BY+ · ARCHITECTURE DE RÉFÉRENCE

L'architecture de BY+, expliquée simplement

Un seul logiciel, des modules que chaque client choisit, et des règles simples que tout le monde peut comprendre. C'est la base du redémarrage de BY+.

À lire par toute l'équipe : développeurs, produit, commercial, direction.
Les pages 1 à 3 et le chapitre 9 (personnalisation) s'adressent à tout le monde ; les autres détaillent la technique.
Septembre 2026.

1 BY+ en une page

> **En une phrase.** BY+ est **un seul logiciel**, écrit **une seule fois**, utilisé par tous les clients. Chaque client choisit une **offre**, c'est-à-dire un ensemble de **modules**, et ne voit, ne charge et ne paie que ceux-là.



Figure 1 : un seul BY+, quatre clients, quatre combinaisons de modules. Aucun client n'a « sa » version du code.

Ce que ça change pour chacun

Pour…	Concrètement
Le client	Il ne voit que ce qu'il a pris : menus, écrans et droits. L'application est plus légère et plus rapide. Il peut ajouter un module à tout moment, sans migration ni attente.
Le commercial	Il vend des offres : des listes de modules. Créer une nouvelle offre ne demande **aucun développement**.
Le produit	Chaque fonctionnalité est un module avec un nom métier, un périmètre clair et un responsable.
Le développeur	Il travaille dans **un dossier** (son module) avec des règles claires. Une modification dans un module ne peut pas casser un autre module.
L'exploitation	Un seul déploiement pour tous les clients. Un correctif est livré une fois, pour tout le monde.

Les 4 mots à connaître

Socle
Ce que tout client a toujours : comptes, droits, sociétés, contacts, fichiers, notifications.

Module
Une fonctionnalité qu'un client peut utiliser seule : Caisse, Paie, Contrôle d'accès…

Pont
Ce qui relie deux modules quand un client a les deux. Ex. : une vente en caisse fait baisser le stock.

Offre
Une liste de modules, vendue sous un nom. Ex. : « Commerce », « RH Essentiel ».

2 L'image à retenir : un smartphone

Pour expliquer BY+ à quelqu'un qui n'est pas développeur, on peut le comparer à un smartphone.

Smartphone	BY+	Pourquoi c'est pareil
Le système du téléphone (écran d'accueil, réglages, comptes, contacts)	**Le socle**	Toujours là, identique pour tous, et chaque application s'en sert.
Une application (Agenda, Photos…)	**Un module**	Elle fait une chose, fonctionne seule, et on peut l'installer ou la retirer sans casser le téléphone.
Le bouton « Partager vers… » entre deux applis	**Un pont**	Il n'apparaît que si les deux applications sont présentes. Aucune ne dépend de l'autre.
Un pack d'applications préinstallées	**Une offre**	C'est un choix commercial, pas un autre téléphone.
Fond d'écran, sonneries, widgets choisis	**La personnalisation**	Chacun adapte son téléphone sans que le fabricant réécrive le système (chapitre 9).

> Tous les téléphones d'un même modèle ont le même système. Ce qui les distingue, ce sont les applications installées et les réglages. **Il en va de même pour les clients BY+** : même code, modules et réglages différents.

Comment savoir si quelque chose est un module ?

On répond à 4 questions. Si la réponse est « oui » aux quatre, c'est un module.

#	Question	Si « non »
1	Un client peut-il s'en servir **sans autre module** ?	C'est une partie d'un module, ou un pont.
2	A-t-il **son entrée de menu** et ses écrans ?	C'est un morceau du socle ou d'un autre module.
3	A-t-il **ses propres droits** (voir, créer, valider…) ?	Trop petit : on le regroupe avec un module voisin.
4	Peut-on le **retirer** sans casser un autre module ?	Tout le monde en a besoin : il va dans le socle.

