error handling minimal to move fast). Refactor code where needed for maintainability. Ensure all new modules have adequate test coverage.

- **Epic 10: UI/UX Enhancements** – *Polish the UI and add missing pieces for better user experience.*

- **UX Improvements:** Now that all features exist, refine the user flow. E.g., add quick actions: a button on property to "Add Viewing" directly rather than going to a separate form page, etc. Possibly introduce Livewire or Alpine.js for smoother interactions (modals for editing, etc.).
- **Consistent Design:** Review CSS and components to ensure consistency. Maybe introduce a Tailwind component library or design system so that forms, tables, alerts, etc., look uniform across the app. Remove any remnants of mismatched styles from the Next.js or default Breeze look if present.
- **Responsive Design:** Ensure the app works on various screen sizes (Breeze is responsive by default, but our custom additions should be checked). AKTONZ agents might sometimes use tablets or phones; make sure core features are accessible on mobile (at least basic responsiveness).
- **User Settings & Profile:** If not added, allow users to edit their profile (name, email, password) within the app, and possibly some agency settings (like agency name, logo, etc., which might have been in a Next.js settings page). This is low-hanging fruit to improve experience.

- **Notifications:** Implement flash notifications or email alerts for key events (e.g., notify an agent if a new enquiry comes in via Apex27 integration, etc.). This keeps users engaged with the system and reduces need to constantly check lists.
- **Client Portal (Planning):** Start design discussions for a future client portal within Savarix (to replace Apex27's portal eventually). This is likely a long-term item, but we can outline how a landlord or seller might log in to see limited data. Possibly use Laravel's multi-auth or a separate guard for external users.

Deliverable of Phase 2: A more mature Savarix application with enhanced letting/sales features, robust integration syncing, and a polished UI. AKTONZ should feel improvements over time (less manual work, more automation). Other tenants (e.g. LCI) can also benefit from these improvements as they roll out. At this stage, Savarix should handle 90%+ of what Apex27 was used for, meaning the dependency on Apex27 could be optional.

Long-Term – Phase 3: Competitive Differentiators and Scaling

Goal: In the long run, make Savarix not just catch up to Apex27, but exceed it in certain areas or provide unique value. Also ensure the platform scales for more tenants and possibly open it for onboarding new agencies.

- **Epic 11: Full Client Portal Integration** – *Replace Apex27's client portal with Savarix's own.*
- Develop a **Landlord/Tenant Portal** module. This could be a separate front-end (could use a lightweight front-end or just Blade with a different layout) where landlords or vendors can log in to see their property status, viewings, documents, and tenants can log in to see rent statements, etc. This reduces reliance on Apex27's portal. It's a large effort: includes creating a user type for clients, secure access control, and exposing relevant data in read-only forms.
- Possibly integrate a helpdesk or maintenance request system in the portal, so tenants can report issues, and those get tracked in Savarix.
- Provide file-sharing (tenancy agreements, invoices) through the portal.

- This would likely be an entire project in itself, but it positions Savarix as a full solution.

- **Epic 12: Third-Party Integrations** – *Match or exceed Apex27 in external integrations.*

- Implement feeds to property listing portals: e.g., Rightmove Real Time Data Feed, Zoopla, etc., so that properties added in Savarix can be automatically uploaded to those portals (if agencies use that). This involves exporting property data in XML/format required and possibly an FTP/API transfer.
- Social media integration: allow one-click posting of a new listing to Facebook/Twitter. Could use their APIs or Zapier integration.
- Calendar integrations: allow agents to sync their Savarix viewings to Google Calendar or Outlook (via iCal feeds or API). This helps adoption.
- Other CRMs: If needed, adapt the integration system to allow connecting to different CRMs besides Apex27 (making Savarix an aggregating platform). Or integrate to other services like email marketing tools for campaigns.

- These integrations expand Savarix's appeal and make it a central hub for the agency's tech.

- **Epic 13: Scalability & SaaS Readiness** – *Prepare Savarix for onboarding many agencies and handling growth.*

- Harden multi-tenancy for scale: If using multi-DB, ensure connection pooling or efficient management of many connections. Evaluate using separate DB servers for large tenants. Implement monitoring for each tenant's usage (perhaps add metrics per tenant).
- Improve multi-tenant management in-app: provide admin UI for the SaaS owner to add new tenants, manage tenant plans, etc. Possibly integrate billing (if Savarix will be sold commercially). Stancl has some support (could integrate Cashier for subscriptions on central app, etc.).
- Codebase maintenance: refactor modules into perhaps domains or modules if the project grows large (for example, use Laravel modular structure for separating CRM, Listings, Tenancies, etc., to keep code organized).
- Continue to keep dependencies updated (Laravel upgrades, package upgrades).

- Security review: perform penetration testing for multi-tenant issues and general security (especially before opening to more clients). Ensure each tenant's data is absolutely secure.

