Modularisation de la suite BY+

Architecture cible, contenu des applications et installation d'une application — guide de compréhension et de mise en place

> **En une phrase :** nous transformons notre application unique en une suite de 6 applications (Shop, ERP, CRM, RH, Access, Event) installables et activables séparément pour chaque client, en rangeant le code en **3 couches** et en le découpant en **packages**, progressivement, sans microservices et sans réécriture.

Pourquoi changer

Nous vendons 6 produits, mais le code (57 modules dans `app/v1`) forme un seul bloc où tout dépend de tout :

Constat	Exemple réel dans le code	Conséquence
Des services partagés dépendent des apps	`Contact` importe Invoice, Pipeline, Project, Event ; `Product` importe Shop	Impossible d'installer Contact sans le CRM, l'ERP et Event
Des apps dépendent d'autres apps	Shop ↔ Event, CRM → Project, RH → Project et Expense	Une modification dans une app peut casser une autre
Composants génériques dans un module métier	RH et Access importent `CRM/components/Card.vue` (22 fois)	RH ne peut pas vivre sans CRM côté front
Tout est câblé à la main	100 lignes d'autoload dans `composer.json`, providers en dur	Rien ne peut être désactivé

La bonne nouvelle : chaque module a déjà son ServiceProvider, ses routes et ses migrations, et le front a un dossier par module. La base est saine ; il faut surtout remettre de l'ordre dans les dépendances.

Ce que nous visons

- **Vendre à la carte :** un client RH ne voit et ne charge que RH et les services dont RH a besoin.
- **Faire travailler les apps ensemble :** un client CRM + ERP voit les factures dans la fiche contact, sans que Contact connaisse la facturation.
- **Modifier sans peur :** une modification dans Event ne peut pas casser RH.
- **Des équipes autonomes :** chaque app a ses frontières, ses tests et ses migrations.

Ce que nous ne faisons pas

- Pas de microservices : une seule application, un seul déploiement, découpés en packages.
- Pas de réécriture : on déplace le code et on corrige ses dépendances.
- Pas de « big bang » : module par module, les livraisons continuent.
- On tolère les mauvaises dépendances existantes ; on interdit d'en créer de nouvelles.

1. L'architecture en 3 couches

Chaque morceau de code est rangé dans une couche. **Une couche peut utiliser celles du dessous, jamais celles du dessus, et une application n'utilise jamais directement une autre application.**



Couche	Définition	Qui l'a ?
1. Socle	Ce dont tout le monde a besoin : tenancy, utilisateurs, droits, sociétés, paramètres, notifications, fichiers, registre des apps.	Tous les clients
2. Services partagés	Briques métier utilisées par plusieurs apps : Contact, Produit, Facturation, Comptes, Personnel, NPS, Messagerie, Formulaires.	Activés automatiquement avec l'app qui en a besoin
3. Applications	Ce que le client achète : ses modules propres (ex. RH = absences, paie, contrats…).	Selon l'abonnement du client

Package ou service ?

Un package est un dossier de code autonome (son `composer.json`, son provider, ses routes, migrations et tests) installé dans la même application. **Un service** est une application séparée, avec son serveur et sa base, appelée par HTTP. Image : les packages sont les applis d'un même téléphone ; les services sont plusieurs téléphones qui doivent s'appeler.

	Package (notre choix)	Service séparé
Communication	Appel de fonction PHP, événements Laravel	Appels HTTP, files de messages
Base de données	Celle du tenant, partagée	Une par service, à synchroniser
Opération sur 2 modules	Transaction SQL classique	Transaction distribuée, compensation
Déploiement	Un seul	Un par service
Coût pour l'équipe	Faible, habitudes Laravel	Élevé : réseau, monitoring, versions d'API

User est utilisé par 45 modules et Society par 34 : en services séparés, chacun de ces usages deviendrait un appel réseau. Un package pourra devenir un service plus tard, si un vrai besoin apparaît (charge très différente, équipe séparée, client hébergé à part).