3 La carte : socle, modules, ponts, offres

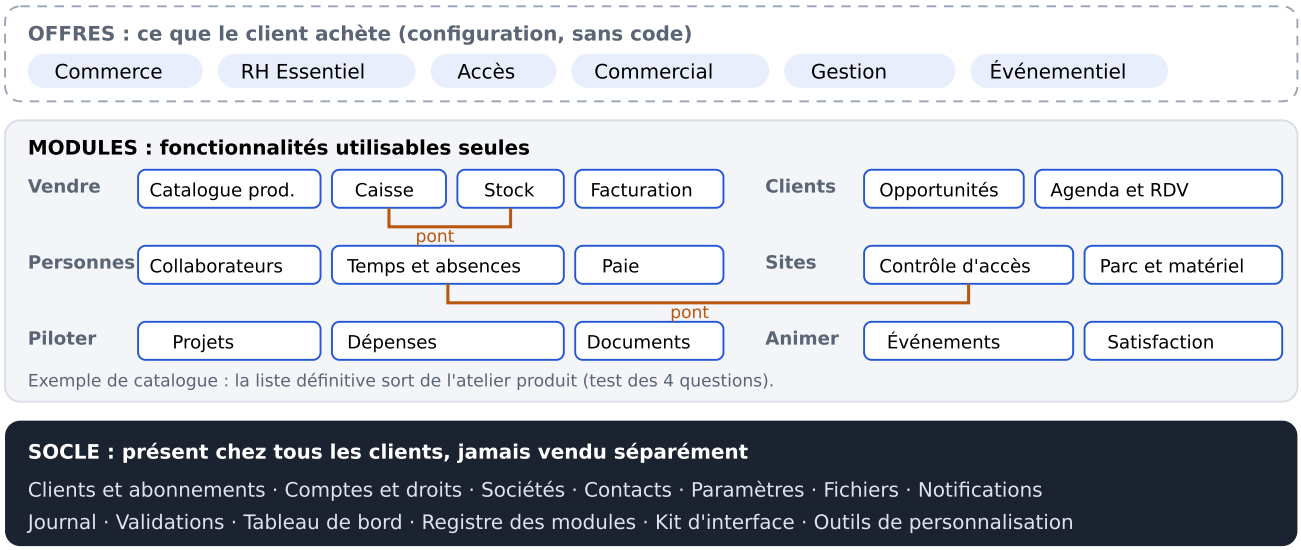


Figure 2 : chaque étage n'utilise que l'étage du dessous. Les modules ne se connaissent pas : les ponts (en orange) les relient.

Les liens autorisés, en une table

Qui	Peut utiliser	Ne peut jamais utiliser
Socle	Rien d'autre que lui-même	Un module, un pont ou une extension
Module	Le socle, et la **vitrine** d'un module qu'il déclare requérir (ex. Paie → Collaborateurs)	L'intérieur d'un autre module ; une extension
Pont	Le socle et la vitrine de ses deux modules	L'intérieur de ses modules
Extension client	Le socle et la vitrine du module qu'elle complète (chapitre 9)	L'intérieur du module
Offre	Une liste de modules	Du code

La **vitrine** d'un module est son dossier `Public/` : les quelques classes qu'il accepte de montrer (interfaces, événements, points d'extension, objets de données). Tout le reste est privé.