- **Epic 14: Optional CRM Feature Enhancements** – *Add any remaining advanced CRM features.*

- Built-in workflow automation: e.g., allow Savarix users to create simple automation rules (similar to Apex27's configurable workflows [21]). For instance, "when a new lead is added, remind me in 7 days if not contacted". This keeps Savarix competitive with modern CRMs.
- Analytics dashboards: add charts and visual analytics in the app, possibly with a JavaScript chart library. Over time, accumulate data for insights (top performing agents, average time to let, etc.).
- AI/assistant features: could differentiate by adding things like automated suggestions (e.g., "This lead has been inactive for 30 days, consider reaching out"), but that's forward-looking.

Each of these long-term epics can be broken into smaller projects and scheduled as needed. The idea is to ensure Savarix not only replaces the legacy system and Apex27 but evolves into a **cutting-edge SaaS for estate agencies**.

Git Workflow & Branching: For all the above epics, we will use a feature-branch workflow: - Create dedicated branches for each epic or major feature (`feature/<name>`), as noted. Use Pull Requests to merge into the `develop` or main branch after code review. - Keep PRs focused (if an epic is too large, break into smaller PRs per sub-feature). E.g., Epic 7 (Apex27 Sync) might have one PR for one-way import, another for push sync. - Tag releases by phase (e.g., a release tag when Phase 1 is done, etc.). This helps rollback if needed. - Write **unit and feature tests** for critical functionality, especially around multi-tenancy (use Stancl's testing tools to simulate tenant contexts). - Use a staging environment for AKTONZ to test new features before production. This is crucial when introducing two-way sync or new financial features. - Ensure database migrations are run properly for each tenant DB (Stancl's `tenants:migrate` helps) and coordinate downtime if needed for big changes (like splitting databases or initial imports).

Testing Strategies: In addition to automated tests, we plan user acceptance testing with actual agency staff for each module. Particularly for data migration and sync, we'll have parallel runs and manual verification by comparing to Apex27 data until we are confident. We'll also monitor logs after go-live for any errors in jobs (especially sync jobs) and quickly patch issues.

Data Migration Plan (AKTONZ Data into Savarix)

A critical step is the **full data migration** of AKTONZ's existing data into the Savarix system. This ensures continuity of business with no data loss. We will handle data from the legacy systems – primarily Apex27 (since AKTONZ's data resides there) and any additional data the Next.js admin stored.

Data Sources: Based on context, AKTONZ's current authoritative data likely resides in **Apex27's database** (accessible via API or export). The Next.js admin might not have its own database for domain data (it may just call Apex27 API). However, if any data was stored locally in the Next.js app (perhaps some custom fields or settings), we will extract those as well (perhaps from a database or JSON if any). We also have any **CSV exports** or dumps provided from Apex27. The plan will assume we have either direct API access to Apex27 data or exported files for contacts, properties, etc.

Migration Strategy: We will create a Laravel **command** or a set of commands to import data into Savarix's database. This will be done in a careful, stepwise manner:

- **Identify Entities and Mapping:** The primary entities to migrate (and their mapping to Savarix) are:
- **Contacts:** All client contacts (including landlords, tenants, buyers, vendors, applicants – basically any person or company AKTONZ has in Apex27). In Apex27, contacts might have types or tags indicating if they are a landlord, applicant, etc. We'll likely have to map those to Savarix's way of distinguishing (perhaps a field or separate boolean flags).
    - *Mapping:* Apex27 Contact -> Savarix Contact. Fields: name, email, phone, address, etc., map one-to-one. Any Apex27 contact ID will be stored in `contacts.external_id` (with an indicator of origin, e.g., `external_source = 'apex27'`). If Savarix has separate tables for person vs company, map accordingly (Apex27 likely has a flag or separate fields for company).
    - *Related:* Apex27 might link contacts to properties (e.g., a landlord linked to a property, or an applicant linked to a viewing). We must import contacts first so that other records can reference them via the new IDs.
    - *Normalization:* Ensure no duplicate contacts get created. Apex27 might have duplicates or linking by email. If the same person exists multiple times, we might consolidate or at least note it. Initially, import as-is to avoid missing data, we can de-duplicate later if needed.
- **Properties:** All property records (both sales and lettings) that AKTONZ deals with.
    - *Mapping:* Apex27 Property -> Savarix Property. Fields: address, type (we need to map Apex27's property type taxonomy to Savarix's, e.g., house, flat, commercial, etc.), status (available, sold, let agreed, etc.), price or rent, and linked owner.
    - If Apex27 has separate fields for sale price vs rent, we map accordingly and mark property as sale or rental type. Possibly split into two models in Savarix or a field indicating listing type.
    - Link to contacts: e.g., landlord (owner) and current tenant for rentals. Apex27 likely associates a landlord contact to each property (for lettings) – ensure to link that in Savarix (property might have a `landlord_id` referencing contacts, which must be set). For sales, property might have a vendor contact.
    - *Assets:* If possible, import photos or documents. Apex27 might not give direct file export easily. This might be deferred or require downloading files via API if provided. For initial migration, focus on core data; we can later bring in media if feasible by scripting downloads from Apex27 (using their integration or asking them for a bulk export).
- **Tenancies (Leases):** Active and past tenancy agreements for AKTONZ properties.
    - *Mapping:* Apex27 Tenancy/Lease -> Savarix Tenancy. Fields: property (link by some property ID), tenant (link to contact), start/end dates, rent amount, deposit, etc.

