Sistema Inteligente de Gestión de Inventario

Documentación (v0.3)

24 de noviembre de 2025

Índice

1. Arquitectura General del Sistema

El proyecto **Smart Inventory** está diseñado como un sistema modular de gestión de inventario que combina:

- Un **frontend simple** basado en HTML, TailwindCSS y JavaScript.
- Una **API REST robusta** implementada con Django + Django REST Framework.
- Un **modelo de datos** estable y orientado a trazabilidad mediante lotes.
- Un módulo futuro de **aprendizaje automático** para predicciones de uso.

El flujo de trabajo completo es:

Frontend (QR/Formulario) ⟶ API REST (/api/scan/) ⟶ Base de Datos (SQLite/PostgreSQL) ⟶ ML (futuro)

El sistema está pensado para ser desplegado tanto en entornos domésticos como en pequeñas oficinas, con la posibilidad de escalar a infraestructuras más complejas si se utilizan bases de datos y servidores más potentes.

2. Frontend

2.1. Navegación y diseño (Fase 3)

En la Fase 3 se ha consolidado la interfaz de usuario para que el sistema sea utilizable tanto en escritorio como en móvil, manteniendo una apariencia limpia y coherente en todas las pantallas.

Navbar global Se ha definido una barra de navegación común que aparece en las vistas principales del sistema. Sus características clave son:

- Branding compacto con el identificador **Smart Inventory** y el acrónimo **SI**.
- Uso de colores basados en una paleta neutra con acentos en `teal` para estados activos y elementos importantes.
- Diseño responsivo: la navbar se adapta a pantallas pequeñas manteniendo accesibles las acciones esenciales (escaneo, gestión de ubicaciones, vistas de lista, etc.).

Lenguaje visual El frontend utiliza **TailwindCSS** como sistema de utilidades para construir la interfaz y **Alpine.js** para la interacción ligera en cliente. Se han aplicado los siguientes criterios:

- Tarjetas con bordes redondeados, sombras suaves y separación clara entre secciones.
- Tipografía consistente, con tamaños diferenciados para títulos, subtítulos y texto de apoyo.
- Estados *hover* y pequeñas transiciones para hacer más legible la interacción sin sobrecargar el sistema.

Home consolidada La página de inicio presenta ahora:

- Un bloque de **branding** donde se identifica el sistema y su propósito (control de stock en casa y oficina).
- Accesos rápidos a las funciones principales: escaneo, lista de productos y gestor de ubicaciones.
- Estructura simétrica y centrada para facilitar el uso en móvil.

Gestor de ubicaciones El *Location Manager* se ha unificado visualmente con el resto de la aplicación:

- Contenedor principal con borde, fondo blanco y separación respecto al fondo general.
- Listado de ubicaciones con jerarquía visual clara, respetando la estructura recursiva (`armario 1 >caja 2 >fondo 1`, etc.).
- Botones y enlaces alineados con el estilo global de la aplicación.

Pantalla de escaneo Aunque la lógica de negocio se describe en secciones posteriores, a nivel de diseño se han separado claramente las acciones posibles:

- **Entrada**: muestra un formulario completo para crear o ampliar productos, incluyendo categoría, unidad, ubicación, stock mínimo y fechas relevantes.
- **Salida**: prioriza el escaneo de códigos QR o la búsqueda rápida de productos existentes, marcando consumos y aperturas de envase.
- **Auditoría**: se mantiene accesible pero secundaria, para no confundirla con los flujos de entrada/salida habituales.

Esta Fase 3 deja la base visual y de UX lista para futuras extensiones, como los módulos de analítica avanzada y de predicción de consumo.

2.2. Interfaz de escaneo

La interfaz principal del sistema está en `scan.html`. Permite ejecutar 4 tipos de movimientos:

- Entrada de stock (IN)
- Salida de stock (OUT)
- Auditoría parcial (AUD)
- Auditoría total (AUDTOTAL)

El frontend envía un JSON de este estilo:

```
{
  "payload": "PRD:<uuid>",          // opcional en IN; obligatorio en OUT
  "movement_type": "IN",
  "quantity": 2,
  "location": "<uuid_de_location>",
  "new_product": {
    "name": "Leche entera",
    "unit": "l",
    "category": "lacteos"
  },
  "mark_open": false,
  "open_days": null
}
```

La ubicación enviada en el JSON es siempre el **UUID público** de la Location, no la ruta completa.

