BY+ · SOCLE · RÉSUMÉ

Le socle BY+, en clair

Le socle est ce que **tout client a toujours**, écrit une fois pour tous : comptes, sociétés, réglages, fichiers, journal, circuits d'approbation. Il vit dans `api/core/`, se branche aux modules par des contrats publiés et ne sort jamais de son propre périmètre. Un client sans Paie a quand même Auth : le socle est une **couche**, pas un empilement.

1. Les 7 idées à retenir

- **Le socle est obligatoire, neutre et stable**. Toute installation a toutes les briques ; aucune licence ne peut en retirer une. Une brique ne connaît aucun module : elle ignore qu'ils existent. Et le `Public/` d'une brique est une API publiée, compatible ascendante pour toute la vie de BY+.
- **15 briques, plus un noyau** qui n'en est pas une. Les 11 lignes du catalogue ne sont pas 11 briques : une ligne = trois briques, et il en manque une côté API — Tenancy.
- **Tenancy est la brique la plus structurante et la moins visible**. C'est elle qui transforme `client.byplus.io` en base `client`, qui bascule la connexion et qui isole les données d'un client de celles d'un autre. Aucun client ne la réclame, donc personne ne l'a écrite dans le tableau.
- **Trois paliers, on descend jamais on ne remonte**. Une dépendance latérale est impossible à maintenir : elle ferait tourner le graphe en rond. Le tout est vérifié en CI par `deptrac.yaml`.
- **Deux racines par brique, et une seule frontière**. `src/` est le code privé, `Public/` est l'API publiée. Une brique sans `Public/` n'est pas une brique de socle : c'est un module interne.
- **Le socle ne décide jamais de ce qu'un client a**. Il ne fait que lire la licence et la mettre en cache. Aucun écran du socle ne peut activer un module ; le code de tous les modules tourne pour tous les clients, et chaque porte d'entrée vérifie.
- **Le journal porte un acteur, pas un utilisateur**. Un job de nuit écrit `system`, un appel par clé API écrit `api_key`, une navigation écrit `user`. Aucune colonne du journal n'est nullable.
- **Ce qui n'est pas dans le socle n'est pas « mis de côté »** : il est mal classé. Contacts, Documents, Budget, NPS, Tâches, Portail client sont des modules ; moyen de paiement est une donnée du module POS.

2. Combien de briques ?

15 briques API, plus un noyau qui n'est pas une brique. Le décompte montrait déjà l'écart : le catalogue parle du menu du client, la réalité parle de domaines d'écran.

Ligne du catalogue	Briques API	Justification
`Comptes, rôles et droits`	`Auth`	1 pour 1
`Sociétés`	`Companies`	1 pour 1
`Paramètres`	`Settings`	1 pour 1
`Fichiers`	`Files`	1 pour 1
`Notifications`	`Notifications`	1 pour 1
`Journal`	`Audit`	1 pour 1
`Circuits de validation`	`Workflows`	1 pour 1

Ligne du catalogue	Briques API	Justification
Tableau de bord	Dashboard	1 pour 1
Personnalisation	Customization	1 pour 1
Mon abonnement · Support · Intégrations	Subscription · Helpdesk · Integrations	1 ligne = 3 briques
Technique	Modules · Sync	le kit d'interface est front, il n'a pas de dossier API
absent du tableau	Tenancy	la brique la plus critique ; voir ci-dessous

2.1 Pourquoi Tenancy manque du tableau

Les 11 lignes du catalogue décrivent le socle vu du client : ce qu'il voit dans son menu. Tenancy est la brique la plus structurante et la moins visible — c'est elle qui transforme un sous-domaine en base, qui bascule la connexion, qui isole les données d'un client de celles d'un autre. Aucun client ne la réclame, donc personne ne l'a écrite dans le tableau. Or sans elle il n'y a pas de BY+ multi-client. C'est la première brique à écrire.

```
client.byplus.io
│
▼
Tenancy : quel client ? ← résolution du sous-domaine
│
├── connexion sur la base client
├── Contexte alimenté : client, société
└── Modules : quels modules ? (Redis)
```

Du sous-domaine à la base : ce que fait Tenancy

2.2 Les trois questions de granularité à trancher

Avant d'écrire le moindre code, trois découpages ne sont pas fixés. Ils changent le nombre de dossiers et donc le coût de chaque brique. Dans ce document, les trois sont traitées comme des briques : c'est le choix par défaut, pas une décision fermée.

#	Question	Enjeu
1	Tenancy est-elle une brique, ou un simple dossier du noyau ?	Brique = testable et lisible ; noyau = moins de couches, mais une brique critique sans isolation
2	Mon abonnement est-il une brique ou une lecture de licence du registre ?	Brique = écran et message d'erreur clairs ; registre = moins de code, mais le client n'a plus d'explication de ce qu'il ne voit pas
3	Sync est-il une brique ou une responsabilité du noyau ?	Brique = le protocole est testable isolé ; noyau = moins de dossiers

3. L'architecture du socle

Le socle ne « monte » pas les modules : il leur fournit un contexte, un journal, des fichiers et des droits. Tout ce qui suit tient dans une seule image — une requête, de bout en bout.