- We also want to import **current balance or arrears** if Apex27 tracks rent paid to date. If Apex27 provides transaction or payment logs, we might at least import the last paid date or mark if any rent is overdue (so agents have context). Alternatively, import the schedule of rent (e.g., monthly rent due dates) if available. This can be complex; a simpler approach: mark all tenancies as up-to-date as of migration date, and handle going forward in Savarix. But better is to bring at least a record of last rent paid or next due date.
- *Dependencies:* This requires contacts and properties to be migrated first (to resolve foreign keys).

• **Viewings:** Upcoming and recent past viewings scheduled.

- *Mapping:* If Apex27's API or export provides viewings (likely yes, since it has a viewings manager), import those into Savarix's Viewing model. Fields: property, contact (viewer), date/time, and maybe outcome/feedback notes.
- We might not import *all history* of viewings (if not needed), but at least upcoming ones and maybe the last few months for context. This keeps agent calendars intact. Past viewings could be skipped if not critical, to save effort, unless AKTONZ wants them.
- Ensure to not import viewings that are for properties that are not imported (shouldn't happen if we import all properties).

• **Offers:** If data on offers (for sales) exist in Apex27, import them.

- *Mapping:* Fields: property, offer amount, offer status, date, and linked contact (buyer). Apex27 might have an "offers" section. If accessible, bring in at least active offers or those on unsold properties.
- This gives continuity in case an offer was in negotiation during the switch – agents can see it in Savarix.

• **Users/Agents:** The AKTONZ staff user accounts.

- If using central Breeze users, possibly we already have AKTONZ users created in Savarix for login (maybe they used it partially). If not, we must create user accounts in Savarix for each staff member that was using the Next.js admin (maybe re-use their same emails and set passwords). If we can't get hashed passwords, we'll send password reset links to set new ones.
- Also assign appropriate Spatie roles (e.g., Admin vs Negotiator) to match what they had. Probably all AKTONZ users in Next were admins or had similar rights; we might define roles in Savarix and assign accordingly.
- If AKTONZ had more fine-grained roles in Apex27, mimic those via Spatie roles (for now possibly just "agency_admin" and "agent" roles).

• **Miscellaneous:** Any other data such as notes, tasks, documents:

- *Notes:* If Apex27 allows notes on contacts or properties, and those are accessible, consider importing at least recent notes into a notes table in Savarix (linking to the respective model). This could be important client info that agents need. If not directly available, it might be acceptable to skip or manually copy critical notes.
- *Tasks/Reminders:* Apex27's configurable workflows might have pending reminders or tasks. These might not be easy to extract. Unless specifically requested, we might not import tasks, but agents might recreate key reminders in Savarix or simply rely on Apex27 until fully off.

    - *Attachments:* Document files (IDs, contracts, etc.) – as mentioned, these are tricky to migrate automatically. We might do a partial manual migration: e.g., for each tenancy, upload the tenancy agreement file to Savarix if Apex27 had it. If volume is low, manual effort could handle it. Otherwise, postpone this to a later step or provide a way for agents to upload important docs into Savarix over time.

- **ID Mapping & Preservation:** We will not preserve the exact numeric IDs from Apex27 or legacy, because Savarix will generate its own primary keys. Instead, we maintain a mapping table or within each record, an `external_id`:

- The import command will maintain in memory (or in a temporary table) a map like: `old_contact_id -> new_savarix_contact_id` for all contacts. Same for properties, etc. This way, when linking (e.g., assign a landlord to a property), it can find the correct new ID from the old.
- We may create a simple mapping table structure in the DB, e.g., a table `import_mappings` with columns: `external_source` (e.g., 'apex27'), `external_type` ('contact','property',...), `external_id`, `new_id`. This table can be populated during import for reference.
- Also, adding a column `external_id` in each model's table (with an index) can be useful long-term for sync (as discussed in integration plan). This avoids needing a separate table for mapping after migration; the model itself knows its original Apex27 ID.

- Note: if Apex27 IDs are globally unique across all their clients or at least across entity type, it's fine. If not (which we assume not globally unique outside context), we combine with a source identifier.

