

Proposition d'architecture Laravel

Passage d'un frontend piloté par JSON (UBCG.boot, data-api.js) à une application Laravel native

Application UBCG — plateforme de gestion interne (CRM, ERP, RH, Finance, Immobilier...)

Objectif : proposer une architecture Laravel 12 plus native, plus simple à maintenir et mieux adaptée à une application dont les écrans sont rendus par Blade, avec des routes, des contrôleurs fonctionnels et Eloquent, et JSON uniquement pour les vrais besoins AJAX et pour l'API mobile/intégrations.

Document de présentation — Développeur et direction

Date : 8 octobre 2026

1. Situation actuelle

Le backend UBCG est un **monolithe modulaire Laravel 12** (23 modules, 151 tables, 775 opérations d'API sous `/api/v1`). Il a été conçu « API d'abord » : chaque module expose des contrôleurs REST qui renvoient des Ressources JSON.

Côté interface, le prototype statique (`ubcg-ui/`, 54 pages HTML) charge un petit chargeur, `assets/boot.js`, qui injecte neuf scripts (`api-client.js`, `data-api.js`, `auth-api.js`, `notify-api.js`...), masque la page derrière un voile, puis exécute `UBCG.boot()` : un appel GET `/me` et les collections dont la page a besoin. Les données sont rangées dans un cache mémoire global (`UBCG.DB`) et chaque page JavaScript fabrique son HTML à partir de ce cache.

Le concept initial (documents 01 et 02) prévoyait de remplacer ces pages par une application **Inertia 2 + Vue 3 + TypeScript**, migrée page par page grâce à un adaptateur `UBCG.DB`.

Architecture actuelle :

Route `/api/v1` → ApiController → FormRequest → Modèle → Resource JSON → `api-client.js` → `data-api.js` (`UBCG.DB`) → JavaScript de la page → DOM

2. Problèmes identifiés

- **Deux applications à maintenir.** Une application Laravel (PHP) et une application cliente (JavaScript, bientôt Vue/TypeScript, Vite, vue-tsc, Vitest, ESLint) qui se parlent par un contrat JSON de 775 opérations. Chaque écran impose de modifier les deux côtés.
- **Un chargeur fragile.** `boot.js` retient les écouteurs `DOMContentLoaded/load` des scripts de page, attend `UBCG.boot()`, puis les rejoue avec un événement synthétique : un mécanisme sans équivalent dans Laravel, difficile à déboguer.
- **Un état global côté navigateur.** `UBCG.DB` duplique la base de données dans la mémoire du navigateur ; il faut gérer son invalidation, ses états périmés et sa synchronisation avec l'API.
- **Des règles dupliquées.** Les permissions existent en JavaScript (`auth.js`, 77 Ko) et en PHP (`PermissionResolver`) ; une parité de 22 915 assertions est nécessaire pour les garder identiques. Les règles de validation sont aussi recopiées dans `api-rules.js`.
- **Des données envoyées puis filtrées côté client.** Le navigateur reçoit des collections entières, puis le JavaScript décide quoi afficher selon le rôle. Avec Blade, le serveur n'envoie que ce que l'utilisateur a le droit de voir.
- **Un débogage long.** Une erreur peut venir de la route, du contrôleur, de la Resource, du client HTTP, de l'adaptateur, du cache ou du JavaScript de la page.
- **Une seconde chaîne d'outils.** Le choix Inertia/Vue ajoute Node, npm, Vite, TypeScript et des compétences Vue, alors que l'équipe est d'abord une équipe Laravel.
- **Une i18n en double.** Le moteur du prototype (clés générées, dictionnaires `assets/i18n/*.js`) et `lang/` de Laravel (erreurs, validation, e-mails) cohabitent.

3. Proposition d'architecture Laravel native

L'objectif n'est pas de supprimer l'organisation modulaire ni la couche API, mais de répartir les responsabilités au bon endroit : **le serveur rend les pages**, JavaScript n'ajoute que de l'interactivité.

