Spoke Integration Conventions

This document defines the conventions and database schema requirements for integrating restaurant spokes with the Innopay payment hub.

Database Schema Requirements

Required Tables

`dishes`

Column	Type	Description
`dish_id`	integer (PK)	Unique identifier for the dish
`name_fr`	text	French name
`name_en`	text	English name
`price`	decimal	Price in EUR
`category_id`	integer (FK)	Reference to categories
...	...	Other fields as needed

`drinks`

Column	Type	Description
`drink_id`	integer (PK)	Unique identifier for the drink
`name_fr`	text	French name
`name_en`	text	English name
`price`	decimal	Price in EUR
`category_id`	integer (FK)	Reference to categories
...	...	Other fields as needed

`categories`

Column	Type	Description
`category_id`	integer (PK)	Unique identifier
`type`	text	Either 'dishes' or 'drinks'
`name_fr`	text	French name
`name_en`	text	English name

`transfers` (for order tracking)

Column	Type	Description
`id`	bigint (PK)	Unique identifier
`from_account`	text	Sender's Hive account
`to_account`	text	Recipient's Hive account (merchant)
`amount`	text	Transfer amount
`symbol`	text	Token symbol (e.g., 'EURO')
`memo`	text	Raw memo from blockchain
`parsed_memo`	text	Dehydrated order data
`received_at`	timestamp	When transfer was received
`fulfilled`	boolean	Whether order has been fulfilled
`fulfilled_at`	timestamp	When order was fulfilled

> **Note**: The `to_account` column is populated by merchant-hub when processing transfers. This enables environment-aware filtering (DEV vs PROD orders).

Memo Dehydration/Hydration

Format

```
d:X,q:Y;b:Z,s:SIZE TABLE N suffix
```

Codes

Code	Meaning	Example
`d:X`	Dish with ID X	`d:1` = dish_id 1
`b:X`	Beverage/Drink with ID X	`b:3` = drink_id 3
`q:Y`	Quantity Y (only if > 1)	`q:2` = quantity 2
`s:SIZE`	Size/option	`s:50cl` = 50cl size
`TABLE N`	Table number	`TABLE 5` = table 5

Rules

1. **ALL dishes use** `d:` regardless of dish category (croques, desserts, appetizers, etc.)
2. **ALL drinks use** `b:` regardless of drink category (beers, wines, softs, cocktails, etc.)
3. Category information is NOT encoded in the memo - it's retrieved from database joins

4. Items are separated by `;`
5. Options within an item are separated by `,`
6. Quantity is only included if > 1
7. Table info comes after items, separated by space

Example

```
d:1,q:2;d:3;b:5,s:50cl TABLE 7 kcs-inno-abcd-1234
```

Means:

- 2x dish #1
- 1x dish #3
- 1x drink #5 (50cl size)
- Table 7
- Distriate suffix for tracking

Hydration (Backend)

To display order details, join with the appropriate tables:

```sql
-- For dishes
SELECT d.*, c.name_fr as category_name
FROM dishes d
JOIN categories c ON d.category_id = c.category_id
WHERE d.dish_id = X;

-- For drinks
SELECT dr.*, c.name_fr as category_name
FROM drinks dr
JOIN categories c ON dr.category_id = c.category_id
WHERE dr.drink_id = X;
```

Distriate Suffix Conventions

The distriate suffix is appended to the memo for tracking and cashback eligibility.

Format

```
{tag}-inno-{random4}-{random4}
```

Tags by Flow

Flow	Tag	Reason

Flow	Tag	Reason
Flow 3 (Guest Checkout)	`gst`	Internal invoice only, no blockchain account created
Flow 4 (Create Account Only)	`kcs`	Eligible for Distriator.com cashback
Flow 5 (Create Account + Pay)	`kcs`	Eligible for Distriator.com cashback
Flow 6 (Pay with Account)	`kcs`	Eligible for Distriator.com cashback
Flow 7 (Topup + Pay)	`kcs`	Eligible for Distriator.com cashback
Call Waiter	`cqb` (or restaurant tag)	Restaurant-specific tracking

Why Different Tags?

- `gst` **(guest)**: Guest checkout doesn't create a blockchain account, so no HBD moves from customer to Innopay. The suffix is purely for internal invoice generation.
- `kcs`: All flows involving a blockchain account are eligible for Distriator.com cashback rewards. Using a consistent tag allows Distriator to track qualifying transactions.