- **Normalization & Data Cleaning:** Before inserting into Savarix, we may clean some data:

- Ensure consistent formatting (e.g., phone numbers, uppercase postcodes, etc.) as per any validation rules Savarix has.
- If Apex27 allowed free-text where Savarix expects a specific format (e.g., property type as predefined options), map or adjust values to fit Savarix's enumeration to avoid invalid data.
- Remove obviously orphaned or duplicate data (e.g., contacts that were entered twice – though merging duplicates automatically is risky, so maybe just flag them for later review rather than merge during migration).

- For addresses, maybe use a consistent format (if Savarix uses a single address field vs multiple, combine or split accordingly).

- **Migration Execution Order:** The order of running the migration is crucial:

- **Users** (Agents) – so that any created by data import can be associated (though likely not needed to link in data, but for completeness).
- **Contacts** – import all contacts first. Assign them new IDs.
- **Properties** – import all properties. Use contact mapping to link landlord/vendor to property's `owner_id` (for example). If property refers to "current tenant" (for rentals) in Apex27, that is actually given by an active tenancy record rather than a property field, so we'll handle that in Tenancies step.
- **Tenancies** – import leases. Use property mapping and contact mapping for tenant to create each tenancy. Also link the landlord if needed (though landlord is already on property). If Apex27 had a concept of tenancy ID separate from property, maintain `external_id` for tenancy too if needed.

- **Offers** – import offers (after properties and contacts are in). Link to property and contact (buyer). Possibly also link to a contact representing the vendor (which is property.owner in our scheme).
- **Viewings** – import viewings (after property & contact). Link accordingly.
- **Notes/others** – after main entities, attach notes or other aux data linking to the new record IDs.

We will wrap each of these in transactions per entity batch or per record as appropriate. However, given potentially large volume, we might commit after each entity type to avoid one giant transaction. Instead ensure referential integrity by careful ordering and coding.

- **Migration Implementation (Laravel command):** We'll create perhaps one command `php artisan tenant:import-aktonz` which inside connects to the source (either reads CSV files or calls Apex27 API). Pseudo-code structure:

```
// assuming tenant identified by code or domain, e.g., 'aktonz'
Tenancy::initialize($aktonzTenant); // switch to AKTONZ tenant DB if
multi-DB
importContacts();
importProperties();
importTenancies();
importOffers();
importViewings();
```

  Each `importX()` will read the source data and do creates. Using Eloquent for convenience (with mass assignment disabled for safety, we'll fill attribute by attribute ensuring no injection). If performance is an issue (large data), consider chunked inserts or direct DB queries, but likely manageable.

We'll include options in the command for flexibility: - `--dry-run`: to simulate and report how many records would be created, without committing, for testing. - `--truncate-first`: to clear existing AKTONZ data in Savarix tenant (in case we need to re-run the import cleanly on a test tenant). Use with caution on production.

Logging: The command will output progress (e.g., "Imported 500 contacts", "Imported 200 properties", etc.) and log any errors (e.g., "Property X skipped due to missing owner reference").

- **Safe Execution & Testing:**
- We will first run the import on a **staging environment**. Possibly with AKTONZ tenant in staging and maybe pointing to a sandbox or static export of Apex27 data. Verify the counts: e.g., if Apex27 had 1000 contacts, ensure roughly 1000 contacts in Savarix after (differences might occur if some were filtered or combined).
- Spot-check random records: pick a contact in Apex27 export and find it in Savarix, confirm name, email, etc., came through correctly. Do this for each entity.
- Verify relational links: e.g., a tenancy in Savarix has property and tenant linked correctly by names.
- After confirming, schedule the production migration during off-hours. Ensure a backup of Savarix DB (and if needed Apex27 data) is taken before running, in case rollback is needed.

- Downtime considerations: If multi-DB, we can import while other tenants live possibly because it's just one tenant's data being inserted. But likely we'll put AKTONZ in read-only mode briefly or do it after hours to avoid any conflicts.

- **Post-Migration Cleanup:**

- Once data is in Savarix, communicate that from a certain point onward, **Savarix is the primary system**. Ensure no one adds new data in Apex27 in parallel. If they do (in the gap), we either have to re-run or manually bring those few entries.
- We will double-check critical data after migration: e.g., all active tenancies are present, all currently available properties exist, all contacts of type landlord/tenant are there. If something is missing, we can still fetch it and add.
- The `external_id` mappings will now support the integration: moving forward, the sync jobs know what was already imported (so they don't duplicate).
- We may keep Apex27 as read-only reference for a short period in case something wasn't migrated, but ideally we aim to migrate everything needed.

Normalization and Testing Commands: We may write additional small commands to verify data consistency after migration. For example: - A command to verify that every property that should have a landlord has one, every tenancy's property and tenant exist. - Or even a script to compare Savarix data to Apex27 via API for a few random records as a sanity check.

Rollback Plan: In the unlikely event that something goes wrong (data mis-migrated or major issues), the fallback would be: - Keep Apex27 running in parallel until confidence is achieved. If needed, AKTONZ could briefly return to Apex27 if Savarix had an outage. But once we cut over and start editing data in Savarix, going back is hard unless we sync back changes. - More practically, fix forward: if some data errors are found, write scripts to correct them in Savarix rather than rolling fully back. - That's why thorough testing beforehand is crucial to avoid this scenario.