4 Vue d'ensemble technique

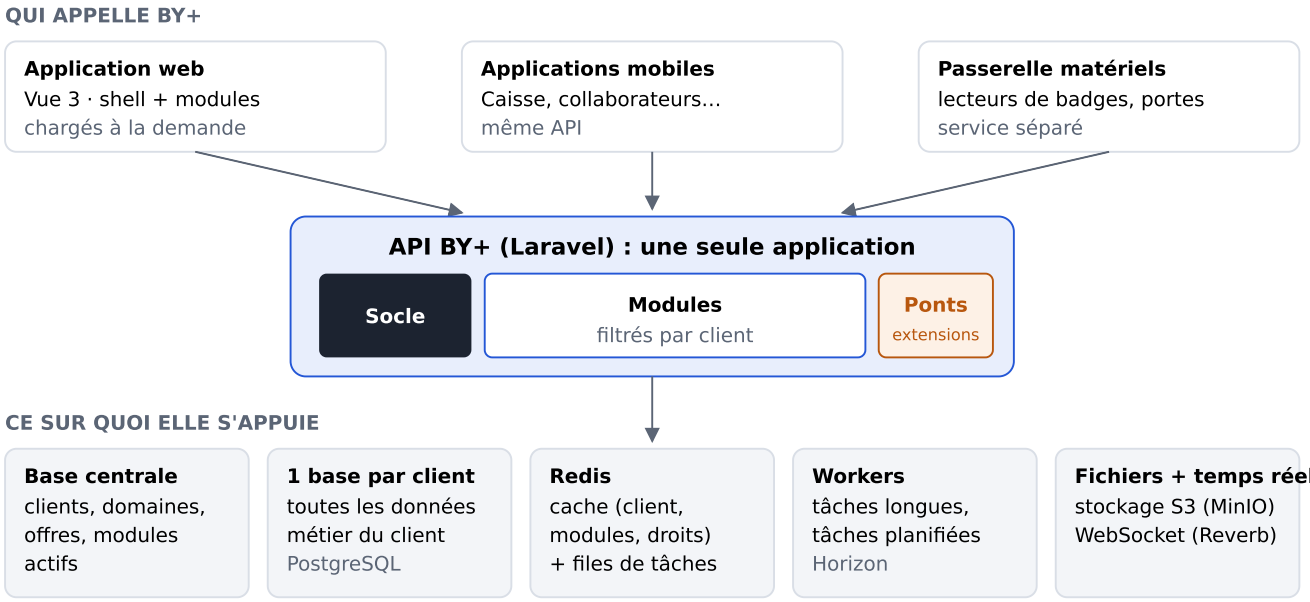


Figure 3 : une API, une base centrale, une base par client, Redis et des workers. Rien de plus.

Les choix techniques

Sujet	Choix	Pourquoi
Forme générale	**Monolithe modulaire** : une application découpée en modules	Simple à développer, déployer et déboguer. Pas de réseau entre modules, et une transaction SQL suffit quand deux modules agissent ensemble.
Back-end	Laravel, PHP 8.4+	Déjà maîtrisé par l'équipe ; écosystème complet (files, cache, temps réel, tests).
Clients	Une **base de données par client** (stancl/tenancy), identifiée par sous-domaine	Isolation forte, simple à expliquer (« chaque client a sa base »), sauvegarde ou restauration d'un client seul.
Base de données	PostgreSQL	Robuste, déjà en place.
Cache et files	Redis + Horizon	Ce qui est lu souvent ne touche pas la base ; les tâches longues sortent des requêtes.
Front-end	Vue 3 + Vite + Pinia + TypeScript	Continuité avec l'existant. Pinia permet d'enregistrer un store à la demande. TypeScript sécurise le code sur la durée.
Interface	Un **kit d'interface** interne (dans le socle), basé sur une seule bibliothèque de composants	Une seule manière de faire un tableau, un formulaire ou une carte. Aucun script global de 5 Mo.
Fichiers, temps réel	S3 (MinIO) · Laravel Reverb	Standards, hébergeables par nous.
Qualité	Pest, Vitest, Playwright · Pint, Larastan, ESLint · **Deptrac**	Deptrac vérifie automatiquement les liens autorisés entre socle, modules, ponts et extensions.

Ce qu'on ne fait pas :

- pas de microservices ;
- pas de code spécifique à un client dans un module ;
- pas de branche Git par produit ni par client ;
- pas de schéma de base différent selon les clients.

Chacun de ces choix complique le système pendant des années pour un gain que nous n'avons pas.

5 Organisation du code

Un seul dépôt pour le back et le front. **Un module porte le même nom côté API et côté web.** Quand on cherche « la Caisse », on sait où regarder.

```
byplus/
├── api/                          ← Laravel
│   ├── core/                     ← LE SOCLE
│   │   ├── Tenancy/  Auth/  Companies/  Contacts/  Files/  Notifications/ ...
│   │   ├── Customization/        ← champs personnalisés, modèles de documents...
│   │   └── Modules/              ← registre : lit les manifestes, sait qui a quoi
│   ├── modules/
│   │   ├── caisse/               ← UN MODULE (voir page suivante)
│   │   ├── stock/
│   │   ├── temps-absences/
│   │   └── ...
│   ├── bridges/
│   │   └── stock-caisse/         ← UN PONT
│   ├── extensions/
│   │   └── sdp-primes/           ← UNE EXTENSION CLIENT (chapitre 9)
│   └── config/offers.php         ← LES OFFRES
│
├── web/                          ← Vue 3
│   └── src/
│       ├── core/                 ← shell, connexion, kit d'interface, i18n, droits
│       ├── modules/
│       │   ├── caisse/           ← même nom que côté API
│       │   └── ...
│       ├── bridges/
│       └── extensions/
│
└── docs/                         ← architecture, catalogue des modules, décisions
```