Architecture proposée pour l'interface du personnel :

Route (`routes.web.php`) → Contrôleur fonctionnel → Eloquent Model / Relations / Scopes → Vue Blade

Pour les opérations qui nécessitent réellement une communication asynchrone, JSON est conservé uniquement à cet endroit :

JavaScript / AJAX (`fetch` + `CSRF`) → Route Laravel → Contrôleur → Eloquent → Réponse JSON

L'API `/api/v1` reste disponible, **en option**, pour l'application mobile, les portails externes, les webhooks et les intégrations. Elle n'est plus utilisée par les pages du personnel.

3.1 Organisation d'un module (exemple Crm)

```

app/Modules/Crm/
  routes.php          API JSON /api/v1          (inchangé, optionnel)
  routes.web.php      pages Blade              (nouveau, chargé par le module)
  Http/Controllers/   contrôleurs API          (inchangés)
  Http/Controllers/Web/ contrôleurs des pages      (ContactController, ClientController...)
  Http/Requests/      FormRequest partagées API + Web
  Http/Requests/Web/  normalisation des formulaires HTML
  Models/ Policies/   Eloquent (TenantModel) + Policies (inchangées)
  Domain/             règles pures, machines à états (inchangées)
  Services/           seulement la logique réutilisable (import CSV, doublons...)
  Lang/{fr,de,en,ar}/ web.php (pages) + errors.php (codes d'erreur)
resources/views/
  layouts/app.blade.php partials/{header,nav,flash} components/ui/* errors/problem
  modules/crm/contacts/ index.blade.php form.blade.php show.blade.php
  public/css/ubcg.css   public/js/ubcg/{app,transition}.js

```

3.2 Pile de middleware des pages

Le registre de modules découvre `routes.web.php` et applique la pile web de `config/modules.php` : **web** (session, CSRF) → **auth** → **tenant** (résolu depuis l'utilisateur, jamais depuis le client) → **identity** → **locale** → **plan** → **throttle**. Les noms de route sont préfixés `<slug>.web.` (par exemple `crm.web.contacts.index`).

4. Organisation proposée des contrôleurs

Un contrôleur par fonctionnalité métier, dans `app/Modules/<Module>/Http/Controllers/Web/`. Les noms suivent les contrôleurs API existants et les pages du prototype.

Module	Contrôleurs (Http/Controllers/Web)	Responsabilité principale
Identity / Access	LoginController, TwoFactorController, PasswordController, ProfileController, AccountController, RoleController	Connexion, 2FA, mot de passe, profil et langue, comptes, matrice des rôles et surcharges de permissions.
Tenancy / Platform admin	SettingsController, WorkspaceController, PlatformTenantController, HealthController	Configuration de l'espace (départements, équipes, postes), supervision des tenants et de la santé.
Dashboards	DashboardController (home, business, manager, realEstate, hr)	Tableaux de bord : chaque méthode charge ses propres indicateurs.
Crm	ContactController, ClientController, CallbackRequestController	Contacts (liste, fiche, Kanban par transition), clients, demandes de rappel.
Projects	ProjectController, ServiceController, DeliveryNoteController, ProjectUpdateController, NonBillableController	Projets et services, bons de livraison, suivi (monitoring), temps non facturables.
Hr	EmployeeController, LeaveController, TimesheetController, ContractController, CertificateController, EvaluationController, JobOpeningController, JobCandidateController, OnboardingController, OffboardingController, EventController, IdeaController	Cycle de vie des employés : congés, pointage, contrats, certificats, évaluations, recrutement, arrivées et départs, événements, boîte à idées.
Finance	QuoteController, InvoiceController, PaymentController, ChequeController, ExpenseController, ReimbursementController, InventoryController, BankAccountController	Devis, factures (idempotentes), paiements, chèques, dépenses, remboursements, inventaire.
Accounting	LedgerController, SalaryController, BonusController, PeriodController	Grand livre, salaires, primes, clôture des périodes.
RealEstate	ProjectController, UnitController, ViewingController, BookingController, BrokerPortalController, BuyerPortalController	Programmes, unités, visites, réservations ; portails courtier et acheteur.