Frontend Menu Data (menu-data.ts)

Item ID Convention

Item IDs in the frontend should ideally match database IDs:

- `dish-{dish_id}` for dishes (e.g., `dish-1`, `dish-2`)
- `drink-{drink_id}` for drinks (e.g., `drink-1`, `drink-2`)

Category Field

The `category` field determines dish vs drink:

- **Dish categories**: Any food items (croques, desserts, appetizers, mains, etc.)
- **Drink categories**: Any beverages (beers, wines, softs, cocktails, etc.)

Example Structure

```
interface MenuItem {
  id: string;            // 'dish-1' or 'drink-1'
  category: string;      // Category name for grouping
  name: { fr: string; en: string; };
  price: number;
  // ... other fields
}
```

Environment Configuration

Architecture Overview

Environment detection is **centralized** in `src/lib/environment.ts`. This single source of truth provides:

- Environment type (`PROD` or `DEV`)
- All environment-specific attributes (URLs, account names, etc.)

Why centralized? Previously, environment detection was scattered across multiple files (`getInnopayUrl()`, `getHiveAccount()`, etc.). This led to:

- Duplicated detection logic
- Inconsistent behavior if detection rules diverged
- Difficulty adding new environment-specific attributes

The refactored approach consolidates everything into one module with a single `isPrivateNetwork()` helper and one `EnvironmentConfig` interface.

EnvironmentConfig Interface

```
interface EnvironmentConfig {
  environment: 'PROD' | 'DEV';
  toAccount: string;        // Merchant's Hive account for this environment
  innopayUrl: string;       // Wallet URL for payment flows
}
```

Attribute	DEV Value	PROD Value
`environment`	`'DEV'`	`'PROD'`
`toAccount`	`'croque-test'`	`'croque.bedaine'`
`innopayUrl`	`http://192.168.178.55:3000`	`https://wallet.innopay.lu`

Detection Logic

```
function isPrivateNetwork(): boolean {
  const hostname = window.location.hostname;
  return (
    hostname === 'localhost' ||
    hostname === '127.0.0.1' ||
    hostname.startsWith('192.168.') ||
    hostname.startsWith('10.') ||
    hostname.includes('vercel.app')  // Preview deployments
  );
}
```

```
// PROD only when on the exact production domain
const isProd = window.location.hostname === 'croque-bedaine.innopay.lu';
```

Usage

React Components (hook):

```
import { useEnvironmentConfig } from '@/lib/environment';

function MyComponent() {
  const { environment, toAccount, innopayUrl } = useEnvironmentConfig();
  // Filter data by environment, display badges, etc.
}
```

Non-React Code (direct function):

```
import { getEnvironmentConfig, isPrivateNetwork } from '@/lib/environment';

const config = getEnvironmentConfig();
console.log(config.toAccount);  // 'croque-test' or 'croque.bedaine'
```

Adding New Environment Attributes

To add a new environment-specific value:

1. Add the attribute to `EnvironmentConfig` interface
2. Add values to both `DEV_CONFIG` and `PROD_CONFIG` objects
3. All consumers automatically get the new attribute

```
// Example: adding merchantHubUrl
interface EnvironmentConfig {
  environment: 'PROD' | 'DEV';
  toAccount: string;
  innopayUrl: string;
}

const DEV_CONFIG: EnvironmentConfig = {
  environment: 'DEV',
  toAccount: 'croque-test',
  innopayUrl: 'http://192.168.178.55:3000',
};
```

> **Note**: `merchantHubUrl` is NOT part of `EnvironmentConfig` because merchant-hub has a single static URL (`https://merchant-hub.innopay.lu`) used in both DEV and PROD environments.

Legacy Environment Variables (Deprecated)

These were previously used but are now superseded by the centralized config:

Variable	Status
`VITE_INNOPAY_URL`	Deprecated - use `config.innopayUrl`
`VITE_HIVE_ACCOUNT`	Deprecated - use `config.toAccount`
`VITE_MERCHANT_HUB_URL`	To be added to EnvironmentConfig
`VITE_SHOP_ID`	Still used (not environment-dependent)

Payment Flows Summary

Flow	Description	Requires Account	Creates Account	Cashback Eligible
3	Guest Checkout	No	No	No
4	Create Account Only	No	Yes	No
5	Create Account + Pay	No	Yes	Yes
6	Pay with Existing Account	Yes	No	Yes
7	Topup + Pay	Yes	No	Yes

---

localStorage Keys

All Innopay-related localStorage keys use the `innopay_` prefix:

- `innopay_accountName` - User's Hive account name
- `innopay_masterPassword` - Master password
- `innopay_activePrivate` - Active private key
- `innopay_postingPrivate` - Posting private key
- `innopay_table` - Current table number (from QR code)

Current Orders (CO) Page

The Current Orders page displays unfulfilled orders to restaurant staff. It syncs with merchant-hub to receive blockchain transfers in real-time.