Anatomie d'un module

```
api/modules/caisse/
├── module.php          ← carte d'identité
├── Public/             ← VITRINE (seule partie
│   ├── Events/             visible des autres)
│   ├── Contracts/
│   └── Hooks/          ← points d'extension
├── Actions/            ← 1 classe = 1 action
│                          métier (Encaisser…)
├── Http/               ← contrôleurs fins
├── Models/
├── routes.php
├── database/
│   ├── migrations/     ← tables caisse_*
│   └── seeders/        ← données de départ
├── lang/
└── tests/
```

```
web/src/modules/caisse/
├── module.ts           ← carte d'identité
├── routes.ts
├── menu.ts
├── store.ts            ← Pinia
├── views/
├── components/         ← privés au module
└── lang/
    ├── fr.json
    └── en.json

Tout composant utile à plusieurs
modules va dans le kit d'interface
du socle (web/src/core/ui).
```

La carte d'identité (manifeste)

```php
// api/modules/caisse/module.php
return [
    'key'         => 'caisse',
    'name'        => 'Caisse',
    'version'     => '1.0.0',
    'requires'    => ['catalogue-produits'],     // modules indispensables (rare, 2 niveaux max)
    'permissions' => ['caisse.vendre', 'caisse.cloturer', 'caisse.parametrer'],
    'roles'       => ['caissier' => ['caisse.vendre']],
    'settings'    => ['arrondi' => null, 'ticket_logo' => true],   // réglages par client
    'seeders'     => [MoyensPaiementParDefaut::class],
    'schedule'    => [ClotturerSessionsOubliees::class => 'daily'],
];

// api/config/offers.php
'commerce'     => ['catalogue-produits', 'caisse', 'stock'],
'rh-essentiel' => ['collaborateurs', 'temps-absences'],
'acces'        => ['controle-acces', 'temps-absences'],
```

6 Comment les modules travaillent ensemble

Il n'existe que **trois** façons de faire coopérer des modules. Toute autre façon est interdite et bloquée par Deptrac.

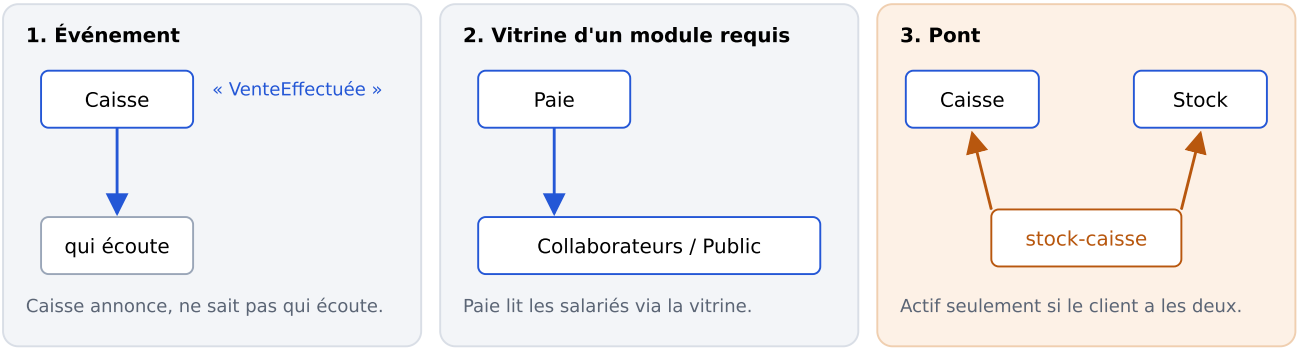


Figure 4 : les trois seules façons de coopérer.