Module	Contrôleurs (Http/Controllers/Web)	Responsabilité principale
Marketing	CampaignController, PartnerController, PartnerInvoiceController, PartnerPortalController	Campagnes, partenaires, factures partenaires, portail partenaire.
Legal	LegalCaseController, LegalContactController, LegalCostLineController	Dossiers juridiques, contacts, lignes de coût.
Providers	ProviderController, ProviderJobController, ProviderCostLineController	Prestataires, missions, coûts.
Donations	DonationController, BeneficiaryController, CategoryController, ZakatController	Dons, bénéficiaires, catégories, calcul de la Zakat.
Billing	SubscriptionController, InvoiceController, PaymentMethodController, CheckoutController, BillingProfileController	Abonnement, factures UBCG, moyens de paiement, passage en caisse.
Privacy	PrivacyCenterController, DataRequestController, ProcessingActivityController, BreachController, SubProcessorController, LegalDocumentController, MyPrivacyController	Registre RGPD, demandes de personnes, violations, sous-traitants, documents légaux, espace « mes données ».
Seo	SeoDashboardController, KeywordController, AuditController, BacklinkController, PublishController, ConfigController	Administration SEO/GEO : mots-clés, audits, backlinks, publication.
Messaging / Notifications	MessageController, NotificationController (+ unread-count AJAX)	Messagerie interne, centre de notifications, compteurs de l'en-tête.
Audit / Files	ActivityController, CommentController, AttachmentController	Journal de communication, commentaires, pièces jointes (dépôt AJAX, téléchargement signé).
Signup / pages publiques	LandingController, RegisterController, DemoRequestController, PublicLegalController	Page d'accueil, inscription, demandes de démo, pages légales (rendues par Blade avec cache HTTP).

Correspondance routes / vues (exemples) : chaque module a un fichier `routes.web.php` et un dossier `resources/views/modules/<slug>/`.

Page du prototype	routes.web.php (module)	Vue Blade
contacts.html	Crm : GET /contacts, /contacts/create, /contacts/{contact}, .../edit	modules/crm/contacts/{index,form,show}
clients.html, client-form.html	Crm : /clients, /clients/create, /clients/{client}/edit	modules/crm/clients/{index,form,show}
projects.html, project-form.html	Projects : /projects, /projects/create, /projects/{project}	modules/projects/projects/{index,form,show}
employees.html, urlaub.html	Hr : /employees, /leaves	modules/hr/employees/*, modules/hr/leaves/*
finances.html, devis.html	Finance : /invoices, /quotes, /cheques, /expenses	modules/finance/invoices/*, modules/finance/quotes/*
dons.html	Donations : /donations, /zakat	modules/donations/donations/*, modules/donations/zakat/*
privacy-center.html	Privacy : /privacy/requests, /privacy/breaches	modules/privacy/requests/*, modules/privacy/breaches/*

5. Exemple de fonctionnement : Crm / contacts

Cette tranche de référence est **déjà écrite** dans le dépôt backend. Les routes sont déclarées dans `app/Modules/Crm/routes.web.php`, avec une permission `can` : sur chaque route (statiques avant paramétrées) :

```
Route::get('contacts', [ContactController::class, 'index'])
    ->middleware('can:contacts.view')->name('contacts.index');
Route::get('contacts/create', [ContactController::class, 'create'])
    ->middleware('can:contacts.create')->name('contacts.create');
Route::post('contacts', [ContactController::class, 'store'])
    ->middleware('can:contacts.create')->name('contacts.store');
Route::get('contacts/{contact}', [ContactController::class, 'show'])
    ->middleware('can:contacts.view')->name('contacts.show');
// ... edit, update (PUT|PATCH), destroy, et l'unique route AJAX :
Route::post('contacts/{contact}/transition', [ContactController::class, 'transition'])
    ->middleware('can:contacts.edit')->name('contacts.transition');
```

Le contrôleur charge uniquement les données de sa page, avec des scopes Eloquent et la pagination :

```
final class ContactController extends WebController
{
    public function index(Request $request): View
    {
        $stage = $request->string('stage')->toString();

        $contacts = Contact::query()
            ->withFunnelFlags()->with('labels')
            ->search($request->string('q')->toString())
            ->when(in_array($stage, ContactFunnel::STAGES, true),
                fn (Builder $q) => $q->inStage($stage))
            ->orderBy('contacts.created_at', 'desc')
            ->paginate(25)->withQueryString();

        return view('modules.crm.contacts.index', [
            'contacts' => $contacts, 'stages' => ContactFunnel::STAGES,
        ]);
    }
    // show(): $this->authorize('view', $contact) -> ContactPolicy existante
}
```

