Scénario de test — Intégration Fidélité ↔ Shop (2026-08-19)

Couvre tout ce qui a été implémenté aujourd'hui dans les 3 dépôts : sécurité recherche client, provisioning `User`, activation via `installModules()`, sync Shop/ShopUser, flux prepare→confirm→checkout avec points.

---

0. Prérequis — à faire AVANT de tester

0.1 Migrations à lancer

Dépôt	Branche	Commande	Portée
`fidelity-api`	(courante)	`php artisan tenants:migrate`	tenant — `society_id` sur `users`/`shop_users`
`core-api`	`by-plus-fidelity-develop`	`php artisan migrate`	**centrale** — `fidelity_activated_at` sur `tenant_information`
`shop-core-api`	`by-plus-shop-mini-develop`	`php artisan tenants:migrate`	tenant — `fidelity_sale_id` sur `shop_checkouts`

0.2 Seeder à relancer (core-api, base centrale)

Le module `FIDELITY` (`SuitModulesEnum::FIDELITY`) doit exister en base pour que `installModules()` puisse le trouver :

```bash
php artisan db:seed --class="Core\v1\Module\Database\Seeders\SuitAppModuleSeeder"
```

Idempotent (`updateOrCreate`), sans risque de le relancer même si déjà fait.

0.3 Config à vérifier (`.env` de chaque dépôt — pas les `.env.example`)

⚠️ **Point le plus probable de blocage silencieux** : ces 3 dépôts doivent tous pointer vers le **même** core-api (celui de la branche `by-plus-fidelity-develop`, où vit `FidelityActivationController`) et le **même** fidelity-api.

Dépôt	Variable	Doit pointer vers
`fidelity-api`	`CENTRAL_URL`	core-api (`by-plus-fidelity-develop`) — utilisé par `AuthenticateWithCentral` pour l'introspection
`core-api`	`FIDELITY_API_URL`	fidelity-api (domaine central, sans sous-domaine tenant)
`shop-core-api`	`FIDELITY_API_URL`	fidelity-api (même valeur que core-api)
`shop-core-api`	`REMOTE_LICENSE_API_URL`	core-api (`by-plus-fidelity-develop`) — utilisé par `FidelityService::isActivated()`

Le `.env.example` de shop-core-api a `REMOTE_LICENSE_API_URL=http://netcore.localhost` alors que
core-api utilise `http://core.localhost` dans son propre `.env.example` — **incohérence à vérifier dans le vrai `.env`**, pas juste supposer que l'exemple est correct.

0.4 Résolution des sous-domaines

Les 3 apps utilisent `{tenant}.{host}` (stancl/tenancy). Si en local, vérifier que le tenant de test (ex. `net`) résout bien sur les 3 domaines : `net.core.localhost` (ou équivalent), `net.fidelity...`, `net.shop-api.local`. Sinon les appels `Http::` échoueront en connexion, pas en 401/422 — plus difficile à diagnostiquer.

0.5 Queue worker actif

Les 2 jobs de sync (`SyncShopToFidelityJob`, `SyncShopUserToFidelityJob`) sont `ShouldQueue` — sans
worker (`php artisan queue:work` ou Horizon) tournant sur **shop-core-api**, ils ne s'exécuteront jamais et les logs "démarrage"/"succès" n'apparaîtront pas avant que tu ne les traites manuellement.

---

1. Suivre les logs pendant le test

Chaque dépôt écrit dans son propre `storage/logs/laravel.log` (ou équivalent configuré). Ouvrir un terminal par dépôt et suivre en direct :

```bash
tail -f storage/logs/laravel.log | grep -E "Fidelity|Company|Sale|ShopController::store|
ShopController::syncUser"
```

à lancer dans les 3 dépôts en parallèle pendant tout le scénario ci-dessous.

---

2. Scénario — parcours complet

Étape A — Activation (core-app / core-api)

1. Se connecter en admin sur le tenant de test.
2. Appeler `POST install/modules` avec le module `FIDELITY` dans `modules[]` (récupérer son UUID
   via `GET get/suit/apps` si besoin).
3. **Logs attendus (core-api)** :
   - `ModuleController::installModules — module FIDELITY installé, déclenchement activation`
   - `FidelityActivationService::activate — démarrage activation`
   - `FidelityActivationService::activate — activation réussie`
4. **Logs attendus (fidelity-api)** :
   - `CompanyController::storeCompany — company créée, dispatch CreateCompanyTenantJob`
   - (après traitement de la queue fidelity-api) `CreateCompanyTenantJob — provisioning du tenant fidelity-api démarré` puis `— tenant fidelity-api provisionné`
5. **Vérifier en base** :
   - core-api (centrale) : `tenant_information.fidelity_activated_at` non nul pour ce tenant.
   - fidelity-api (centrale) : nouvelle ligne dans `companies`, nouveau `tenants` avec un ID **égal** à l'ID du tenant core-api.
6. **Appel de contrôle** : `GET fidelity/activation-status` (core-api) → `{"activated": true}`.

Si l'activation échoue silencieusement (aucun log d'erreur mais `activated` reste `false`) : vérifier `CENTRAL_URL`/`FIDELITY_API_URL` (§0.3) en premier — c'est la cause la plus probable.