Mécanisme	Quand l'utiliser	Exemple
Événement	Prévenir que quelque chose s'est passé, sans savoir qui est intéressé	`VenteEffectuee`, `AbsenceValidee`, `ContactFusionne`
Vitrine	Lire des données d'un module qu'on **requiert** (déclaré dans `requires`)	Paie appelle `Collaborateurs\Public\Contracts\Annuaire`
Pont	Faire travailler ensemble deux modules **facultatifs** l'un pour l'autre	`stock-caisse` : chaque vente crée une sortie de stock

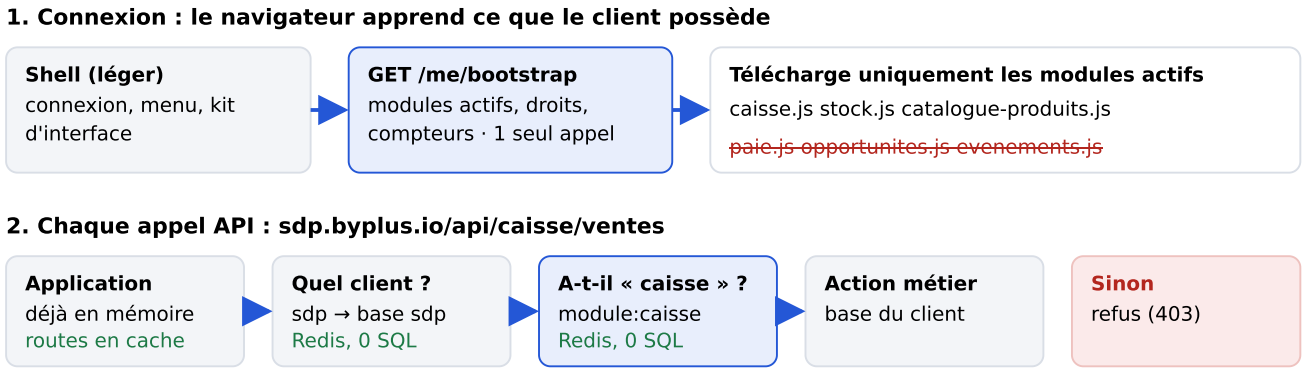
Et à l'écran ? Des emplacements

Le socle définit des **emplacements nommés** dans l'interface : onglets de la fiche contact, widgets du tableau de bord, actions d'une ligne. Un module, un pont ou une extension y dépose ses composants. La fiche contact ne connaît pas la facturation : c'est le pont `facturation-contacts` qui ajoute l'onglet « Factures », uniquement chez les clients qui ont Facturation.

```
// web/src/bridges/facturation-contacts/bridge.ts
export default {
  when: ['facturation'],                      // Contacts est dans le socle
  slots: { 'contact.onglets': () => import('./OngletFactures.vue') },
}
```

7 Chaque client ne charge que ses modules

> **Le principe.** Sur le **serveur**, le code de tous les modules est chargé une seule fois et reste en mémoire. À chaque requête, on vérifie en une lecture Redis que le client a le module. Dans le **navigateur**, on ne télécharge réellement que les modules du client : c'est là que se voit le gain.



Module activé ou retiré → version changée dans Redis → message temps réel → le navigateur recharge ses modules.

Figure 5 : le navigateur charge seulement ce qu'il faut ; le serveur vérifie sans jamais interroger la base pour savoir qui a quoi.

La même vérification partout

Le code de tous les modules tourne pour tous les clients. **Chaque porte d'entrée vérifie donc que le client a le module.** C'est automatique dès que le module suit le gabarit.

Porte d'entrée	Vérification (fournie par le socle)
Routes API	Le socle enveloppe automatiquement les routes du module dans le middleware `module:caisse`
Tâches planifiées	Le socle ne lance la tâche `schedule` du manifeste que pour les clients qui ont le module
Écouteurs d'événements	Classe de base `ModuleListener` : ignore l'événement si le module n'est pas actif
Tâches en file	Revérifient le module au démarrage (le client a pu le retirer entre-temps)
Écrans	Le navigateur n'a même pas le code des modules absents

La performance dès la conception

Règle	Effet
Config, routes et événements en cache ; opcache ; aucun état global (application compatible Octane)	L'application démarre presque instantanément ; Octane pourra être activé sans réécriture
Client, modules actifs et droits en Redis	Zéro requête SQL pour savoir « qui a quoi »