By executing this migration plan, AKTONZ's historical and active data will be in Savarix's database, enabling a smooth transition. We ensure that after migration, **Savarix contains all entities AKTONZ is accustomed to**, so there's minimal disruption in their operations.

Apex27 CRM Integration Design (One-Way Import and Two-Way Sync)

To integrate Apex27 with Savarix, we will design a robust sync system. The objective is to **initially import** all necessary data from Apex27 (covered in migration) and then keep Savarix and Apex27 **in sync** moving forward. This allows AKTONZ (and any other tenant using Apex27) to use Savarix as their primary interface without losing data or functionality from Apex27.

Integration Overview: We will build a dedicated component (e.g., an `Apex27Service` in Laravel) to communicate with the Apex27 API. This will handle making HTTP requests to fetch or push data. We will also implement **scheduled jobs** and possibly event listeners for ongoing sync. The integration will cover key data objects: Contacts, Properties, Tenancies, Viewings, Offers (as available via API).

Initial One-Way Import: (This overlaps with the migration plan) - We have already planned a one-time import of all Apex27 data for AKTONZ into Savarix. This can be done via the API or data export. If using the API: - Use Apex27's API endpoints for fetching all contacts, all properties, etc. If the API has pagination, loop through to get all records. - This can be done via artisan command as described, or even manually triggered from an admin UI. - Ensure rate limiting of API is respected (we might throttle requests if needed).

One-Way Ongoing Updates (Pull): - **Scheduled Pull Jobs:** Set up a schedule (e.g., every night at 2am, or every hour if near-real-time is desired and API allows) to pull new or changed data from Apex27: - Use **incremental updates** if possible: Check if Apex27 API supports querying "records updated since X timestamp". Many CRMs provide a lastModified filter or we can store a last sync timestamp and pull everything updated after that [22] . If available, this significantly reduces load. - If no such filter, we might have to pull all records and compare, but that could be heavy. Alternatively, Apex27 might offer webhooks for changes (to avoid polling). If webhooks exist, we can set up an endpoint in Savarix to receive Apex27 push notifications when a record changes, then update our DB accordingly in real-time. - We will configure a **tenant-specific approach**: Each tenant that uses an external CRM will have their own API credentials stored (likely in a central table `external_credentials` keyed by tenant). The job will iterate through each tenant with Apex27 integration enabled, set the tenant context, then perform sync for that tenant. - For each entity type: - *Contacts:* Fetch new/updated contacts from Apex27. For each, if `external_id` already exists in our contacts table, update the record in Savarix (e.g., if phone number changed in Apex27). If it does not exist, create a new contact in Savarix (this covers cases where someone added a contact in Apex27 after migration). - *Properties:* Same approach. Update existing ones by `external_id` match; create new if found (e.g., a new property added in Apex27). - *Tenancies:* If Apex27 shares tenancy info (maybe in property or separate endpoint), sync any changes in tenancy status or new tenancies. Often agencies won't add a new tenancy in Apex27 if they are now using Savarix, but during transition or if some staff use Apex27 by habit, this covers it. - *Viewings & Offers:* For completeness, fetch any newly scheduled viewings or offers logged in Apex27 (perhaps via portal or another staff) and replicate them. - After pulling, log what was changed. Possibly send a summary email to admins if significant differences (optional, for transparency).

- **One-Way Outbound (Push from Savarix to Apex27):** (This is the second step of two-way)
- We will implement triggers so that whenever certain records are created or updated in Savarix, we call Apex27 API to perform the equivalent operation:
    - For example, after a new **Contact** is created in Savarix (within AKTONZ tenant), a Laravel event (like `ContactCreated`) triggers a job to POST that contact to Apex27 via API. On success, we get back an Apex27 ID which we store in `external_id` of that contact (if not already stored).
    - If a **Property** is added or edited, similarly push updates (fields like price change, status change) to Apex27.
    - **Tenancies:** If Savarix is used to record a new tenancy or update one (dates, status), consider updating Apex27 as well if they manage that. If Apex27 has no direct tenancy API, we might skip or update property status.
    - **Offers/Viewings:** If agents log offers or schedule viewings in Savarix, pushing those to Apex27 ensures the Apex27 portal (if still used by clients) is aware. For viewings, pushing to Apex27 might trigger their notifications (which could be useful until Savarix does its own).