Étape B — Création boutique (shop-core-api)

1. `POST admin/store/shop` avec un nom, devise, fuseau horaire valides.
2. **Log attendu (shop-core-api)** : `SyncShopToFidelityJob — démarrage` puis `— boutique synchronisée avec succès` (après traitement par le worker, §0.5).
   - Si `isActivated()` retourne `false` à cet instant, le job n'est **jamais dispatché** — chercher `FidelityService::isActivated — vérification (non cachée)` avec `"activated": false` pour comprendre pourquoi (souvent : Étape A pas faite, ou mauvais `REMOTE_LICENSE_API_URL`).
3. **Log attendu (fidelity-api)** : `ShopController::store — boutique créée/retrouvée`.
4. **Vérifier en base (fidelity-api, tenant)** : `shops` a une ligne avec le **même ID** que la boutique shop-core-api.

Étape C — Affectation d'un caissier

1. `POST admin/add/user/in/shop` avec `shop_id` (celui de l'étape B) et un `users[]` (User existant côté shop-core-api).
2. **Log attendu (shop-core-api)** : `SyncShopUserToFidelityJob — démarrage` puis `— caissier synchronisé avec succès`.
3. **Log attendu (fidelity-api)** : `ShopController::syncUser — caissier synchronisé` avec `user_created` et `shop_user_created` à `true` (première fois) ou `false` (si rejoué).
4. **Vérifier en base (fidelity-api, tenant)** : `users` a une ligne avec le même ID que le User shop-core-api ; `shop_users` lie ce user à la boutique de l'étape B.

Étape D — Premier appel authentifié direct du caissier (provisioning lazy)

Optionnel si l'étape C a déjà tout synchronisé, mais utile pour vérifier le chemin "lazy" :

1. Avec le Bearer du caissier, appeler n'importe quelle route `central-auth` de fidelity-api (ex. `GET fidelity/customers/search?query=xxx` côté shop-core-api, qui relaie).
2. **Log attendu (fidelity-api)**, seulement si le `User` n'existait pas encore : `AuthenticateWithCentral — User provisionné automatiquement (premier contact)`.

Étape E — Recherche client

1. Créer un `Customer` de test côté fidelity-api (self-register, ou directement en base pour le test).
2. `GET fidelity/customers/search?query=<nom ou numéro>` (shop-core-api, avec le Bearer caissier).
3. **Log attendu (shop-core-api)** : `FidelityService — GET /get/customer/search` avec `status: 200`.
4. Vérifier que le client attendu est bien dans la réponse.

Étape F — Vente avec points (prepare → confirm → checkout)

1. Donner des points au client de test (directement en base `customer_points`, ou via le flux normal si déjà fonctionnel côté fidelity-api).

2. `POST fidelity/sales/prepare` (shop-core-api) avec `customer_id`, `total_amount`, `redeemed_points` > 0.
   - **Log attendu (shop-core-api)** : `FidelityService — POST /sales/prepare` (`status: 200`).
   - **Log attendu (fidelity-api)** : `SalesController::prepare — validation requise, en attente OTP/QR` (si des points/avantages sont utilisés) **ou** `— vente enregistrée directement` (si aucun avantage, cas hors scope de ce test).
   - Noter le `sale_validation_id` retourné.
3. `POST fidelity/sales/confirm` avec ce `sale_validation_id` et l'OTP (visible en base `sale_validations.otp_code` en environnement de test).
   - **Log attendu (fidelity-api)** : `SalesController::confirm — vente confirmée (OTP validé)`.
   - Noter le `sale.id` retourné dans la réponse — **c'est le `fidelity_sale_id` à passer au checkout final**.
4. `GET fidelity/sales/{sale_validation_id}/status` → doit renvoyer `confirmed: true`.
5. `POST pos/checkout` (le checkout normal, inchangé) en ajoutant `fidelity_sale_id` = l'ID de vente
   obtenu à l'étape 3.
   - **Log attendu (shop-core-api)** : `Pay::save — checkout rattaché à une vente fidélité`.
   - **Vérifier en base (shop-core-api, tenant)** : `shop_checkouts.fidelity_sale_id` = la valeur envoyée.

Étape G — Vérification de la faille corrigée (`sales/status`)

1. Répéter l'étape F.2-F.4 pour obtenir un `sale_validation_id` valide pour la **Boutique A**.
2. Se connecter comme un caissier d'une **Boutique B différente** (même tenant) et appeler `GET fidelity/sales/{sale_validation_id}/status` avec l'ID obtenu pour la Boutique A.
3. **Attendu : 404** (pas 200) — sinon la faille corrigée aujourd'hui a régressé.
4. **Log attendu (fidelity-api)** : `SalesController::status — accès refusé (boutique non propriétaire)`.

---

3. Points de vigilance connus (pas des bugs à chercher, déjà documentés)

- `searchCustomers` reste en recherche floue non scopée (décision utilisateur du 2026-08-19) — ne pas s'étonner qu'une recherche par prénom seul retourne des clients d'autres comptes BY+ en test multi-tenant.
- Pas de backfill rétroactif : seuls les boutiques/caissiers créés/affectés **après** l'activation sont synchronisés automatiquement.
- Secteur d'activité toujours `"Non renseigné"` après activation via `installModules()` (pas de formulaire) — normal, décision actée.