Un seul appel de démarrage (`/me/bootstrap`)	Pas de cascade d'appels à l'ouverture
Un fichier JS par module ; bibliothèques lourdes (graphiques, calendrier, éditeur) chargées dans l'écran qui les utilise	Le premier chargement reste léger, quel que soit le nombre de modules de BY+
Tâches longues toujours en file (exports, imports, envois)	Les requêtes restent rapides
Pagination obligatoire, index sur les colonnes filtrées, chargement paresseux interdit (`preventLazyLoading`)	Pas de requêtes N+1 cachées
Budget de taille du shell et suivi du temps de réponse en CI	Toute régression est vue avant la mise en production

8 Les données

Où	Contient	Ne contient jamais
Base centrale	Clients, domaines, offres souscrites, modules et extensions actifs, historique des activations	De donnée métier (ventes, salariés…)
Base d'un client	Tables du socle et de **tous** les modules, qu'il les ait ou non ; ses réglages et personnalisations	De donnée d'un autre client

Les règles des tables

1. **Chaque table appartient à un seul module** et porte son préfixe : `caisse_sessions`, `stock_mouvements`. On sait toujours qui en est responsable.
2. **Toutes les tables existent chez tous les clients.** Toutes les bases ont donc le même schéma, et une mise à jour se fait en une seule commande pour tous. Activer un module est instantané : ses tables sont déjà là, vides.
3. **Une clé étrangère ne pointe que vers le socle ou vers un module requis.** Jamais vers un module facultatif : ce lien-là passe par un pont.
4. **Un module ne lit ni n'écrit jamais dans les tables d'un autre module.** Il passe par la vitrine.
5. **Les données restent quand on retire un module.** Leur suppression est une action séparée, explicite, précédée d'un export.

Activer un module chez un client

Activer une **offre** revient à activer chacun de ses modules. Les ponts s'activent tout seuls quand leurs deux modules sont présents.



Figure 6 : aucune migration à l'activation. Chaque étape peut être relancée sans risque en cas d'erreur.

9 Personnaliser pour un client

> **La règle de base.** On ne modifie **jamais** le code d'un module pour un seul client : pas de `if ($client === 'sdp')`, pas de copie du module, pas de branche. Une demande de personnalisation passe par une **échelle de 5 niveaux**, et on s'arrête au premier qui suffit.



Figure 7 : l'échelle de personnalisation. Plus on monte, plus c'est coûteux à maintenir.

Niveau	Ce qu'on fait	Qui	Code ?
1. Réglage	Le module a déjà l'option (déclarée dans `settings`) : on l'active pour ce client	Support ou client	Non
2. Configuration	Outils du socle, disponibles dans tous les modules : champs personnalisés, listes de valeurs, circuits de validation, modèles de documents (PDF, mails), rôles et droits, libellés	Client ou intégrateur	Non
3. Option pour tous	Le besoin peut servir à d'autres : on l'ajoute au module comme nouveau réglage, désactivé par défaut	Équipe du module	Oui, dans le module
4. Extension client	Besoin vraiment propre au client : un petit module à part, qui se branche sur le module sans le modifier	Équipe intégration	Oui, hors du module
5. Intégration	Le besoin relève d'un autre logiciel du client : on le connecte par l'API ou des webhooks	Client ou intégrateur	Non, dans BY+

> **La question qui tranche entre 3 et 4 : « un autre client pourrait-il le vouloir ? »** Oui ou peut-être : niveau 3, tout le monde en profite et la fonctionnalité devient même vendable. Non, vraiment spécifique : niveau 4.
> **Et une fonctionnalité entièrement nouvelle ?** On applique le test des 4 questions : si elle a du sens pour d'autres, elle entre au catalogue comme **nouveau module** ; sinon, c'est une extension client.