Une requête cliente, de bout en bout

Ce que fait le noyau, et dans quel ordre

GET https://sdp.byplus.io/ventes — aucun paramètre de client n'est accepté : il est déduit



Ce qui n'appartient à aucune brique — Core\Kernel

Contexte, Acteur, Macros, ApiController, Resource. Kernel/ est la seule entrée de core/ sans brick.php : le chargeur fait glob('core/*/brick.php'), donc le noyau est automatiquement exclu du lot des briques.

Un client sans Paie a quand même Auth. Un module ne « monte » pas dans le socle : il s'y branche par Public/ et n'en sort jamais. Une brique ou un module qui réimplémente une mécanique du noyau, c'est un bug — pas une dérogation.

Une requête cliente : le noyau, puis la brique, jamais l'inverse

3.1 Trois caractéristiques

a) Le socle est une couche, pas un empilement. Un client sans Paie a quand même Auth. Un module ne « monte » pas dans le socle : il s'y branche par `Public/` et n'en sort jamais.

b) Un socle n'est pas un ERP. Tout ce qui connaît le métier est un module. Le socle ne sait pas ce qu'est une vente, un salarié ou un stock : il sait qui a le droit d'en créer un.

c) Un socle est obligatoire, neutre et stable.

Propriété	Conséquence
Obligatoire	Toute installation a toutes les briques. Aucune licence ne peut en retirer une.
Neutre	Une brique ne connaît aucun module : elle ignore qu'ils existent.
Stable	Le Public/ du socle est une API publiée : compatibilité ascendante garantie pour toute la vie de BY+.

3.2 Le noyau (Core\Kernel)

Le noyau ne contient que ce qu'aucune brique ne peut posséder : les mécaniques transverses dont personne ne réécrit la version.

```
api/core/Kernel/
├── Context.php   ← l'acteur courant : client, user, société, abilities
├── Actor.php     ← QUI fait l'action : system | user | api_key | client
├── Macros/       ← Collection, Eloquent, Str, Carbon
└── Http/
    ├── ApiController.php  ← permission + tenancy + json, un seul parent
    └── Resource.php
```

Élément	Pourquoi aucune brique ne peut le porter
`Context`	Ni Tenancy — elle ignore l'utilisateur — ni Auth : sinon 8 briques dépendent d'Auth
`Actor`	Le journal doit tracer un job lancé sans utilisateur ; null n'est pas une identité
`Macros`	Sans lui, 15 briques réimplémentent chacune whenNotEmpty() à leur façon
`ApiController`	Sans lui, 15 contrôleurs de base, qui divergent au premier correctif

> **Le Contexte est dans le noyau, pas dans Auth.** Auth l'alimente après authentification ; n'importe quelle brique le lit sans dépendre d'Auth. Sans cela, 8 briques dépendraient d'Auth et le graphe deviendrait illisible. `Kernel/` est la seule entrée de `core/` sans `brick.php` : c'est vrai par construction, le chargeur fait `glob('core/*/brick.php')`. Aucune liste à tenir, aucun drapeau, aucune exception à documenter. C'est ce qui prouve qu'il n'en est pas une.

Ni `Concerns/`, ni `Enums/`, ni `Exceptions/` : un trait appartient à la brique qui publie la fonctionnalité. Utiliser le journal signifie dépendre de Audit — c'est la bonne dépendance : elle est explicite et elle rend la brique propriétaire de sa fonctionnalité. Règle : si une brique ou un module implémente une mécanique déjà dans le noyau, c'est un **bug** — pas une dérogation.

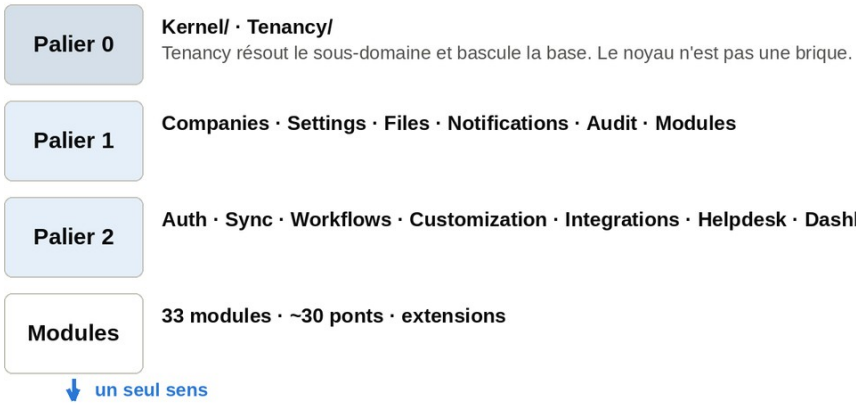
3.3 Les 3 paliers de dépendances

Les 3 paliers de dépendances

Règle : on descend, jamais on ne remonte, jamais on ne va de côté

```
palier(brique) = 1
       + max(palier(dép.))
```

Une brique sans dépendance est au palier 0 ; toute dépendance fait monter d'un cran. Une dépendance latérale ferait tourner le graphe en rond.