Page Location

- **Next.js**: `/app/admin/current_orders/page.tsx`
- **Vite/React**: `/src/pages/admin/CurrentOrders.tsx`
- **URL**: `/admin/current_orders` or `/admin/current-orders`

Merchant-Hub Integration Flow

```
┌─────────────┐       ┌──────────────┐       ┌─────────────┐
│   CO Page   │  ──   │ merchant-hub │  ──   │    Redis    │
│   (spoke)   │       │    (hub)     │       │   Streams   │
└─────────────┘       └──────────────┘       └─────────────┘
      │                      │                      │
      │  1. POST /api/wake-up                       │
      │ ──────────────────── │                      │
      │  (register as poller or consumer)           │
      │                      │                      │
      │  2. GET /api/poll (if poller)               │
      │ ──────────────────── │  HAF query           │
      │                      │ ──────────────────   │
      │                      │  XADD transfers      │
      │                      │ ──────────────────── │
      │                      │                      │
      │  3. GET /api/transfers/consume              │
      │ ──────────────────── │  XREADGROUP          │
      │                      │ ──────────────────── │
      │  (returns transfers) │                      │
      │                      │                      │
      │  4. POST /api/transfers/ack                 │
      │ ──────────────────── │  XACK                │
      │  (after DB insert)   │ ──────────────────── │
      │                      │                      │
```

Consumer ID Convention

Consumer IDs MUST be **stable across page refreshes** to receive pending messages:

```
function getStableConsumerId(): string {
  const storageKey = '{restaurant}_co_consumer_id';
  let consumerId = localStorage.getItem(storageKey);
  if (!consumerId) {
    consumerId = `{restaurant}-co-${Math.random().toString(36).slice(2, 8)}`;
    localStorage.setItem(storageKey, consumerId);
  }
  return consumerId;
}
```

Merchant-Hub URL

The merchant-hub has a single static URL used in both DEV and PROD environments:

```
// src/lib/constants.ts
export const MERCHANT_HUB_URL = 'https://merchant-hub.innopay.lu';
// Alias: https://merchant-hub-theta.vercel.app
```

Required API Calls

All endpoints are relative to `MERCHANT_HUB_URL`:

Endpoint	Method	Purpose	Params
`/api/wake-up`	POST	Register shop, elect poller	`{ shopId }`
`/api/poll`	GET	Trigger HAF blockchain query	(none)
`/api/transfers/consume`	GET	Get pending transfers	`restaurantId`, `consumerId`, `count`
`/api/transfers/ack`	POST	Acknowledge processed transfers	`{ restaurantId, messageIds }`

Example Usage

```
import { MERCHANT_HUB_URL } from '@/lib/constants';

// Wake-up call
await fetch(`${MERCHANT_HUB_URL}/api/wake-up`, {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ shopId: 'croque-bedaine' }),
});

// Consume transfers
const response = await fetch(
  `${MERCHANT_HUB_URL}/api/transfers/consume?restaurantId=croque-
bedaine&consumerId=${consumerId}&count=100`
);
```

Polling Intervals

- **Poll interval**: 6 seconds (HAF query, if elected poller)
- **Sync interval**: 6 seconds (consume from Redis, all pages)

Transfer Payload Structure

The merchant-hub returns transfer objects with these fields:

Field	Type	Description
`messageId`	string	Redis stream message ID (for ACK)
`id`	string	Blockchain transfer ID (unique)
`from_account`	string	Sender's Hive account
`to_account`	string	Recipient's Hive account (merchant)

Field	Type	Description
`amount`	string	Transfer amount
`symbol`	string	Token symbol (e.g., "EURO")
`memo`	string	Raw memo (dehydrated order)
`parsed_memo`	string	Human-readable memo
`received_at`	string	ISO timestamp

Important: The `to_account` field MUST be included in the payload from merchant-hub and stored in the `transfers` table. This enables multi-environment filtering (DEV vs PROD) since each environment uses a different merchant account.