- **Conflict Avoidance:** We need to carefully manage conflicts to avoid ping-pong updates:

    - If we have both pull and push, there is a risk of a change bouncing back and forth. To mitigate this, implement checks: e.g., when pulling updates from Apex27, for each field, if Savarix also has a newer change that hasn't been pushed yet, decide which to keep. Simplest strategy is to treat one system as primary for certain data. We could declare Savarix as the primary after migration for most data, meaning ideally staff use Savarix only. Then Apex27's changes would mostly come from the portal or automated stuff.
    - Alternatively, use timestamps: If Apex27's updated_at is more recent than Savarix's updated_at for that record, then accept Apex27's changes (overwrite Savarix). If vice versa and we have already pushed, then skip or maybe re-push if needed. Maintaining a

last_sync timestamp and last_sync_direction could help (to know when the last sync happened and who was source).
    - Possibly implement a simple **two-way sync algorithm**: for each record, compare both sides, and merge changes intelligently (could be field by field, but that's complex; we might say the latest edit wins for the whole record).
    - In practice, to keep it simpler, we might encourage a clear cut: during Phase 1 and 2, AKTONZ agents should primarily use Savarix for data entry, and clients might use Apex27 portal. Thus, changes from Apex27 side could mostly be client-entered or automated (like a client updates their contact info on portal). Those we pull in. Changes from agent side, we push out. This reduces concurrent edits.

- **API Error Handling:** We must handle cases where Apex27 API is down or returns errors:
    - Use a queue with retry for push jobs. If a push fails (network or validation error), log it and mark for retry. If it consistently fails (e.g., required field missing for Apex27), alert an admin to manually resolve.
    - For pulls, if API is unreachable at scheduled time, catch exception, and perhaps try later or report. It's not critical to miss one schedule as it will catch next time, but prolonged downtime should alert us.
- **Rate Limits & Performance:** If Apex27 has strict rate limits, we might need to space out requests. For example, if pushing a bulk of changes, group them or limit X per minute. The integration should be mindful of this to avoid hitting caps and getting blocked.

Metadata Tracking: We will store extra metadata in Savarix DB to manage the sync: - `external_id` on each synced model (as discussed). - Possibly an `external_updated_at` timestamp on records to record the last known timestamp from Apex27 for that record (so we know if Apex27's version changed since we last checked). - A table for sync logs: Each time we run a sync job, log timestamp, how many items created/updated for each entity, and any errors. This table helps in debugging and also proving the sync is running. - If using webhooks: a table to log incoming webhook events (to track what came in). - If necessary, a lightweight mapping table for complex mapping (e.g., if Apex27 and Savarix have different enumeration values, store mapping of codes).

Where to place code: - We'll create an **Integration layer** within the Laravel app: - For example, an `app/Services/Apex27/` directory. Inside, classes like `Apex27Client` (for low-level HTTP calls, authentication handling), and separate classes for each entity sync, e.g., `ContactSync`, `PropertySync` handling mapping and transformation logic. - Alternatively, implement it as a Laravel **package** if we want modularity, but likely in-app is fine for now. - Use repository pattern or direct model calls as convenient, but likely direct since we are within Laravel already. - We will also have **Jobs** (in `app/Jobs`) such as `SyncContactsFromApex27`, `PushContactToApex27`, etc., to encapsulate each sync operation. These jobs can be dispatched by the scheduler or by model events. - Configuration for API (base URL, API key, etc.) will live in `config/services.php` or a new `config/apex27.php`. Credentials could be stored per tenant in the DB, in which case maybe encrypted in the DB since multiple tenants = multiple keys potentially. We'll build a simple UI for an admin to enter their Apex27 API key in Savarix (or we seed AKTONZ's key directly).

Sync Diagram: The following illustrates the flow of data between Savarix and Apex27 after integration:

- **Initial Import (One-Time):** Apex27 (AKTONZ data) → *Import script* → Savarix AKTONZ tenant database.

- **Ongoing One-Way Sync (Pull):** A scheduled job on Savarix calls Apex27 API (GET endpoints) to retrieve any new or updated records, then updates Savarix DB. This ensures Savarix catches up with any changes made externally (e.g., via Apex27's client portal).
- **Ongoing One-Way Sync (Push):** On certain events or schedule, Savarix calls Apex27 API (POST/PUT) to send any changes made in Savarix to Apex27. This keeps Apex27 up-to-date so that if AKTONZ still uses Apex27's features (reports, portal), the data remains consistent.
- **Two-Way Maintenance:** Both sides maintain an ID map so no duplicates occur. Savarix's `external_id` links to Apex27's IDs to avoid re-creating records. Conflicts are resolved by timestamps or predetermined priority (e.g., Savarix as master for most data after cutover).

In summary, Savarix will act as a central hub, regularly **pulling** from and **pushing** to Apex27. Initially the focus is on pulling to ensure no data loss. Once stable, pushing allows Savarix-led workflows. Over time, if Savarix implements all needed features (client portal, etc.), AKTONZ might phase out Apex27 entirely; at that point we could disable the sync or keep a minimal one for backup. But until then, this two-way integration guarantees that both systems mirror each other's key information.

(No image available – diagram described in text above for clarity.)