Vérifié en CI par deptrac.yaml — cinq règles, cinq filets que le code ne peut pas passer

Palier1NeDependQueDuPalier0
dépendance latérale entre briques d'un même palier
Palier2NeDependQueDesPaliers0et1
Auth qui dépendrait de Workflows
UnModuleNeDependQueDuSocleEtDesModulesRequis
un module qui utilise un autre module sans pont
UnModuleNeConnaitAucunAutreModule
le contournement du registre par un import direct
AucunSocleNImporteDeModule
le socle qui utilise un module, un pont ou une extension

Le mot « palier » remplace « niveau », qui désigne déjà l'échelle de personnalisation et le niveau de réglage d'un champ : le code ne connaît que core_palier, pas core_niveau.

Trois paliers, un seul sens ; cinq règles deptrac pour le tenir

Brique	Dépend de	Et jamais de
`Kernel`	rien	rien — ce n'est pas une brique
`Tenancy`	Kernel	tout le reste
`Companies`	Tenancy	Auth — c'est Auth qui lit Companies
`Settings`	Tenancy	tout
`Files`	Tenancy	Auth, Documents
`Notifications`	Tenancy, Settings	tout
`Audit`	Tenancy	Auth — il lit le Contexte, pas la brique

Brique	Dépend de	Et jamais de
`Modules`	Tenancy	tout — le registre ne connaît que des manifestes sur disque
`Auth`	Tenancy, Companies, Modules	Workflows, Helpdesk
`Sync`	Tenancy, Auth	tout
`Workflows`	Notifications, Audit	Auth — il lit le Contexte
`Customization`	Files, Settings	Auth
`Integrations`	Notifications, Audit	tout
`Helpdesk`	Files, Notifications	Auth
`Dashboard`	Modules	tout
`Subscription`	Modules	tout

> **« Niveau » est un mot ambigu dans ce produit** : il désigne l'échelle de personnalisation et le niveau de réglage d'un champ. Pour les dépendances, on dit **palier**. Le code ne connaît que ce nom-là : `core_palier`, pas `core_niveau`.

Le graphe est acyclique et sans dépendance latérale. Il est vérifié en CI sur le code importé, exactement comme les liens entre socle, modules, ponts et extensions. `Files` ne dépend de `Tenancy` que pour deux choses — nommer un dossier de préfixe par client, et purger les fichiers orphelins lors d'une bascule de base. Si ces deux besoins disparaissent, la dépendance tombe et `Tenancy` pourrait remonter au palier 1, ce qui placerait 6 briques au palier 2 au lieu de 8.

4. Les 15 briques, une par une

4.1 Vue d'ensemble

Brique	Palier	Dossier	Ce qu'elle possède	Ce qu'elle publie
Noyau	0	`Kernel/`	Contexte, acteur, macros, contrôleurs de base	rien : ce n'est pas une brique
Tenancy	0	`Tenancy/`	Résolution du client, bascule de base	ClientResolver
Companies	1	`Companies/`	sociétés, agences, informations légales	CompanyDirectory, CurrentCompany
Settings	1	`Settings/`	pays, devises, unités, langues, numérotations	Referentials, Sequencing
Files	1	`Files/`	stockage S3, métadonnées, quotas	Storage, Attachments
Notifications	1	`Notifications/`	in-app, canaux, tokens d'appareils	Notifier
Audit	1	`Audit/`	journal des actions, journal des accès	Journal
Modules	1	`Modules/`	cache Redis, registre, extensions par client	ModuleContext
Auth	2	`Auth/`	utilisateurs, rôles, permissions, sessions	Permissions, UserDirectory
Sync	2	`Sync/`	curseurs de synchronisation	Synchronizable
Workflows	2	`Workflows/`	circuits, étapes, validations	Approvable
Customization	2	`Customization/`	champs, listes de valeurs, modèles, libellés	Customizable, Labels, DocumentRenderer

Brique	Palier	Dossier	Ce qu'elle possède	Ce qu'elle publie
Integrations	2	`Integrations/`	clés API, scopes, webhooks sortants	Webhook, ApiKey
Helpdesk	2	`Helpdesk/`	tickets, conversations, catégories	Ticket
Dashboard	2	`Dashboard/`	accueil, disposition des widgets	Widget
Subscription	2	`Subscription/`	rien : il lit la licence	Subscription

4.2 Chaque brique a trois verbes : possédé, exposé, refusé

Le motif est le même partout — et c'est la colonne « refuse » qui est la plus utile, parce qu'elle dit où s'arrête le socle.

Brique	Refuse
`Tenancy`	De gérer les clients : créer, facturer, suspendre un client n'est pas du ressort du socle.
`Auth`	Le droit d'inventer un droit. Chaque droit appartient au module ou à la brique qui l'a déclaré.
`Companies`	Les contacts et les collaborateurs, qui la référencent par clé étrangère.
`Settings`	Les réglages propres à un module : ceux-ci sont déclarés dans le manifeste du module.
`Files`	D'être une GED. Documents est un module ; Files est l'infrastructure en dessous.
`Notifications`	La règle métier « quand » notifier. Elle est dans le module.
`Audit`	D'être un journal technique. C'est un journal métier, lisible par un humain.
`Modules`	De contenir de logique métier. Le registre ne sait pas ce qu'est une vente ; il sait qui peut en faire une.
`Sync`	D'accorder un accès direct à la base. Toute surface externe passe par le protocole.
`Dashboard`	De contenir une statistique métier. Le socle ne calcule pas un chiffre de vente.
`Integrations`	Toute clé d'administration. Une clé du socle sert au client, jamais à piloter BY+.
`Workflows`	De connaître le métier : c'est le module qui déclare ce qui se soumet.