La vue Blade reçoit directement les modèles ; l'échappement est automatique et les boutons sont filtrés par permission :

```
@foreach ($contacts as $contact)
<tr>
    <td><a href="{{ route('crm.web.contacts.show', $contact) }}">{{ $contact->name }}</a></td>
    <td dir="ltr">{{ $contact->phone }}</td>
    <td><x-ui.badge>{{ __( 'crm:web.stages.' . $contact->funnel_stage ) }}</x-ui.badge></td>
</tr>
@endforeach
@can('contacts.create')
    <x-ui.button variant="primary" :href="route('crm.web.contacts.create')">{{ __( 'crm:web.actions.new' ) }}</x-ui.button>
@endcan
```

Il n'est donc plus nécessaire de créer un endpoint `boot` pour cette page ni de stocker des collections dans `UBCG.DB`. Le formulaire (création et modification) réutilise les règles de validation de l'API (`StoreContactRequest/UpdateContactRequest`) ; un champ caché `version` remplace l'en-tête `If-Match` pour le verrou optimiste.

6. Gestion des actions JavaScript (AJAX)

Le remplacement du chargeur ne signifie pas supprimer JavaScript. Il reste utile pour les interactions sans rechargement : déplacement Kanban, compteurs de l'en-tête, dépôt de fichiers, commentaires. Dans ces cas, Laravel fournit une route dédiée qui renvoie une petite réponse JSON.

6.1 Déplacement Kanban : POST `/ressource/{id}/transition`

Le même moteur que l'API (`TransitionService` : verrou de ligne, arête autorisée, permission, motif, gardes, confirmation, audit) est exposé aux pages par le trait `HandlesWebTransitions`. Requête (JSON, jeton CSRF dans `X-CSRF-TOKEN`) :

```
POST /contacts/0198f3.../transition
Accept: application/json      X-CSRF-TOKEN: <meta csrf-token>
{ "to": "property_buyer", "via": "mouse", "reason": null, "confirm": false }
```

Réponse	Code	Réaction de la page (transition.js)
200	ok = true	Recharge la page : données fraîches, message flash du serveur.
409	transition.confirmation_required	Affiche les effets (traduits) dans une confirmation ; si accepté, renvoie la requête avec confirm = true.
409	transition.guard_failed	Affiche le message traduit du garde (ex. « Le libellé automatique Immobilier est présent... »).
422	transition.reason_required	Demande un motif (longueur minimale fournie par le serveur) puis renvoie la requête.
422 / 403	transition.not_allowed, rbac.forbidden	Affiche le message ; aucune modification.
419	csrf.mismatch	Session expirée : recharger la page.

Le serveur reste le seul juge : l'interface propose les boutons que l'utilisateur peut utiliser (allowedFor), mais ne décide jamais. Chaque déplacement est inscrit dans le journal d'audit chaîné avec son motif.

6.2 Autres usages AJAX prévus

Endpoint	Module	Usage
GET /notifications/unread-count, GET /messages/unread-count	Notifications, Messaging	Compteurs de l'en-tête (data-badge-url, actualisés toutes les 60 s).
POST /{ressource}/{id}/transition	Projects, Hr, Finance, RealEstate...	Statuts et Kanban : même trait HandlesWebTransitions.
POST /{entité}/{id}/comments, /attachments	Audit, Files	Commentaires et dépôt de pièces jointes sans rechargement (inspection, antivirus, chiffrement inchangés).
GET /contacts?q=... (fragment HTML)	Crm et autres listes	Recherche instantanée facultative : la page fonctionne déjà sans JavaScript.
PATCH /me/preferences	Identity	Langue et thème de l'utilisateur.