2. Le socle et les services partagés

Le socle : identique pour toutes les applications

Brique	Modules actuels	Rôle
Tenancy	`Central`	Tenants, bases de données, domaines
Utilisateurs et droits	`User` (+ Passport, Laratrust)	Connexion, rôles, permissions
Sociétés	`Society`	Sociétés, agences, établissements
Paramètres	`Setting`, `Normalization`	Paramètres généraux, référentiels
Registre des apps	`Module`	Apps activées par client (module_installs)
Notifications	`Notification`, `PushNotification`	Notifications internes, mail, push
Journal	`Log`	Historique des actions
Fichiers	`FileManager`	Stockage et upload
Circuit de validation	`ValidationCircuit`	Workflows d'approbation
Tableau de bord	`Dashboard`, `Widget`, `Statistic`	Accueil et widgets alimentés par chaque app
Front	@byplus/core, @byplus/ui	axios, auth, i18n, permissions, composants communs

Quels services partagés pour quelle application

Service partagé	Shop	ERP	CRM	RH	Access	Event
Contact	✓	✓	✓	✓	✓	✓
Produit	✓	✓	✓			
Facturation / Devis		✓ simple	✓ complète	?		
Comptes		✓				
Personnel		✓		✓		
NPS	✓	✓	✓			✓
Messagerie		✓				✓
Formulaires			✓			✓
Template Builder						✓

Facturation simple (ERP) et complète (CRM) forment un seul service : l'app active plus ou moins de fonctionnalités. NPS, Messagerie et Formulaires apparaissent dans plusieurs apps, ils deviennent donc des services partagés.

3. Le contenu de chaque application

Chaque application = le socle + ses services partagés + ses modules propres. Les chiffres indiquent le nombre d'imports trouvés dans le code actuel ; la dernière colonne liste ce qui viole la règle et doit être corrigé.

Application	Fonctionnalités	Modules propres (code)	Services partagés	À corriger
BY+ Shop	Stocks, articles, boutique, entrepôt, caisse, état des ventes, packs	Shop, Synchronisation	Contact, Produit, NPS	Event (12) — et Event importe Shop (6) : dépendance croisée
BY+ ERP	Finance, flotte, matériel, explorateur, projet, budget, rapports, fil d'actualité, dépenses	Expense, Fleet, Material, FolderManager, ArchiveFolder, Project, ProjectBudget, ProjectReporting, Story, OnlyOffice	Contact, Produit, Facturation simple, Comptes, Personnel, NPS, Messagerie	RH : Payslip, Contract, Holiday ; CRM : Appointment, Pipeline (1 chacun)
BY+ CRM	Pipeline, facturation complète, RDV, rétroplanning	Pipeline, Appointment, RetroPlanning, Schedule, CustomerPortal, Ao	Contact, Produit, Facturation complète, NPS, Formulaires	ERP : Story (13), Project (8), FolderManager (2)
BY+ RH	Absences, fiche de paie, contrat, collaborateurs, avances, évaluations	Holiday, Payslip, Contract, Employee, SalaryAdvance, Objective, CVs	Personnel, Contact (+ Facturation ?)	ERP : Project (13), Expense (5)
BY+ Access	Gestion des accès, présence	Access, Presence	Contact	Aucune — app idéale pour commencer
BY+ Event	Portail événementiel, QVCT, ateliers, formulaires, billetterie	Event, Qvct, Karaoke	Contact, Formulaires, NPS, Template Builder, Messagerie	Shop (6), Synchronisation (1)

Décisions à prendre en équipe

- **Shop et Event dépendent l'un de l'autre.** La partie commune (probablement la vente de billets) devrait devenir un service partagé.
- **Story (fil d'actualité) et FolderManager (explorateur)** sont utilisés par CRM et RH : services partagés plutôt que modules ERP ?
- **RH utilise Invoice** (factures prestataires de la paie ?) : RH dépend de Facturation, ou lien remplacé par un événement ?
- **Modules à classer :** PostOffice, Survey, Schedule, CustomerPortal, Ao, CVs, Karaoke, OnlyOffice, WebMail.
- **Apps déjà séparées :** access-api, crmApi et events-management — les réintégrer comme packages ou les garder comme services ?