Comment fonctionne une extension client

Elle a la même forme qu'un module, vit dans `extensions/` et n'est activée que pour son client. Elle se branche uniquement par les mécanismes déjà prévus :

- **Événements** : elle réagit à `BulletinCalcule`, `VenteEffectuee`…

- **Emplacements d'écran** : elle ajoute un onglet, un bouton, un widget.
- **Points d'extension** : des interfaces que le module déclare **volontairement** dans `Public/Hooks/` (ex. `RegleDePrime`, `NumerotationFacture`, `ValidationAbsence`). Le module fournit l'implémentation par défaut ; l'extension la remplace **pour son client seulement**.

```
// api/extensions/sdp-primes/extension.php
return [
    'key'     => 'sdp-primes',
    'client'  => 'sdp',                    // ne peut être activée que pour ce client
    'extends' => ['paie'],                 // module qu'elle complète
    'bind'    => [
        // point d'extension déclaré par le module Paie dans sa vitrine
        \Modules\Paie\Public\Hooks\RegleDePrime::class => SdpRegleDePrime::class,
    ],
    'owner'   => 'équipe intégration',
];
```

Le module Paie ne sait pas que l'extension existe : il appelle `RegleDePrime`, et le socle lui fournit la bonne implémentation selon le client de la requête.

Les 5 règles des extensions

1. **Le module ne connaît jamais l'extension.** Deptrac interdit tout lien de `modules/` vers `extensions/`.
2. **Il manque un point d'extension ?** L'équipe du module l'ajoute proprement, de façon générique et utilisable par tous. On ne contourne jamais le module.
3. **Chaque extension a un responsable, des tests et un contrat** : développement facturé, maintenance annuelle, mise à jour obligatoire quand le module évolue.
4. **Règle des 3 clients** : dès que trois clients demandent la même chose, elle devient un réglage du module (niveau 3) et l'extension est supprimée.
5. **Revue annuelle** : on garde les extensions utiles, on supprime les autres, et celles devenues génériques rentrent dans le module.

Exemples

Demande du client	Réponse
« Je veux un champ *Matricule interne* sur mes contacts »	2 : champ personnalisé
« Mes absences doivent être validées par 2 managers »	2 : circuit de validation
« Je veux mon logo et mes mentions sur les factures »	2 : modèle de document
« Je veux arrondir les prix en caisse à 5 F CFA »	3 : nouvelle option « arrondi » dans Caisse, utile à d'autres
« Ma prime d'ancienneté suit notre accord d'entreprise »	4 : extension `sdp-primes` via le point d'extension `RegleDePrime`
« Envoyez mes ventes à mon logiciel comptable »	5 : webhook ou connecteur
« Je veux gérer mes tournées de livraison »	Test des 4 questions : nouveau module au catalogue si d'autres clients en ont besoin, sinon extension

10 Les 10 règles d'or

Cette page s'affiche au mur. Si toute l'équipe respecte ces dix règles, l'architecture tient dans la durée. Les règles marquées ⚙ sont vérifiées automatiquement en CI.

#	Règle
1	**Un seul code, un seul dépôt, une seule branche principale.** Jamais de branche par produit ni par client.
2	**Un client = une offre + des réglages.** Une demande propre à un client suit l'échelle de personnalisation (chapitre 9) ; jamais de code client dans un module. ⚙
3	**Un module = une fonctionnalité utilisable seule**, validée par le test des 4 questions. Même nom dans `api/modules` et `web/src/modules`.
4	**Un module ne touche jamais l'intérieur d'un autre module.** Il ne peut utiliser que la vitrine d'un module qu'il requiert. ⚙
5	**Deux modules facultatifs l'un pour l'autre coopèrent par événement ou par pont.** ⚙
6	**Chaque table appartient à un module et porte son préfixe.** Clé étrangère seulement vers le socle ou un module requis. ⚙
7	**Tout ce qu'un module expose vérifie qu'il est actif** : routes, tâches, écouteurs, jobs. Le gabarit le fait pour vous. ⚙
8	**Ce qui est lu souvent est en cache ; ce qui est long part en file.**
9	**Le navigateur ne charge que les modules du client.** Un composant partagé va dans le kit d'interface du socle. ⚙
10	**Chaque offre a son test** : client neuf, activation de l'offre, puis ses écrans répondent et les autres sont refusés. ⚙