4.3 Les points clés, brique par brique

- **Settings** : la numérotation est un service, pas une convention. Chaque module déclare sa séquence ; le socle la fait avancer sans trou et sans collision, y compris en concurrence.
- **Notifications** : l'envoi est toujours asynchrone. Une notification ne bloque jamais une requête métier, et un canal en panne n'en bloque pas un autre.
- **Audit** : toute consultation de données par un tiers est un événement de journal, pas une chasse gardée — qui, chez quel client, à quelle heure, ce qu'il a consulté.
- **Modules** : le socle filtre seul. C'est le premier test pratique du chargement à la demande — si un widget est exposé pour un client qui n'a pas le module, le filtrage est cassé.
- **Sync** : un module expose ce qu'il veut synchroniser ; le socle gère le protocole. C'est le pendant API de la règle « le module déclare, le socle exécute ».
- **Customization** : les valeurs des champs sont dans une table du socle, pas dans la table du module. Un module ne perd donc aucun champ ajouté par un client, et n'a jamais à être migré.
- **Files** : le trait n'est pas la brique. Le trait est la manière de s'en servir ; la brique est l'implémentation. Sans elle, personne ne stocke, ne vérifie le quota, ne génère les vignettes et ne contrôle l'accès.

5. Le gabarit : privé et publié

5.1 Faut-il un dossier, ou un sous-dossier ?

Une brique est un domaine d'écran. Quatre questions suffisent pour trancher.

Question	Si oui
A-t-il son propre menu ?	→ nouvelle brique
A-t-il ses propres tables que rien d'autre ne possèderait ?	→ nouvelle brique
A-t-il ses propres droits, qu'aucune autre brique ne déclare ?	→ nouvelle brique
Aurait-il son propre responsable ?	→ nouvelle brique

Trois « oui » sur quatre → nouvelle brique. Sinon, sous-dossier de la brique existante. Cas litigains : Office et les departements sont des **modèles** de Companies ; ValidationStep une ligne d'un circuit ; UserSession une ligne d'un utilisateur.

5.2 L'arborescence

```
byplus/
├── api/
│   ├── core/                  ← LE SOCLE — PSR-4 : Core\
│   │   ├── Kernel/            ← palier 0 — sans brick.php
│   │   ├── Tenancy/           ← palier 0
│   │   ├── Companies/ Settings/ Files/
│   │   │   Notifications/ Audit/ Modules/     ← palier 1
│   │   └── Auth/ Sync/ Workflows/ Customization/
│   │       Integrations/ Helpdesk/ Dashboard/  ← palier 2
│   │       Subscription/
│   ├── modules/               ← PSR-4 : Modules\   (modules)
│   ├── bridges/               ← PSR-4 : Bridges\   (ponts)
│   ├── extensions/            ← PSR-4 : Extensions\
│   └── config/
│       ├── core/              ← un fichier par brique (déploiement)
│       ├── offers.php         ← LES OFFRES (configuration pure)
│       └── deptrac.yaml       ← les liens autorisés (deptrac)
```

Quatre racines PSR-4 : Core\, Modules\, Bridges\, Extensions

Il y a trois natures de réglage, et chacune part dans une direction différente. Les confondre est la première cause d'un réglage qu'un client ne peut pas modifier alors qu'il le devrait : un réglage de **déploiement** vit dans `config/core/<brique>.php` + `.env` ; un réglage de **client** vit dans la base du client, exposée par un contrat ; un réglage de **module** est déclaré dans `settings` du manifeste. Un réglage de client n'est jamais dans un fichier de config, et un secret n'est jamais dans la base du client.

5.3 brick.php n'est pas module.php

C'est la confusion à éviter en premier, parce que tout le reste en découle. `brick.php` tient en cinq lignes, et cinq lignes seulement :

```php
// api/core/Companies/brick.php
return [
    'key' => 'companies',
    'name' => 'companies',
    'description' => 'Sociétés, agences, informations légales et fuseaux',
    'owner' => 'core',
    'api_version' => '1.0',
];
```

Un identifiant, un nom, une description, un responsable, une version d'API. Rien d'autre. Le test d'identité tient en une question : est-ce que cette clé décrit **qui est** la brique, ou **ce qu'elle fait** ? Si une clé du manifeste commence à contenir une condition, une expression ou une liste de classes, elle est du code déguisé.