Savarix Progress Assessment & Tech Debt Recommendations

Finally, we assess the overall progress of the Savarix agency app, identify any **fragile areas or technical debt**, and propose improvements. This ensures the platform is robust as we add features and tenants.

Current Feature Implementation Status (as observed in Savarix code): - Many core modules are in place but at varying levels of completeness: - **Authentication & Roles:** Implemented via Breeze (Laravel's simple auth scaffolding) and Spatie Roles. This part is solid and production-ready for basic use (login, registration for new tenants, etc.). There might be some enhancements needed (password resets email template, invite users flow, etc., if not already done). - **Multi-Tenancy Framework:** Stancl v3 is integrated, which is a modern and powerful foundation. The tenancy bootstrapping and config exist, but some usage issues (as noted in audit) need fixing. Overall, the structure for multi-tenancy is in place (the hardest part), which is good progress. - **Contacts Module:** Partially complete. Basic CRUD works in the UI with validations. Lacking advanced features like bulk import, duplicate merge, or category tagging. Also likely missing integration triggers (which we will add). This module is usable but could use enhancements (like better search filters, and showing related info like "Properties owned by this contact" etc. in UI). - **Properties Module:** Implemented to allow adding/editing property listings. Has essential fields (address, price/rent, status). Possibly missing richer details (number of bedrooms, features, etc.) which might be needed for a complete listing – we should check if those are intended to be stored and if not, consider adding. The UI is functional (likely a simple list and detail page). We should verify it handles both sales and rental properties distinctly if needed. - **Tenancies/Leases:** A basic model exists but might not be fully wired into UI before our planned Epic. It might be that the schema has a `tenancies` table but no polished UI yet. If that's the case, implementing the UI was left for later (which we will now do). If not present at all, we'll create it as per plan. - **Viewings & Appointments:** Possibly not implemented yet (we didn't see references in the code excerpts). If anything, maybe a placeholder or an unsused migration exists. So likely a missing feature – to be built as part of our integration plan. - **Offers/Sales:** Not evident in current code – likely missing. Sales workflow not implemented yet. - **Dashboard/Reports:** Possibly a simple homepage or dashboard exists (maybe showing a welcome and some counts). Not an advanced reporting dashboard. So that's an area for improvement. - **Settings & Configurations:** Savarix might have some central or tenant-specific settings (like company profile, branding, etc.). If not, that's a minor gap – agencies will eventually want to upload their logo, etc. - **UI/UX State:** Because it's built on Breeze, the UI is functional but somewhat boilerplate. Breeze provides minimal styling. They likely have begun customizing with Tailwind. Some

areas might still have default styling or lack polish (e.g., forms could be unstyled HTML elements etc., if not styled). - **Responsive design and cross-browser:** Needs testing; Breeze is generally responsive, but our added pages need to be checked. We should test the app in different browsers and devices as part of QA in upcoming sprints. - **Email/Notifications:** Not sure if Savarix currently sends any emails (aside from possibly user verification or password reset which Breeze covers). Features like notifying a user of a scheduled viewing probably aren't there yet. We will consider adding such notifications in mid-term. - **APEX27 Integration:** At the moment, likely none (no code for external API calls was noticed). So that's entirely new to add.

Fragile Areas: - **Tenancy Scope Enforcement:** As noted, any places where data was leaking indicates fragility in multi-tenancy handling. This is top priority to fix to avoid data breaches. We will increase automated testing around this to catch any future regressions. - **Spatie Roles Setup:** If the Spatie integration steps (cache separation, tenant migrations) were not fully followed, it could cause issues when multiple tenants and roles come into play. This could manifest as, e.g., roles from one tenant showing up in another's UI if cache not flushed. We must test scenario of two tenants creating roles of same name etc., to ensure isolation. If we find issues, apply Stancl's recommended fix (as in the audit section). - **Code Structure and Maintainability:** Given the fast-paced development, some controllers or services may be growing messy. For example, if all logic sits in controllers, we might want to refactor into service classes or use form request classes for validation to keep things clean. We should identify any overly complex or long methods that need refactoring. - **Testing Coverage:** We need to check test coverage. If the project lacks tests, that's tech debt. Writing tests for existing features will help avoid breaking them as we modify things. We should allocate time in each sprint to add tests for critical flows (auth, tenancy scoping, CRUD operations). - **Stancl v3 Deprecation:** Stancl v3 is stable, but by 2025 there might be v4 or newer. Not urgent, but consider planning an upgrade to ensure long-term support. Upgrading tenancy package can be non-trivial, but we should at least follow the upgrade guide once Phase 1 is done, to not lag behind. - **Performance:** With small data now, all is fine. But as data grows or if one tenant has thousands of records, some queries might become slow if not optimized. E.g., listing 1000+ properties might need pagination (likely implemented) and indices on search fields. We should review database indices and add where missing (e.g., ensure foreign keys have indexes, ensure any commonly searched fields like email, reference numbers have indexes). - **UI Consistency & Layout:** Possibly some pages might not use a consistent layout or have copy-pasted code from Breeze that could be streamlined. Standardizing layouts (master template) and using Blade components for repeated elements (like forms, modals) will reduce future tech debt of UI. - **Naming Confusion:** As noted, the domain language has "tenant" in two contexts (SaaS tenant vs rental tenant). In code, hopefully they differentiate (e.g., the Stancl tenant model vs a Person who is renting). If not, we may consider renaming the rental tenant concept to "LeaseParticipant" or "Renter" in code to avoid confusion. This is something to evaluate to prevent developer mistakes. Possibly they avoided this by calling the Stancl model "Agency" or similar. - **Legacy Code or Stubs:** Check if any leftover or commented code from initial scaffold or experiments is present (like default Breeze routes that might conflict, etc.). Clean those up to avoid confusion.