7. Vues, composants, en-tête et internationalisation

7.1 Layout et composants Blade

Un layout unique (layouts/app.blade.php) et des composants anonymes resources/views/components/ui/* : x-ui.button, card, badge, alert, input, select, textarea, checkbox, page-header, empty-state, plus la pagination pagination.ubcg. Le design est **plat** : aucun dégradé, bordure de 1 px, jetons de couleur repris de la charte du prototype (marine, bleu, gris), mode sombre par prefers-color-scheme. Le harnais vérifie l'absence de dégradé.

7.2 En-tête : notifications et messages

partials/header.blade.php affiche la marque, la cloche, la messagerie, la langue et l'utilisateur. Les compteurs viennent de deux petits endpoints AJAX ; les boutons n'apparaissent que lorsque les routes notifications.web.* et messaging.web.* existent (Route::has), ce qui permet de migrer module par module. Le menu latéral est composé par NavigationRegistry : chaque module déclare ses entrées (navigation() dans son provider) et seules celles autorisées par can: sont affichées.

7.3 Langues fr / de / en / ar et RTL

- **Fichiers Laravel** : lang/<l>/ui.php (socle), app/Modules/<M>/Lang/<l>/web.php (pages) et errors.php (codes d'erreur), appelés par __('ui.common.search') ou __('crm:web.title.index').

- **Choix de la langue** : SetLocale (préférence de l'utilisateur, puis Accept-Language, puis langue du tenant, puis français).
- **RTL** : le layout écrit `<html lang dir> ; dir="rtl"` pour ar. La feuille de style n'emploie que des propriétés logiques (`margin-inline-start`, `inset-inline-end...`), un même CSS sert donc LTR et RTL. Les numéros de téléphone et e-mails sont isolés en `dir="ltr"`.
- **Contrôle** : le harnais vérifie que chaque clé `__()` des vues existe dans les quatre langues, avec les mêmes `:placeholders`.
- **Reprise du prototype** : les dictionnaires `assets/i18n/` sont convertis par script en fichiers `lang/` ; le moteur `i18n` JavaScript disparaît des écrans migrés.

8. Ce qui est conservé : autorisation, tenancy, audit, fichiers, ports

Élément	Rôle dans l'architecture Blade
Policies + permissions.php	Une seule source de vérité : <code>can:<module.action></code> sur la route, <code>\$this->authorize('update', \$modèle)</code> dans le contrôleur, <code>@can(...)</code> dans la vue. <code>auth.js</code> côté client n'est plus nécessaire.
Tenancy (TenantModel, RLS)	Inchangée. Un identifiant d'un autre tenant donne 404 dans les pages comme dans l'API ; le tenant vient de l'utilisateur authentifié.
Audit chaîné	Chaque création, modification, suppression et transition continue d'écrire une entrée de journal (motif et canal via compris).
Fichiers chiffrés (Files)	Inspection, antivirus, chiffrement enveloppe et URL signées courtes inchangés ; seul le formulaire de dépôt passe par Blade/AJAX.
Ports et Contracts	Les modules continuent de communiquer par ports (<code>ClientLookup</code> , <code>ContactTaggerPort</code> , <code>ExpenseSyncPort...</code>) ; <code>Deprac/check 46</code> protègent les frontières.
Argent, numérotation, CSV, PDF, files	<code>Money</code> , <code>DocumentNumberService</code> , <code>CsvWriter</code> , <code>PdfRenderer</code> , <code>jobs Horizon</code> : inchangés.
API JSON <code>/api/v1</code>	Conservée en option pour l'application mobile, les portails, les webhooks et les intégrations (<code>OpenAPI 3.1</code>).

9. Que deviennent les services, la StateMachine, les ports et le frontend actuel ?

Il n'est pas nécessaire de supprimer toutes les classes Service. La recommandation est de supprimer la couche de transport JSON globale (boot, adaptateur, cache) et de ne conserver que les services qui portent une vraie logique réutilisable : limites et quotas, paiement, intégrations externes, génération de documents, import CSV, détection de doublons, calculs (TVA, paie, Zakat). Une simple requête Eloquent qui alimente une page reste dans le contrôleur lorsqu'elle n'est pas réutilisée.