Clé	module.php	brick.php	Pourquoi
`key, name, version, description`	oui	oui	l'identité et la traçabilité d'installation

Clé	module.php	brick.php	Pourquoi
`owner`	non	oui	le noyau n'est pas maintenu par une équipe de module
`api_version`	non	oui	le Public/ d'une brique est une API : elle se versionne
`requires`	oui	jamais	une brique n'en requiert pas une autre : elle en hérite par le palier. Un requires rouvrirait la dépendance latérale
`permissions`	oui	jamais	les droits sont des classes dans src/Policies ; les lister oblige à les ressaisir, donc à les laisser diverger
`roles`	oui	jamais	un rôle est un paquet de permissions détenu par Auth
`sellable`	oui	jamais	le socle est obligatoire : la question ne se pose pas, donc le drapeau ne dit rien
`settings`	oui	palier 1 seulement	au palier 2, un réglage client est un champ personnalisé : il vit dans Customization
`schedule`	filtré par client	pour tous	une tâche du socle est une tâche de tous les clients ; si elle doit être filtrée, elle appartient à un module
`tables`	oui	jamais	le schéma se déduit des migrations ; tout le socle partage le préfixe core_
`Public/`	vitrine, usage interne	API publiée	c'est tout l'objet du chapitre suivant

> **Deux raisons justifient le fichier brick.php**, et les deux sont dans le code. **Raison 1 — le préfixe de table** : le dossier s'appelle Companies, mais ses tables ne peuvent pas s'appeler companies. Un préfixe global est déclaré une fois et chaque migration y fait référence au lieu de recopier la chaîne : `Table::for('companies')` → `core_companies`. **Raison 2 — la version d'API** : `api_version` permet à un module de dire « j'ai besoin de la version 1.2 de Files », et au chargeur de refuser l'installation plutôt que de planter à la première jointure.

5.4 Le template d'une brique

```
api/core/Companies/
├── brick.php              ← identité de la brique (5 clés)
├── Public/                ← API PUBLIÉE (compatibilité ascendante)
│   ├── Contracts/         ← interfaces consommées par les modules
│   ├── Concerns/          ← les traits que la brique MET À DISPOSITION
│   ├── Events/            ← événements émis
│   ├── DTOs/              ← objets de données en lecture
│   ├── Hooks/             ← points d'extension remplaçables
│   └── Resources/         ← sérialisation stable pour les apps
└── src/                   ← CODE PRIVÉ (nul de l'extérieur ne l'atteint)
    ├── Models/            ← Eloquent + Models/Enums/
    ├── Actions/           ← 1 classe = 1 action métier
    ├── Http/{Controllers,Requests,Middleware}/
    ├── Services/          ← seulement s'il y a de l'orchestration
    ├── Policies/          ← les droits
    ├── Observers/  Events/  Listeners/  Jobs/
    ├── Console/Commands/
    ├── Exceptions/
    ├── database/{migrations,seeders,factories}/
    ├── routes/{web,console}.php
    └── lang/{fr,en}.json
```

Deux racines seulement : src/ et Public/ — la frontière privé/publié est la seule qui compte

Une brique sans `Public/` n'est pas une brique de socle : c'est un module interne. Si elle n'a rien à publier, elle appartient à une autre brique. Et l'inverse : si deux briques veulent publier le même contrat, ce sont deux contrats ou c'est une brique unique. On ne duplique jamais un `Public/`.

Le noyau ne publie aucun trait. `Files` publie `HasAttachments`, `Workflows` publie `Workflowable`, `Customization` publie `HasCustomFields`, `Audit` publie `Journalizable`, `Companies` publie `BelongsToCompany`. Un trait sans fonctionnalité à gérer n'a pas de raison d'être centralisé.

5.5 Les 5 portes d'entrée

Toute brique expose la même quintuple, sans exception. Une brique qui n'en a pas cinq est incomplète. Différence avec un module : une brique de socle n'est **jamais** gardée par `module:<clé>`, elle est toujours active.

Porte	Dossier	Obligatoire
HTTP	`src/routes/web.php`	oui
Console	`src/routes/console.php + src/Console/Commands`	oui
Événements	`src/Events + src/Listeners`	oui
File d'attente	`src/Jobs`	oui
Droits	`src/Policies`	oui

5.6 Les contrats publiés

C'est la liste exacte de ce que les modules ont le droit d'utiliser.

Brique	Contrat	Ce qu'un module peut faire
`Tenancy`	`ClientResolver`	Savoir dans quel client il s'exécute
`Kernel`	`Context`	Client, utilisateur, société courants ; can($droit) ; actor()
`Kernel`	`Actor`	Savoir qui fait l'action : system, user, api_key, client — pour journaliser depuis un job
`Companies`	`CompanyDirectory`	Lister les sociétés accessibles, lire une société
`Companies`	`CurrentCompany`	Société courante, fuseau horaire
`Settings`	`Referentials`	Devise, unité, langue, formats
`Settings`	`Sequencing`	Prochain numéro d'une séquence déclarée par le module
`Files`	`Storage`	Écrire, lire, supprimer, URL, consommation
`Files`	`Attachments`	Attacher un fichier à son objet, le supprimer avec lui
`Notifications`	`Notifier`	in-app, mail, push, SMS, gabarits
`Audit`	`Journal`	Tracer une action, lire l'historique d'un objet
`Modules`	`ModuleContext`	active(), modules(), bridgeActive()
`Auth`	`Permissions`	Vérifier un droit, lister les droits du rôle courant
`Auth`	`UserDirectory`	Lire un utilisateur, le chercher, lister par société
`Workflows`	`Approvable`	Soumettre, approuver, rejeter, lire l'état d'un circuit
`Customization`	`Customizable`	Déclarer des champs sur ses objets, les lire, les valider
`Customization`	`DocumentRenderer`	Rendre un PDF ou un mail depuis un modèle
`Customization`	`Labels`	Surcharger un libellé
`Customization`	`OptionList`	Lire une liste de valeurs
`Integrations`	`Webhook`	Déclarer un événement, obtenir une clé et des scopes
`Helpdesk`	`Ticket`	Ouvrir, répondre, clôturer, relier à un objet métier

