BY+ · FRONT (CORE-APP)

Faire communiquer les modules dans le front

Version du 7 octobre 2026. Complète le `README.md` de core-app et `docs/architecture.pdf` de l'API (§ 6, « Comment les modules travaillent ensemble »).

> **La règle de base**
>
> **Un module front n'importe jamais un autre module.** ESLint le bloque déjà (`no-restricted-imports` sur `@/modules/*` , `@/bridges/*` , `@/extensions/*`). Comme dans l'API, les modules coopèrent seulement par les quatre moyens décrits ci-dessous.

1. Les quatre moyens, en miroir de l'API

Besoin	Dans l'API	Dans le front
Lire les données d'un module qu'on **requiert** (Entrepôt lit les produits du Catalogue)	Vitrine `Public/Contracts`	**Appeler l'API de ce module** (`/catalogue/…`) depuis son propre `api.ts` . Le contrat entre les deux, c'est l'API, jamais le code Vue de l'autre module.
Afficher quelque chose **dans l'écran d'un autre module** (onglet « Stock par entrepôt » sur la fiche produit)	—	**Emplacement** (*slot*) : le module hôte déclare un emplacement nommé, l'autre module y dépose son composant.
Deux modules **facultatifs l'un pour l'autre** (POS et Entrepôt)	Pont `pos-entrepot`	**Pont front** `src/bridges/pos-entrepot/bridge.ts` , chargé seulement si le client a les deux modules.
Prévenir que quelque chose s'est passé (un mouvement validé : rafraîchir un compteur)	Événement	**Bus d'événements typé** dans `src/core` . Il sert à l'interface seulement : la logique métier reste dans l'API.

2. Ce que le noyau (`src/core`) doit fournir

Le noyau sait déjà découvrir les modules (`registry.ts`), charger seulement ceux du client et monter leurs routes et leur menu (`loader.ts`). Il lui manque les éléments suivants, à construire une fois, **avant le premier module**.

2.1 `requires` dans la carte d'identité

Ajouter `requires?: string[]` à `FrontModule` , avec la même valeur que `module.php` côté API. Au chargement, un module requis absent du front est signalé en développement.

2.2 Les emplacements

- `src/core/modules/slots.ts` : le registre *nom de l'emplacement → composants*, rempli par le loader.
- `src/core/ui/components/ModuleSlot.vue` : posé par le module hôte, il affiche les composants déposés, avec un contexte.
- Un module ou un pont déclare ses composants dans `slots` . Ils sont chargés à la demande (`() => import(…)`) et filtrés par droit s'ils en déclarent un.
- Nom d'un emplacement : `{module}.{écran}.{zone}` , par exemple `catalogue.product.tabs` , `catalogue.product.actions` ou `dashboard.widgets` . Le module hôte les liste dans son `module.ts` , pour que les autres sachent où se brancher.

```
<!-- modules/Catalogue/views/ProductView.vue : l'hôte ne sait pas qui remplit -->
<ModuleSlot name="catalogue.product.tabs" :context="{ productId }" />

// modules/Entrepot/module.ts : Entrepôt requiert Catalogue, il se branche directement
export default defineModule({
  key: 'entrepot',
  requires: ['catalogue'],
  routes: [...],
  slots: {
    'catalogue.product.tabs': {
      component: () => import('./components/ProductStockTab.vue'),
      label: 'entrepot.product_tab',
      permission: 'entrepot.stock.view',
    },
  },
})
```

2.3 Les ponts

- `defineBridge({ key, when, routes?, slots?, listeners? })` dans `src/core/modules/types.ts`.
- Un registre `import.meta.glob('@/bridges/*/bridge.ts')`, comme celui des modules.
- Dans `loadModules`, les ponts sont chargés seulement si **tous** les modules de `when` sont actifs chez le client.

```
// src/bridges/pos-entrepot/bridge.ts
export default defineBridge({
  key: 'pos-entrepot',
  when: ['pos', 'entrepot'],
  slots: {
    'pos.shop.settings': { component: () => import('./SupplyingWarehouse.vue') },
  },
})
```

2.4 Le bus d'événements

`src/core/events.ts` fournit `emit(nom, données)` et `on(nom, rappel)`. Les événements sont typés par une liste connue (`'entrepot:stock-moved'`, `'pos:sale-completed'`…), sur le modèle des noms d'événements de l'API. Un écran qui écoute se désabonne quand il est démonté.

2.5 ESLint et tests

- Un pont peut importer les deux modules de son `when`, et rien d'autre. Une extension n'importe que `@/core`.
- Tests Vitest du loader :
  - un pont est chargé seulement quand ses deux modules sont actifs ;
  - un emplacement est rempli, et reste vide si aucun module ne le remplit ;
  - un composant avec un droit est caché si l'utilisateur n'a pas ce droit.

3. Exemples avec les modules Commerce

Cas	Moyen	Mise en place
Choisir un produit dans un mouvement de stock	API du module requis	`modules/Entrepot/api.ts` appelle `/catalogue/products?q=`. Rien n'est importé de `modules/Catalogue`.
Stock par entrepôt sur la fiche produit	Emplacement	Catalogue pose `catalogue.product.tabs` ; Entrepôt y dépose son onglet. Pas de pont : Entrepôt requiert Catalogue.

Cas	Moyen	Mise en place
Entrepôt d'approvisionnement d'une boutique	Pont	`bridges/pos-entrepot` dépose un réglage dans `pos.shop.settings`, seulement chez les clients qui ont POS et Entrepôt.
Rafraîchir le compteur « stock bas » après une validation	Événement	L'écran de mouvement émet `entrepot:stock-moved` ; le widget du tableau de bord écoute et recharge.

4. Ordre conseillé

1. **Noyau** : emplacements, ponts, `requires`, bus, règles ESLint et tests. C'est une petite brique, à finir avant le premier module.
2. **Catalogue** : premier module, il pose ses emplacements (onglets et actions de la fiche produit).
3. **POS** et **Entrepôt**, qui s'y branchent ; puis le pont `pos-entrepot`.

> **À ne pas faire**
>
> - Importer un composant, un store ou un `api.ts` d'un autre module, même requis.
> - Tester dans le noyau si un module est actif (`if (modules.includes('pos'))`) : c'est le rôle des ponts.
> - Faire passer de la logique métier par le bus (calculs, écritures) : elle reste dans l'API.
> - Partager un composant en le copiant : un composant utile à plusieurs modules va dans `src/core/ui`.