Élément	Décision	Raison
StateMachine, TransitionService, Domain/	Conservés	Logique métier partagée par l'API et les pages ; testée en pur. Les pages l'appellent via <code>HandlesWebTransitions</code> .
Ports / Contracts / Services métier	Conservés	Frontières entre modules, intégrations, import, doublons, paiement, PDF.
ApiController, Resources, OpenAPI	Conservés pour <code>/api/v1</code>	Mobile, portails, intégrations ; les pages du personnel ne les utilisent plus.
FormRequest	Partagées	Les formulaires Blade réutilisent les règles de l'API ; une sous-classe <code>Web</code> ne fait que normaliser les champs HTML.

Élément	Décision	Raison
Services « de page » (simple lecture pour un écran)	Retirés	La requête vit dans le contrôleur ou dans un scope du modèle.
assets/boot.js, data-api.js, api-client.js, api-rules.js, auth-api.js, notify-api.js, UBCG.DB	Supprimés de la cible	Remplacés par les contrôleurs et vues Blade ; supprimés module par module, après test.
assets/auth.js, i18n.js (côté client)	Supprimés de la cible	Permissions par @can, textes par lang/.
ubcg-ui/ (prototype statique, 54 pages)	Deviens les gabarits Blade	Le HTML/CSS est repris dans ressources/views ; le prototype est retiré à la fin de la migration.
Inertia / Vue / TypeScript / Vite	Abandonnés (ADR-007 remplacé)	Remplacés par Blade + JavaScript sans dépendance. Livewire reste optionnel, seulement pour un écran très interactif que le développeur le souhaiterait.

10. Avantages attendus

Aspect	Bénéfice
Lisibilité	Chaque contrôleur correspond à une fonctionnalité métier clairement identifiée ; une page = une route, un contrôleur, une vue.
Maintenance	Une modification sur les contacts, les factures ou les congés reste localisée dans un module ; plus de modification simultanée API + client.
Débogage	Le chemin Route → Controller → Model → View est direct ; plus de chargeur, d'adaptateur ni de cache à soupçonner.
Laravel natif	Routes, middleware, FormRequest, Policies, Eloquent, Blade, lang : un développeur Laravel est productif tout de suite.
Moins de dépendances	Pas de Node/Vite/TypeScript/Vue obligatoires ; suppression de UBCG.boot, UBCG.DB, data-api.js (≈ 2,6 Mo de JavaScript du prototype à terme).
Performance	Chaque page ne charge que ses données (pagination, with()), au lieu de collections entières ; rendu serveur immédiat, sans voile de chargement.
Sécurité	Les données sont filtrées côté serveur par Policies et @can ; échappement Blade par défaut (le harnais refuse {!! !!}) ; CSRF natif ; pas de secret ni de matrice de permissions exposés au navigateur.
Internationalisation	Un seul système (lang/) pour fr, de, en, ar ; RTL par dir et propriétés CSS logiques.
Évolutivité	Une nouvelle fonctionnalité s'ajoute dans son contrôleur et sa vue sans toucher à un point central.
Tests	Tests Feature Pest sur de vraies pages (session, CSRF, HTML), en plus des tests d'API existants.
Mobile et intégrations	L'API /api/v1 et son contrat OpenAPI restent intacts.

11. Migration recommandée

Il est préférable de ne pas supprimer l'existant d'un seul coup. L'API et les pages coexistent : un module n'est retiré du prototype qu'après que ses pages Blade sont testées.

Étape	Contenu	Critère de sortie
0. Socle (livré)	routes.web.php auto-chargé, WebController, layout, composants ui, menu par module, pages d'erreur HTML, transition AJAX, tranche Crm / contacts.	Harnais vert (check 220) ; relecture par le développeur.
1. Identity	Connexion web, 2FA, mot de passe oublié, profil et langue, page d'accueil (/login, /logout).	Un utilisateur se connecte et ouvre /contacts ; tests Feature.