Brique	Contrat	Ce qu’un module peut faire
`Sync`	`Synchronizable`	Marquer un modèle comme synchronisable
`Dashboard`	`Widget`	Déclarer un widget et ses paramètres
`Subscription`	`Subscription`	Lire l’offre, les modules, les places restantes, l’échéance

5.7 Ce que le socle n’expose jamais

Interdit	Raison
Un modèle, une table, un contrôleur	Le module passe par le contrat, jamais dans la base du socle
Une méthode « utilitaire » fourre-tout	Un contrat est un domaine — fichiers, devises, droits — pas une liste de fonctions
Un droit déclaré par le socle pour le compte d’un module	Chaque droit appartient à son module
Un événement métier inventé par le socle	Le socle émet des événements de socle : UserDeactivated, SessionRevoked. Pas SaleCompleted.
Un accès en écriture à la licence	Elle est une entrée du système, pas une de ses sorties
Un accès direct au disque ou au chemin d’un fichier	Le module stocke un identifiant opaque, jamais une clé S3

6. Le registre des modules

La brique la plus structurante. Sans elle, « un client = une licence » n’est pas une propriété du système. Le socle ne décide jamais de ce qu’un client a : il ne fait que lire la licence et la mettre en cache. Aucun écran du socle ne peut activer un module.

Activer un module chez un client

Le socle ne décide jamais de ce qu'un client a : il lit la licence, la met en cache, et ne l'écrit jamais.

Licence du client (entrée du socle) → offre + modules + places + dates → lecture seule

1. Vérifier
licence, modules déjà actifs, extensions, quota
refuse si la licence ne permet pas

2. Compléter
activer d'abord les modules requis
récursif, profondeur 2

3. Préparer
droits, rôles, données de départ
seeders rejouables

4. Activer
écrire le contexte, base comprise, en dernier
rien ne tourne à moitié

5. Prévenir
cache vidé, version changée
le client recharge au GET /me/bootstrap

Toutes les tables existent déjà chez tous les clients
Activer un module est instantané : les tables sont là, vides. C'est ce qui rend l'activation atomique et annulable.

```
Redis  ⟶  ModuleContext
  tenant:{id}:modules     actifs
  tenant:{id}:extensions  activées
  tenant:{id}:version     changements
```

Le code d'un module tourne pour tous les clients : chaque porte d'entrée vérifie.

Routes API	Tâches planifiées	Écouteurs	Jobs en file	Événements émis
QUI VÉRIFIE	QUI VÉRIFIE	QUI VÉRIFIE	QUI VÉRIFIE	QUI VÉRIFIE
middleware module:<clé>	schedule filtrée par client	ModuleListener ignore l'événement	revérifient le module au démarrage	le socle ne déclare pas le module
SI ABSENT	SI ABSENT	SI ABSENT	SI ABSENT	SI ABSENT
403, avant le contrôleur	tâche non exécutée	effet sans effet	job abandonné	aucun envoi

Règle absolue : une porte non gardée est un bug de sécurité, pas un bug fonctionnel.
Les trois pièges rendus impossibles : un module qui se déclare lui-même disponible (seul ModuleContext expose active()) ; un cache invalidé à la main (toute modification de licence incrémente la version, le cache se reconstruit seul) ; une liste de modules en dur dans le code (les modules viennent de la lecture des dossiers).

De la licence au cache : cinq étapes, aucune migration, toutes relançables

Toutes les tables existent déjà chez tous les clients : les tables sont là, vides. C’est ce qui rend l’activation **atomique** et **annulable**. Le code de tous les modules est chargé une seule fois et reste en mémoire pour tous les clients ; le filtrage se fait à chaque porte d’entrée, pas au chargement du code.

Source	Chargé	Mis en cache
`Manifestes modules/*/module.php`	1 fois au démarrage	mémoire + opcache
`Fichiers de routes de chaque module`	1 fois au démarrage	mémoire
`Événements et emplacements déclarés`	1 fois au démarrage	mémoire
`Modules actifs du client`	à la demande	Redis, 0 requête SQL
`Droits du client`	à la demande	Redis, 0 requête SQL
`Extensions du client`	à la demande	Redis, 0 requête SQL

6.1 Les 5 portes d’entrée gardées

Porte d’entrée	Qui vérifie	Si le module est absent
Routes API	Middleware module:<clé> enveloppant les routes du module	403, avant le contrôleur
Tâches planifiées	Le planificateur n’exécute la schedule du manifeste que pour les clients qui l’ont	Tâche non exécutée

Porte d’entrée	Qui vérifie	Si le module est absent
Écouteurs	Classe de base ModuleListener : ignore l’événement	Effet sans effet
Jobs en file	Revérifient le module au démarrage	Job abandonné
Événements émis	Le socle ne déclare pas le module comme écouteur	Aucun envoi, aucun traitement

> **Règle absolue : une porte non gardée est un bug de sécurité, pas un bug fonctionnel. Le test correspondant est obligatoire pour chaque module et il est dans la CI.**

6.2 Résolution des exigences, puis activation

```
active(pos)
├─ requires catalogue → activé
├─ requires contacts → activé
├─ pont pos-entrepot → actif car pos ET entrepôt sont présents
├─ pont pos-loyalty  → actif car pos ET fidélité sont présents
└─ pont pos-nps      → inactif (nps absent)
```

Activer un module chez un client, c’est cinq étapes, aucune migration, chacune relançable sans risque. Toutes les tables existent déjà chez tous les clients : les tables sont là, vides. C’est ce qui rend l’activation atomique et annulable.