> Les services partagés doivent eux aussi être nettoyés : ils importent encore Event (23), Expense (14), Pipeline (8) et Shop (6). C'est la priorité de la phase de correction.

4. Comment on découpe concrètement

Découper, c'est deux choses : **déplacer** chaque module dans un package, et **remplacer** les imports interdits par l'un des trois mécanismes ci-dessous.

Organisation des dossiers

```
core-api/
  packages/byplus/
    kernel/            <- Central, User, Society, Setting, Module, Notification...
    contact/  product/  invoicing/  ...
    app-rh/
      composer.json    <- nom, dépendances (byplus/kernel, byplus/contact...), autoload
      module.php       <- manifeste : clé, modules, dépendances
      src/  database/migrations/  routes/  tests/
```

Le `composer.json` racine ne contient plus qu'une ligne par package (dépôt de type `path`) ; le provider est découvert automatiquement.

```
// packages/byplus/app-shop/module.php
return [
    'key'      => 'shop',
    'requires' => ['kernel', 'contact', 'product', 'nps'],
    'modules'  => ['shop', 'stock', 'warehouse', 'pos', 'sales_report', 'pack'],
];
```

Faire communiquer les modules sans les coller

Besoin	Mécanisme	Exemple
Un module du dessous doit afficher des données du dessus	**Branchement** : le module du dessus s'ajoute au module du dessous	Facturation ajoute la relation invoices() et l'onglet « Factures » à Contact
Prévenir d'autres modules qu'une chose s'est produite	**Événement** : on annonce, ceux qui sont installés réagissent	Product émet ProductUpdated, Shop l'écoute pour son stock
Lire des données d'un service du dessous	**Contrat** (interface) : on dépend d'une promesse, pas d'une classe	RH lit un contact via ContactDirectory

```
// AVANT — Contact/Models/Contact.php (Contact dépend de Facturation)
use Core\v1\Invoice\Models\Invoice;
public function invoices(): HasMany { return $this->hasMany(Invoice::class); }

// APRÈS — dans le provider du package Facturation (Contact ne sait plus rien)
Contact::resolveRelationUsing('invoices', fn (Contact $c) => $c->hasMany(Invoice::class));

// ÉVÉNEMENT — Product annonce, Shop écoute s'il est installé
event(new ProductUpdated($product));
Event::listen(ProductUpdated::class, SyncShopStock::class);
```

Côté front : `@byplus/ui` (composants communs, dont Card.vue), `@byplus/core` (axios, auth, i18n, permissions), et un package par app qui exporte `registerApp({ routes, menu, store })`. Le shell ne charge que les apps activées pour le client.

5. L'installation d'une application

À retenir : en SaaS, le code de toutes les apps est déployé une seule fois et toutes les tables sont migrées pour tous les clients. « Installer » une application pour un client = **l'activer**. Aucune migration ne tourne à ce moment-là.



Aucune migration à cette étape : les tables existent déjà pour tous les clients.

Les étapes en détail (exemple : Shop pour le client sdp)

1. **Déclenchement :** un administrateur clique sur « Activer Shop », ou on lance `php artisan byplus:install shop --tenant=sdp`.
2. **Vérification des dépendances :** le socle lit le manifeste de Shop (requires : kernel, contact, product, nps) et regarde ce que le client possède déjà.
3. **Activation des services requis :** si Contact, Produit ou NPS ne sont pas actifs, ils sont activés d'abord.
4. **Seeders d'installation :** création des permissions (shop_view, stock_edit…), des rôles par défaut et des paramètres initiaux.
5. **Enregistrement :** ajout des lignes dans `module_installs` pour ce client.
6. **Ouverture :** le middleware `app.enabled:shop` laisse passer les appels `/api/shop/...` ; au prochain chargement, le front charge le menu, les routes et les écrans de Shop.
7. **Branchements actifs :** les écouteurs de Shop se mettent en marche (ex. mise à jour du stock quand un produit change).