Étape	Contenu	Critère de sortie
2. Crm + en-tête	Clients, demandes de rappel, journal de communication et pièces jointes sur la fiche ; Notifications et Messaging (compteurs AJAX).	Le prototype <code>contacts.html/clients.html</code> n'est plus utilisé.
3. Projects + Dashboards	Projets, services, bons de livraison, suivi, tableaux de bord.	Tests de transition et d'isolation par tenant.
4. Hr	Employés, congés, pointage, contrats, certificats, évaluations, recrutement, arrivées/départs.	Parité de la matrice de permissions maintenue.
5. Finance, Accounting, Donations	Devis, factures, paiements, chèques, dépenses, remboursements, grand livre, dons.	Idempotence et verrous « signé = verrouillé » vérifiés par des tests de page.
6. RealEstate, Marketing	Programmes, unités, réservations ; portails courtier, acheteur et partenaire.	Portails sur la pile <code>portal</code> ou <code>web</code> selon la décision.
7. Legal, Providers, Privacy, Billing, Seo, Platform	Modules restants et pages publiques/landing (Blade avec cache HTTP).	Toutes les pages du prototype ont un équivalent Blade.
8. Nettoyage	Suppression de <code>boot.js</code> , <code>data-api.js</code> , <code>api-client.js</code> , <code>auth.js</code> côté client et du dossier <code>ubcg-ui/</code> ; mise à jour de la documentation.	Plus aucune référence à <code>UBCG.boot</code> (contrôle automatique).

12. Architecture cible

Navigateur → Route (web) → Middleware (auth, tenant, identity, locale, plan, can:) → Controller fonctionnel → FormRequest / Policy → Eloquent (Model, relations, scopes) → Blade (layout + composants) → HTML + CSS + JavaScript léger

AJAX seulement si nécessaire : JavaScript (fetch + CSRF) → Route web → Controller → Model / TransitionService → JSON

Mobile / intégrations (optionnel) : Client → /api/v1 → ApiController → Resource JSON (OpenAPI 3.1)

Services conservés : TransitionService + StateMachine, import CSV, détection de doublons, paiement / facturation, PDF, intégrations, calculs (TVA, paie, Zakat)
 Transverse conservé : Tenancy + RLS, Policies + permissions.php, Audit chaîné,
 Files chiffrés, Ports/Contracts, Money, files d'attente

13. Conclusion pour validation du projet

La proposition consiste à rapprocher UBCG des conventions Laravel classiques. L'approche « API d'abord + client JavaScript » n'est pas incorrecte, mais elle impose deux applications, un état global dans le navigateur et une seconde chaîne d'outils, alors que l'interface du personnel peut être rendue directement par Laravel, avec Blade, Eloquent et des contrôleurs fonctionnels.

Recommandation : adopter progressivement une architecture par fonctionnalité : routes web par module, contrôleurs dédiés, Eloquent pour l'accès aux données, Blade pour le rendu, JSON uniquement pour les besoins AJAX et pour l'API mobile/intégrations optionnelle.

Décision proposée : valider une migration progressive vers cette architecture Laravel native, sans réécriture complète, sans modifier l'API existante ni la sécurité (tenancy, permissions, audit), et en abandonnant l'option Inertia + Vue + TypeScript du concept initial.

État de la tranche de référence (honnêteté sur ce qui est vérifié)

- **Livré dans le dépôt** : Crm / contacts (routes, contrôleur Web, deux FormRequest, trois vues, composants, menu, textes fr/de/en/ar, CSS, JavaScript, test Feature) ; documents d'architecture et guide de module mis à jour.
- **Vérifié par le harnais autonome** : syntaxe PHP, chargement des routes web, présence des contrôleurs, vues, composants, équilibre des directives Blade, clés de traduction dans les quatre langues, CSS plat, JavaScript valide, API inchangée.
- **Non vérifié** : Laravel n'a jamais été exécuté dans l'environnement de rédaction ; aucune page n'a été rendue, le test Feature n'a pas tourné, la connexion web (/login) reste à écrire. La première commande à lancer sur un poste complet est `php artisan view:cache`, puis le test `ContactWebTest`.

Points à valider avec le développeur

- Livewire : aucun besoin identifié pour l'instant ; à réévaluer uniquement pour un écran très interactif.
- Emplacement des vues : `resources/views/modules/<slug>` (choix retenu) ou `app/Modules/<M>/Resources/views` (autonomie du module).
- Kanban : boutons de déplacement (livré) ou glisser-déposer (petit module JavaScript à ajouter).
- Tests de navigateur (Pest Browser / Dusk) pour les parcours critiques.