Étape	Action	Remarque
1. Vérifier	Licence, modules déjà actifs, extensions, quota	Refuse si la licence ne permet pas
2. Compléter	Activer d’abord les modules requis	Récursif, profondeur 2
3. Préparer	Droits, rôles, données de départ (seeders déclarés)	Seeders rejouables (upsert)
4. Activer	Écrire le contexte, base comprise, en dernier	L’ordre est important : rien ne tourne à moitié
5. Prévenir	Cache vidé, version changée	Le client recharge au GET /me/bootstrap suivant

6.3 Les trois pièges à rendre impossibles

Piège	Prévention
Un module qui se déclare lui-même disponible	Seul ModuleContext expose active(). Aucun if sur un nom de module dans un module.
Un cache invalidé à la main	Toute modification de licence incrémente la version. Le cache se reconstruit seul.
Une liste de modules en dur dans le code	Les modules viennent de la lecture des dossiers. Aucune liste dans le socle.

7. Ce que le socle ne contient pas

C’est la moitié la plus utile de la spécification. Un candidat qui ne trouve pas sa case n’est pas « mis de côté » : il est mal classé. Quelques exemples qui suffisent à fixer la règle :

Candidat	Ce que c’est vraiment	Raison
Contacts · Collaborateurs	2 modules vendables seuls	Un client RH seul n’a pas de tiers, et peut avoir des salariés sans être une société.

Candidat	Ce que c’est vraiment	Raison
Catalogue produits · Formulaires	2 modules de base (sellable: false)	Requis par POS, Entrepôt, Facturation, NPS ; jamais vendus seuls ; inutiles à un client RH.
Documents	Module	GED que l’utilisateur ouvre depuis un menu. Files est l’infrastructure en dessous.
Chat · Mail · Fil d’actualité · Notes de service	4 modules	Communiquer n’est pas universel : le socle notifie, il ne fait pas de messagerie.
Moyens de paiement	Donnée du module POS	Le socle ne connaît pas les moyens de paiement.
Search transverse	À trancher	Une recherche globale est séduisante, mais elle impose à chaque module d’exposer un index.

7.1 Les confusions à institutionnaliser

Paire	Différence	Conséquence si on les confond
Files (socle) / Documents (module)	Infrastructure de stockage / GED	Le module perd l’accès au stockage mutualisé et au quota
Settings (socle) / réglages de module	Référentiels et numérotation / options propres	Un paramètre universalisable finit caché dans un module
Utilisateur (socle) / Contact / Collaborateur	Compte qui se connecte / Tiers extérieur / Salarié	Un client externe obtient des droits internes
Workflows (socle) / l’objet validé	Le moteur / l’objet	Le moteur connaît le métier, et casse à chaque nouveau module
Customization (socle) / extension client	Champs, listes, modèles, libellés / tout le reste	Une extension se multiplic pour ce qui était une configuration

8. Règles et garde-fous

Les 13 règles ci-dessous sont vérifiées automatiquement en CI. Ce ne sont pas des conseils.

#	Règle	Vérification
1	Le socle n’utilise aucun module, pont ou extension. Ni en lecture, ni en écriture, ni en test.	graphe de dépendances
2	Le socle s’applique sa propre règle : aucune brique n’atteint l’intérieur d’une autre, seulement Public/.	idem
3	Seules les couches inférieures sont autorisées. Pas de cycle, pas de dépendance latérale.	idem
4	Aucune brique, aucun module ne lit ni n’écrit les lignes d’une autre brique. Y référencer une clé étrangère est autorisé et normal.	requêtes SQL analysées
5	Toute clé étrangère pointe vers le socle ou vers un module requis. Jamais vers un module facultatif.	intégrité du schéma
6	Toute table porte un préfixe unique : core_* pour le socle, pos_* pour POS.	structure des migrations
7	Un module peut ajouter tables, droits, routes, tâches, widgets, événements. Il n’ajoute pas de brique.	le registre refuse une brique inconnue
8	Toute porte d’entrée d’un module est gardée par le registre.	test de gating par module
9	Le socle n’écrit jamais la licence, ni l’état d’un client hors de sa base.	analyse des modèles écrits

#	Règle	Vérification
10	Aucune liste de modules, d'offres ou de clients en dur dans le socle.	recherche de tableaux de chaînes
11	Tout ce qui est lu souvent est en cache, tout ce qui est long part en file.	interdiction de SQL dans les points d'entrée
12	Le socle n'implémente aucune mécanique déjà fournie : contexte, pagination, formats, upload, journal, workflow.	recherche de classes dupliquées
13	Zéro nom de client dans le code.	un test échoue si un nom apparaît

> **La règle 4, formulée comme il faut.** « Interdit de toucher aux tables d'une autre brique » est inapplicable : une dépense doit pouvoir joindre les pièces qui lui sont rattrapées, et personne ne songerait à interdire une référence. Ce qui est interdit, c'est lire ou écrire des lignes dans une table qui n'est pas la sienne. Ce qui est normal, c'est y pointer une clé étrangère. La distinction est celle entre `SELECT * FROM core_companies` et `companies_id = 42` : le premier casse le contrat, le second est le contrat.