Ce que le client ne voit pas

Les menus CRM, RH et Event n'apparaissent pas et leurs routes restent bloquées. La fiche Contact n'a pas d'onglet « Factures », car Facturation n'est pas installée. S'il prend le CRM plus tard, même procédure : seuls Facturation et CRM s'ajoutent.

Les migrations : pourquoi tout migrer pour tous

C'est déjà le fonctionnement actuel : `MigrateDatabase` crée toutes les tables à la création d'un client, et `tenants:migrate` lance les migrations de tous les modules pour tous les clients. On le garde.

	Tout migrer pour tous (choix retenu)	Migrer seulement les apps installées
Schéma des bases	Identique pour tous les clients	Différent d'un client à l'autre
Mise à jour	Une commande pour tous	Client par client, selon ses apps
Installer une app plus tard	Instantané, les tables existent	Rejouer tout l'historique de l'app
Clés étrangères entre modules	Aucun risque	Cassent si la table cible n'existe pas
Inconvénient	Tables vides chez certains clients	Beaucoup plus de complexité

Exception : un client hébergé à part n'installe que certains packages (`composer require byplus/app-shop`, qui amène automatiquement product, contact et kernel). Seules leurs migrations existent. D'où la règle : une migration ne crée jamais de clé étrangère vers la table d'une app dont elle ne dépend pas.

6. L'installation en détail : spécification technique

Cette section décrit précisément ce que fait le système quand on installe une application pour un client, ce que cela implique dans le code, et ce qu'il faut changer par rapport à l'existant.

Ce qui existe déjà aujourd'hui

Élément	État actuel dans le code	Problème
Registre central	Tables `suit_apps`, `modules`, `module_installs` (tenant_id, module_id, enabled, domain) dans la base centrale	Pas de version, pas de statut, pas de trace de qui a installé
Endpoint d'installation	`POST install/modules` (ModuleController::installModules)	Si tout est déjà installé, le même appel **supprime** les lignes : installer et désinstaller sont le même bouton
Dépendances	Aucune	On peut activer Shop sans Produit
Seeders par module	`SuitModulesEnum::getSeeders()` liste des seeders (Project, Qvct, Event...)	**Jamais appelé** : les données de référence ne sont pas créées à l'installation
Contrôle côté API	Middleware `module.access` déclaré dans bootstrap/app.php	**Utilisé sur 0 route** : l'API d'une app non installée reste accessible
Contrôle côté front	Le routeur appelle `check/module/{name}` ; les routes ont `requiresAccess`	Seule protection réelle, facile à contourner
Permissions	Laratrust, créées à la volée par `ManagePermissionTrait::createPermission`	Pas créées à l'installation, rôles par défaut non configurés
Migrations	Toutes les tables pour tous les clients	OK, on garde

Bonne nouvelle : la granularité existe déjà. Une application est un groupe de modules (61 dans SuitModulesEnum), et les niveaux de facturation existent sous forme de modules distincts (GENERAL_INVOICE pour la facturation simple, CRM_INVOICE pour la complète).