3. Modelo de Datos Relevante

Los modelos implicados en la lógica del endpoint `/api/scan/` son:

3.1. Product

- `id` (UUID interno)
- `name`, `category`, `unit`

- `location`: ubicación principal del producto
- `min_stock`
- `qr_payload`: identificador permanente `PRD:<uuid>`

3.2. Batch (Lote)

- `product` (FK)
- `quantity` (stock del lote)
- `expiration_date`
- `entry_date`
- `opened_units` (0 o 1)
- `opened_at`
- `open_expires_at`

3.3. Location

- `id` (UUID)
- `name`
- `parent` (FK a Location, permite estructura recursiva)
- `tenant`

3.4. Movement

- `id`
- `product` (FK)
- `batch` (FK)
- `movement_type` (`IN`, `OUT`, `AUD`, `AUDTOTAL`)
- `quantity`
- `location`
- `created_at`

4. Lógica del Endpoint /api/scan/

El endpoint `/api/scan/` recibe siempre una carga JSON con un `movement_type` y decide qué handler interno ejecutar.

- **_handle_in**: Entrada de stock
- **_handle_out**: Salida de stock
- **_handle_aud**: Auditoría de ubicación
- **_handle_audtotal**: Auditoría global

Antes de delegar en un handler, la API ejecuta una fase de validación común:

- Validación de `movement_type`
- Validación de `quantity >0` (IN/OUT)
- Validación de `payload` y PRD
- Validación de `location`
- Validación de `mark_open` y `open_days`

5. Movimiento IN (Entrada)

El movimiento `IN` se encarga de registrar nuevas unidades de stock. La lógica simplificada es:

1. Se requiere obligatoriamente:
   - `name`
   - `unit`
   - `location` válida
2. Se busca un `Product` que coincida en:
   - nombre normalizado,
   - ubicación,
   - tenant.
3. Si existe, se reutiliza; si no existe, se crea.
4. Se crea un lote nuevo asociado al producto.
5. Se registra un `Movement IN`.
6. Se devuelve o asigna el `PRD:<uuid>` permanente.

6. Movimiento OUT (Salida)

El movimiento `OUT` es el más complejo del sistema porque incluye:

1. Consumo de lotes abiertos.
2. Marcaje de apertura de envase (`mark_open`).
3. Consumo FIFO estándar.
4. Comprobaciones de stock mínimo y errores de concurrencia.

6.1. Casos principales

Consumo normal (sin `mark_open`)

- Si hay lotes abiertos, se consumen primero.
- Después se consumen lotes cerrados en orden FIFO por fecha de entrada.

Marcaje de apertura (`mark_open = true`)

- Si no hay lote abierto, se marca uno como abierto.
- Se fija `opened_at` y `open_expires_at` según `open_days`.

Errores típicos

- Intentar abrir un lote cuando ya hay uno abierto.
- Consumir más unidades de las disponibles.
- Mezclar apertura y consumo en el mismo movimiento de forma inválida.

7. Auditoría Parcial (AUD)

La auditoría parcial permite revisar el stock de una ubicación concreta.

- Se pasa una `location` concreta.
- El sistema devuelve:
  - productos en esa ubicación,
  - lotes y cantidades,
  - caducidades.

8. Auditoría Total (AUDTOTAL)

Recorre todas las ubicaciones del tenant y devuelve una estructura:

- ubicación
- productos dentro
- totales de unidades
- caducidad mínima por producto

9. Reglas de Consistencia y Errores

Para mantener la base de datos coherente, el sistema aplica un conjunto de reglas y códigos de error bien definidos.

9.1. Validaciones generales

- **movement_type**: debe ser uno de {`IN`, `OUT`, `AUD`, `AUDTOTAL`}.
- **quantity**: debe ser mayor que 0 en IN/OUT.
- **location**: debe existir y pertenecer al mismo tenant.
- **payload**: si se usa, debe ser un `PRD:<uuid>` válido.

9.2. Errores típicos

- `invalid_movement_type`
- `invalid_quantity`
- `invalid_mark_open`
- `invalid_open_days`
- `insufficient_stock`
- `concurrency_race`
- `location_required`

Estas reglas aseguran que todas las operaciones de escaneo se reflejen de manera consistente en la base de datos y que sea posible auditar todo el historial del sistema de forma fiable.