8.1 Rendre les paliers exécutables

Une règle qu'on vérifie à l'œil est une règle qu'on violera. Les quatre règles de dépendance tiennent dans un fichier de trente lignes, parce que les paliers sont déclarés une fois et que les règles en découlent.

```
# config/core/deptrac.yaml
deptrac:
  layers:
    - { name: Kernel,  collector: { type: directory, value: core/Kernel/. } }
    - { name: Palier0, collector: { type: directory, value: core/Tenancy/. } }
    - { name: Palier1, collector: { type: directory, value: core/(Companies|
        Settings|Files|Notifications|Audit|Modules)/. } }
    - { name: Palier2, collector: { type: directory, value: core/(Auth|Sync|
        Workflows|Customization|Integrations|Helpdesk|Dashboard|Subscription)/. } }
    - { name: Modules, collector: { type: directory, value: modules/. } }

  ruleset:
    Palier1NeDependQueDuPalier0:
      from: Palier1
      to: [Kernel, Palier0]
    Palier2NeDependQueDesPaliers0et1:
      from: Palier2
      to: [Kernel, Palier0, Palier1]
    UnModuleNeDependQueDuSocleEtDesModulesRequis:
      from: Modules
      to: [Kernel, Palier0, Palier1, Palier2]
    UnModuleNeConnaitAucunAutreModule:
      from: Modules
      to: [~Modules]
    AucunSocleNImporteDeModule:
      from: [Kernel, Palier0, Palier1, Palier2]
      to: [Modules]
```

Cinq règles, et chacune est un filet que le code ne peut pas passer

La matrice des modules n'est pas écrite : elle est générée. Un tableau nbrModule × nbrModule écrit à la main, et il mentra à la première ligne. Le manifeste, lui, ne peut pas mentir : il est lu au démarrage. La commande lit les `requires` des manifestes, croise avec les ponts déclarés, et produit le fichier — versionné, donc révisable en diff, mais généré, donc jamais périmé sans que la CI le dise. C'est le même choix que pour `config/offers.php` : la donnée pure est un fichier, la logique qui la dérive est une commande.

9. Le plan de construction

Les questions de granularité sont tranchées : Tenancy, Sync et Mon abonnement sont des briques, et le socle ne bouge plus pendant que les modules se construisent. Il reste donc des choses à faire, dans cet ordre.

Ordre	Ce qui se construit	Terminé quand
1	Kernel + Tenancy	client.byplus.io est résolu en base client, en 0 requête inutile, et deux appels simultanés de clients différents ne se croisent jamais
2	Companies + Settings + Files	un client a ses sociétés, ses devises, et stocke un fichier avec son quota vérifié
3	Audit + Notifications + Modules	le journal porte un acteur qui n'est jamais nul ; le registre lit la licence et la met en cache
4	Auth	un agent de la console et un utilisateur client ont deux portes séparées ; les droits viennent des classes de src/Policies
5	Les contrats publiés	un module de démonstration utilise 5 contrats sans jamais toucher au src/ d'une brique
6	Workflows + Customization	un module écrit six lignes et obtient soumission, approbation, notifications et historique
7	Sync + Dashboard + Subscription	un widget exposé à tort pour un module absent fait échouer la CI
8	Integrations + Helpdesk	un webhook part signé, et un ticket se traite dans le socle
9	Les 13 règles en CI	deptrac passe, la matrice des modules est régénérée, aucun nom de client n'existe dans le code

9.1 Checklist de recette

Le socle est prêt quand un client doté d'une licence se connecte, trouve une installation cohérente, et ne peut pas atteindre ce qu'il n'a pas payé.

- **Cohérence** : le sous-domaine ouvre la bonne base ; la société courante et son fuseau sont corrects ; le compteur d'une séquence ne saute pas, même en concurrence ; un fichier déposé est servi, et purgé naturellement s'il devient orphelin.
- **Garde-fous** : un client sans module ne voit ni sa route, ni sa tâche, ni son écouteur, ni son job ; la modification d'une licence invalide le cache sans intervention manuelle ; un job de nuit est journalisé comme `system` ; aucun nom de client n'existe dans le code.
- **Structure** : chaque brique a bien ses deux racines `src/` et `Public/` ; `brick.php` tient en cinq clés ; les migrations sont chargées en place, sans copie ; les tests vivent dans `tests/{Feature,Unit}/<Brique>/` ; `deptrac.yaml` passe.

> **Rappel. Le socle n'est pas une couche technique : c'est la traduction dans le code de la phrase « tout client a toujours ceci ». Toute proposition qui ne peut pas être formulée ainsi appartient à un module, à un pont ou à une extension.**