Le manifeste complet d'une application

Tout ce que l'installateur doit savoir est déclaré par le package lui-même, dans son manifeste :

```
// packages/byplus/app-shop/module.php
return [
    'key'         => 'shop',
    'version'     => '1.4.0',
    'requires'    => ['kernel', 'contact', 'product', 'nps'],   // dépendances
    'modules'     => ['STOCK_ARTICLE', 'SHOP', 'WAREHOUSE', 'POS', 'SALES_REPORT', 'PACK'],
    'permissions' => ShopPermissionEnum::class,                 // créées à l'installation
    'roles'       => ['admin' => '*', 'caissier' => ['pos_use', 'shop_view']],
    'seeders'     => [ShopDefaultSettingsSeeder::class, DefaultWarehouseSeeder::class],
    'install'     => ShopInstaller::class,  // hook optionnel pour les cas particuliers
];
```

Les étapes de l'installateur

Un service unique du socle, `AppInstaller`, exécute ces étapes dans l'ordre. Il est appelé par l'écran d'administration, par la commande `php artisan byplus:app:install shop --tenant=sdp`, et par la création d'un client (apps par défaut).

Étape	Où	Ce qui se passe techniquement	En cas d'échec
0. Autoriser	API	Seul un super-admin (ou l'abonnement du client) peut installer. Vérification de la licence si elle existe.	Refus 403, rien n'est fait
1. Résoudre les dépendances	Socle	Lire les manifestes, construire le graphe requires, tri topologique → plan ordonné : contact, product, nps, shop. Détection des cycles. Option --dry-run pour afficher le plan sans rien faire.	Cycle ou app inconnue : refus, rien n'est fait
2. Poser un verrou	Cache / Redis	`Cache::lock("app-install:sdp")` : deux installations simultanées pour le même client sont impossibles. Statut installing dans le registre.	Verrou pris : « installation déjà en cours »
3. Vérifier le schéma	Base tenant	Contrôler qu'aucune migration n'est en attente pour ce client (le déploiement les a déjà passées). Aucune migration propre à l'app.	Migrations en attente : lancer tenants:migrate pour ce client, puis reprendre
4. Préparer les données	Base tenant	Pour chaque package du plan, dans l'ordre, dans tenancy()->run() : créer les permissions de l'enum, les attacher aux rôles déclarés, lancer les seeders (données de référence, paramètres par défaut), puis le hook install() (ex. entrepôt par défaut pour Shop, étapes de pipeline pour CRM, numérotation des factures).	Statut failed + journal ; tout est idempotent, on relance sans risque
5. Enregistrer	Base centrale	Lignes `module_installs` : enabled = true, version, installed_at, installed_by, status = active. Fait **en dernier** : une app n'est jamais marquée active sans ses données.	Rien d'actif ; relance possible
6. Vider les caches	Cache + front	Invalider le cache des apps du client, incrémenter apps_version, diffuser l'événement temps réel (laravel-echo-server) pour que les navigateurs ouverts rechargent le menu.	Sans gravité : le cache expire seul
7. Annoncer	Événements	Émettre `AppInstalled(tenant, "shop")` : journal d'audit, notification aux admins, réactions des autres apps.	Sans gravité
8. Tâches longues	Queue (Horizon)	Imports initiaux, synchronisations, indexations : lancés en job, jamais pendant la requête.	Rejouables depuis Horizon
9. Libérer et contrôler	Socle	Libérer le verrou. Contrôle final : check/module renvoie true pour chaque module installé.	Alerte si incohérence

Pourquoi la base centrale en dernier ? Le registre (base centrale) et les données (base du client) sont dans deux bases différentes : une seule transaction SQL ne peut pas couvrir les deux. On prépare donc tout côté client avec des opérations rejouables (updateOrCreate), et on n'active l'app qu'une fois tout prêt. En cas d'échec, on relance simplement la commande.