Situations courantes

Situation	Réponse
« On veut vendre une nouvelle fonctionnalité »	Test des 4 questions → nouveau module (dossier API + dossier web + manifeste) → on l'ajoute à une ou plusieurs offres
« Le commercial veut une nouvelle offre »	Une ligne dans `config/offers.php`. Aucun développement
« Ce client veut que ça marche un peu autrement »	Échelle de personnalisation (chapitre 9) : réglage, configuration, option pour tous, puis extension client en dernier recours
« Le module A a besoin d'une info du module B »	A requiert B ? → vitrine de B. Sinon → pont A-B, ou événement
« Ce composant sert à plusieurs modules »	Il monte dans le kit d'interface du socle
« Tout le monde a besoin de cette fonctionnalité »	Elle va dans le socle (décision d'architecture, notée dans `docs/`)

11 Redémarrer : dans quel ordre construire

Étape 1 : le catalogue

avant d'écrire du code

- Atelier produit + commercial + technique : lister les besoins des clients, les découper en modules avec le test des 4 questions, composer les offres.
- Livrable : `docs/catalogue.md`, avec pour chaque module sa clé, son nom, son objectif, ses droits, ses modules requis et son responsable.

Terminé quand : le catalogue v1 et les offres sont validés par produit et commercial.

Étape 2 : le socle

la fondation

- Clients et base par client, connexion, droits, sociétés, contacts, fichiers, notifications, journal.
- Le **registre des modules** : lecture des manifestes, cache Redis, middleware, filtrage des tâches, activation, extensions par client.
- Les **outils de personnalisation** : réglages, champs personnalisés, modèles de documents, circuits de validation.
- Le shell web, `/me/bootstrap`, le chargement à la demande, le kit d'interface, les emplacements.
- La CI avec tous les garde-fous (Deptrac, ESLint, tests par offre, budgets de performance).

Terminé quand : on peut créer un client, lui activer une offre vide et s'y connecter.

Étape 3 : le module modèle

la référence pour tous les autres

- Construire un premier module complet de bout en bout, choisi simple, par exemple Contrôle d'accès ou Temps et absences.
- En tirer un **gabarit** : `php artisan byplus:make-module` génère les deux dossiers, le manifeste et les tests (même chose pour `make-extension`).

Terminé quand : un développeur qui ne connaît pas le projet crée un module fonctionnel en une journée avec le gabarit.

Étape 4 : les modules, offre par offre

dans l'ordre fixé par le commercial

- On construit les modules d'une offre complète avant de passer à la suivante, pour pouvoir la vendre au plus tôt. Les ponts viennent quand leurs deux modules existent.
- Le code de l'ancien BY+ sert de **référence métier** (règles de gestion, calculs), pas de base à copier. Les adaptations faites pour certains clients dans l'ancien BY+ sont reclassées avec l'échelle de personnalisation.

Terminé quand : chaque offre passe son test et est validée par un client pilote.

Étape 5 : faire passer les clients existants client par client

- Pour chaque module, un **script d'import** des données depuis l'ancien BY+. On le répète sur une copie avant la bascule réelle.
- Bascule d'un client quand tous les modules de son offre (et ses éventuelles extensions) sont prêts. L'ancien BY+ passe en maintenance seule (correctifs), puis s'arrête.

Terminé quand : le dernier client est passé et l'ancien BY+ est arrêté.

Glossaire

Mot	Définition
Socle	Ce que tout client a toujours ; jamais vendu séparément
Module	Une fonctionnalité utilisable seule ; un dossier dans l'API et un dans le web, avec le même nom
Manifeste	La carte d'identité d'un module (`module.php`) : nom, droits, réglages, modules requis, données de départ, tâches
Vitrine	Le dossier `Public/` d'un module : la seule partie que les autres peuvent utiliser
Pont	Le code qui relie deux modules ; il n'est actif que si le client a les deux
Offre	Une liste de modules vendue sous un nom ; pure configuration
Réglage	Une option d'un module, activable client par client, sans code
Point d'extension	Une interface déclarée par un module dans `Public/Hooks/` , que l'on peut remplacer pour un client
Extension client	Un petit module propre à un seul client, qui se branche sur un module sans le modifier
Emplacement	Un endroit nommé de l'interface (onglet, widget) où un module, un pont ou une extension peut ajouter un composant
Shell	La coquille légère de l'application web, toujours chargée ; elle charge ensuite les modules du client
Base centrale / base client	Qui a quoi, d'un côté ; les données métier de chaque client, de l'autre