Data Flow (Client-Side Spoke)

For Vite/React spokes that don't have server-side APIs:

1. **Wake-up**: Register with merchant-hub
2. **Poll** (if poller): Trigger HAF query every 6s
3. **Consume**: Fetch transfers from Redis stream
4. **Insert**: Upsert into local Supabase `transfers` table (including `to_account`)
5. **ACK**: Acknowledge to merchant-hub (removes from Redis pending)
6. **Display**: Fetch unfulfilled orders from local DB, filtered by `to_account`

Data Flow (Server-Side Spoke)

For Next.js spokes with server-side APIs:

1. **Wake-up**: Register with merchant-hub (client-side)
2. **Poll** (if poller): Trigger HAF query (client-side)
3. **Sync**: Call local `/api/transfers/sync-from-merchant-hub` (server-side handles consume + insert + ACK)
4. **Display**: Fetch unfulfilled from local `/api/transfers/unfulfilled`

UI Conventions

Order Card Display

- **Dishes**: Dark red text (`text-red-800` / `#8B0000`), bold
- **Drinks**: Green text (`text-green-700` / `#008000`), bold
- **Separator**: Dashed line between dishes and drinks (if both present)
- **Quantity**: Gray, right-aligned before item name

Order Metadata

- **Table**: From memo parsing (`TABLE X`)
- **Client**: `@{from_account}`
- **Amount**: `{amount} {symbol}`

- **Time**: Color-coded based on age
    - Green: < 10 minutes old
    - Red: > 10 minutes old (late order)

Action Buttons

Scenario	Button 1	Button 2
Order has dishes, not transmitted	Orange "Transmettre en cuisine"	Gray "C'est parti!" (disabled)
Order has dishes, transmitted	(none)	Blue "C'est parti!"
Order is drinks-only	(none)	Blue "C'est parti!"

Kitchen Transmission

- Local state only (not persisted to DB)
- Shows timestamp: "Transmis en cuisine à HH:MM"
- Stored in `Map<orderId, timestamp>`

Call Waiter Orders

- Red border, pulsing animation
- Red background (`bg-red-50`)
- Detected by: `memo.toLowerCase().includes('appel')`

Memo Hydration

The CO page must hydrate dehydrated memos for display. A memo consists of **three semantic units**:

1. **Order Content** (required): The items ordered using `d:` and `b:` codes
2. **Table Information** (optional): `TABLE N` indicating which table placed the order
3. **Distriate Suffix** (required for detection): Contains the `-inno-` substring for tracking

```
// Input: "d:3;b:5,s:50cl TABLE 7 kcs-inno-abcd-1234"
// Parsed as:
//   - Order content: "d:3;b:5,s:50cl"
//   - Table info: "TABLE 7" → table = 7
//   - Distriate suffix: "kcs-inno-abcd-1234"
// Output: [
//   { type: 'item', quantity: 1, description: 'Croque Sa Mer', categoryType: 
'dish' },
//   { type: 'separator' },
//   { type: 'item', quantity: 1, description: 'Leffe (50cl)', categoryType: 
'drink' }
// ]
```

> ⚠ **CRITICAL CAVEAT**: Orders are detected by the presence of the `-inno-` substring in the distriate suffix. If an order memo is missing this suffix (e.g., due to a bug in the payment flow), **the order will**

> **NOT be picked up by the detection system**. Always ensure the distriate suffix is appended to all order memos!

Hydration requires fetching menu data from the database:

- `dishes` table: `dish_id` → `name`
- `drinks` table: `drink_id` → `name`

localStorage Keys (CO-specific)

Key	Purpose
`{restaurant}_co_consumer_id`	Stable consumer ID for Redis
`innopay_table`	Current table (shared with customer menu)

Order Stacking

Orders are displayed oldest-at-bottom, newest-at-top:

```
<div className="flex flex-col-reverse gap-4">
  {orders.map(order => <OrderCard key={order.id} />)}
</div>
```

Version History

Version	Date	Changes
1.0	2026-01	Initial conventions document
1.1	2026-01	Added Current Orders (CO) page conventions
1.2	2026-01	Added `to_account` column to transfers table; centralized environment config in `src/lib/environment.ts` with full architecture documentation; documented memo's 3 semantic units (order content, table info, distriate suffix) and `-inno-` detection caveat