Esquisse de l'installateur

```
// Socle — esquisse de l'installateur
public function install(Tenant $tenant, string $app, ?User $by = null): void
{
    $plan = $this->registry->resolve($app);                  // [contact, product, nps, shop]
    Cache::lock("app-install:{$tenant->id}", 600)->block(5, function () use ($tenant, $plan, $by) {
        $this->assertNoPendingMigrations($tenant);
        foreach ($plan->missingFor($tenant) as $package) {
            $tenant->run(fn () => $package->install());       // permissions, rôles, seeders, hook
            $this->registry->markActive($tenant, $package, $by); // base centrale, en dernier
        }
        $this->registry->flushCache($tenant);
        event(new AppInstalled($tenant, $plan->target()));
    });
}
```

Ce que l'installation implique dans tout le code

Point essentiel en SaaS : **le code de toutes les apps tourne pour tous les clients**. Chaque point d'entrée doit donc vérifier que le client possède l'app. Il y en a six :

Point d'entrée	Risque si on ne vérifie pas	Règle à appliquer
Routes API	Un client sans Shop appelle /api/shop/... et obtient des données	Chaque package enveloppe ses routes dans `app.enabled:shop` (évolution de module.access), avec cache par client
Écouteurs d'événements	ProductUpdated déclenche la synchro stock de Shop chez un client sans Shop	Les écouteurs d'une app héritent d'une classe de base qui ignore l'événement si l'app n'est pas active
Tâches planifiées (schedule)	Une commande nocturne de Shop tourne pour tous les clients	Boucler uniquement sur `Tenant::withApp("shop")`
Jobs en file	Job lancé puis app désactivée entre-temps	Le job revérifie l'app au démarrage
Relations branchées	invoices() existe sur Contact même sans Facturation	Sans danger (tables vides), mais l'UI et les API n'exposent l'onglet que si l'app est active
Front	Menus et écrans visibles	Liste des apps actives récupérée à la connexion ; seuls leurs packages sont chargés ; garde du routeur conservée

Autres impacts : l'écran des rôles n'affiche que les permissions des apps installées (déjà fait par getInstalledModulePermissions) ; le tableau de bord n'affiche que les widgets des apps actives ; la recherche globale et les exports filtrent par app.

Désactiver, désinstaller, mettre à jour

Action	Ce qui se passe	Réversible ?
Désactiver	enabled = false : routes bloquées, menu caché, écouteurs et tâches arrêtés. **Les données restent.** Refusé si une app active en dépend (on ne désactive pas Produit tant que Shop est actif).	Oui, réactivation instantanée
Supprimer les données	Action séparée et explicite : export d'abord, puis commande de purge des tables de l'app pour ce client. Jamais automatique.	Non
Services partagés orphelins	Si plus aucune app n'utilise Produit, il reste actif (ses données peuvent servir). Désactivation manuelle possible.	—
Mettre à jour une app	Le déploiement lance tenants:migrate pour tous. Les données propres à l'app (nouvelle permission, nouveau paramètre) passent par des « étapes d'installation » versionnées, jouées uniquement pour les clients qui ont l'app, et tracées comme des migrations.	Selon l'étape

Évolution du registre central

```
module_installs : + version, + status (installing | active | failed | disabled),
                  + installed_at, + installed_by, + disabled_at
app_install_logs : tenant_id, app, action (install | disable | upgrade),
                   step, status, message, created_at
app_install_steps (base tenant) : app, step, ran_at   ← étapes versionnées déjà jouées
```

Travaux à réaliser pour l'installation

- Séparer l'endpoint actuel en deux : `POST apps/{app}/install` et `POST apps/{app}/disable` (fin du bouton bascule).
- Créer AppRegistry (lecture des manifestes, résolution des dépendances) et AppInstaller dans le socle.
- Brancher les seeders déjà listés dans `SuitModulesEnum::getSeeders()` ; les rendre idempotents.
- Appliquer le middleware sur les groupes de routes de chaque module, en commençant par Access et Presence.
- Ajouter la classe de base des écouteurs et le filtre Tenant::withApp() pour les tâches planifiées.
- Ajouter les colonnes du registre et la table de journal ; commande byplus:app:install avec --dry-run.
- Front : invalidation du cache des modules et rechargement du menu à la réception de AppInstalled.
- Tests automatiques par app : client neuf → installer → permissions créées, routes de l'app en 200, routes des autres apps en 403.