Technical Debt Recommendations: - **Enforce Coding Standards:** Use Laravel Pint (code formatter) and PHPStan (static analysis) in CI to keep code quality high. This prevents certain classes of bugs and makes the codebase consistent. - **Refactor Where Needed:** For example, if the Property controller is doing too much (handling file uploads, business logic), move some logic to a PropertyService or to model events. If forms are manually handled, maybe use Form Requests for validation to tidy controllers. - **Modularize Permissions:** As features grow, ensure to define and use Spatie permissions consistently for new actions (like viewing a report, etc.). This avoids hard-coding checks and allows flexibility if roles change. - **Documentation:** Create a simple internal developer README or use code comments to explain tricky parts (especially the tenancy stuff). New developers should quickly grasp how tenancy is managed (like "We use multi-DB, so run tenants:migrate, etc."). - **Monitor and Log:**

Implement logging for important events (user login failures, integration errors as mentioned). Possibly integrate Sentry or another error tracking to catch exceptions in production and fix them. This is proactive tech debt management. - **Database maintenance:** For multi-DB, consider writing a script to easily clone schema changes to all tenant DBs in development (Stancl's migrate command covers production, but in dev we might need to ensure new migrations run for sample tenants we have). Also plan for backups of each tenant DB. - **Client Feedback Loop:** Continue to get feedback from AKTONZ (as pilot tenant) on any pain points. If they mention, for instance, "It takes too many clicks to do X", treat that as an opportunity to improve UX (tech debt in UX sense). - **Remove Legacy Artifacts:** Once AKTONZ admin is retired and Apex27 is largely integrated, remove any code that was temporarily used for bridging. For example, if any quick scripts or debug routes were added, clean them up. Ensure the codebase doesn't carry unnecessary weight.

Progress Outlook: Savarix's development so far has laid a solid groundwork (multi-tenancy, basic CRM features). The upcoming migration and integration efforts will rapidly bring it to a full-featured state for AKTONZ. We should be mindful to build in a **scalable, maintainable way** as we add features, to avoid accruing more tech debt. Regular code reviews and refactoring sessions should be part of the process.

Roadmap Recap: We have outlined short-term and mid-term epics to systematically address missing features and integration. By end of Phase 2, Savarix should be quite robust and mostly self-sufficient. Long-term epics ensure we continue improving and not stagnate. The key is to prioritize delivering the immediate needs (Phase 1) without cutting corners that would cause issues later – hence the emphasis on tenancy correctness and data integrity early.

In conclusion, with this plan, we will **successfully merge the AKTONZ admin functionality into Savarix**, achieve a seamless data migration, maintain data integrity through strong tenancy isolation, and integrate Apex27 such that AKTONZ experiences a continuous service. The Savarix platform will evolve from a nascent state to a **production-ready, full-featured multi-tenant CRM** for estate agencies, setting the stage for onboarding more tenants and deprecating any legacy systems.

Sources:

- Stancl Tenancy v3 Documentation – outlining single vs multi-database tenancy considerations [19] [13] and Spatie roles integration for multi-tenancy [15] (used to guide tenancy isolation strategy).
- Apex27 CRM Features (public info) – used to gauge feature completeness (contacts, viewings, invoicing, etc.) [2] [5].

---

[1] [2] [3] [4] [5] [6] [8] [9] [10] [11] [20] [21] Apex27 | CRM Features
https://apex27.co.uk/estate-agent-software-features

[7] [15] [16] [17] Integration with Spatie packages | Tenancy for Laravel
https://tenancyforlaravel.com/docs/v3/integrations/spatie/

[12] [13] [14] [19] Single-database tenancy | Tenancy for Laravel
https://tenancyforlaravel.com/docs/v3/single-database-tenancy/

[18] Multi-database tenancy | Tenancy for Laravel
https://tenancyforlaravel.com/docs/v3/multi-database-tenancy/

[22] Searching Records using HTTP module - How To - Make Community
https://community.make.com/t/searching-records-using-http-module/26636