Sistema Inteligente de Gestión de Inventario

Documentación (v0.1)

22 de noviembre de 2025

Índice

1. Arquitectura del Sistema

El proyecto **Smart Inventory** se estructura en torno a una arquitectura modular que combina un frontend ligero, un backend robusto basado en *Django* y una capa de análisis de datos e inteligencia artificial construida sobre *Python* y librerías de aprendizaje automático. El objetivo es ofrecer un sistema de gestión de inventario flexible, escalable y capaz de incorporar predicciones inteligentes a partir del comportamiento histórico de uso.

1.1. Frontend: Interfaz de Usuario

El frontend actúa como la puerta de entrada al sistema. Está desarrollado en HTML y CSS con *TailwindCSS*, junto con *JavaScript* nativo para manejar la interacción con el usuario y las peticiones al servidor.

La página principal (`scan.html`) ofrece un formulario dinámico y un lector de códigos QR integrado mediante la librería *Html5Qrcode*. El usuario puede registrar movimientos de inventario (entradas, salidas, auditorías parciales o totales) ya sea escaneando un código o introduciendo la información manualmente.

Cada operación genera una petición *POST* en formato JSON, que es enviada al backend a través de la API REST:

```
{
  "payload": "PRD:UUID",
```

```
  "movement_type": "IN",
  "quantity": 2,
  "location": "Cocina > Armario 1 > Estante 3",
  "new_product": {
    "name": "Leche entera",
    "category": "Lácteos",
    "unit": "Litros",
    "min_stock": 1,
    "expiration_date": "2025-11-01"
  }
}
```

1.2. Backend: Django + Django REST Framework

El backend constituye el núcleo lógico del sistema. Se encarga de validar los datos, aplicar las reglas de negocio y mantener la coherencia de la base de datos. Está implementado con el framework *Django* y su extensión *Django REST Framework* (DRF), lo que permite gestionar fácilmente los endpoints de la API y los modelos de datos.

El flujo de trabajo interno es el siguiente:

1. El frontend envía una solicitud JSON a la API REST.
2. Django verifica el tipo de movimiento (`IN`, `OUT`, `AUD`, `AUDTOTAL`).
3. Se aplican las reglas correspondientes:
   - **IN:** Crea o acumula un producto existente, registrando un nuevo lote si es necesario.
   - **OUT:** Resta unidades del stock, sin permitir valores negativos.
   - **AUD:** Devuelve el inventario de una ubicación específica.
   - **AUDTOTAL:** Recorre todas las ubicaciones anidadas y genera un resumen global.
4. Django ORM traduce las operaciones a consultas SQL y actualiza las tablas correspondientes.
5. El sistema responde al cliente con un JSON estructurado indicando el resultado de la operación.

1.3. Base de Datos

El almacenamiento de la información se realiza mediante un sistema relacional (SQLite en desarrollo y PostgreSQL en producción). Las entidades principales son:

- **Product:** información básica del producto y stock mínimo.
- **Batch:** registro de lotes con fecha de entrada y caducidad.
- **Location:** ubicaciones jerárquicas (ubicaciones dentro de ubicaciones).
- **Movement:** historial de operaciones (entradas, salidas y auditorías).

Cada producto se vincula a una ubicación específica y puede tener múltiples lotes. Cada movimiento modifica las cantidades de los lotes o genera nuevas entradas.

1.4. Módulo de Aprendizaje Automático

El componente de inteligencia artificial utiliza los datos históricos del inventario para detectar patrones y generar alertas predictivas. Mediante librerías como *pandas*, *NumPy* y *scikit-learn*, se entrenan modelos para:

- Estimar el consumo promedio por producto o categoría.
- Sugerir reposiciones antes de alcanzar el stock mínimo.
- Detectar anomalías o patrones de uso inusuales.
- Calcular probabilidades de caducidad temprana.

El modelo produce resultados almacenados en una tabla de recomendaciones, que el backend expone a través de un endpoint `/api/recommendations/`. Posteriormente, estos datos se visualizan en el frontend mediante un dashboard interactivo.

1.5. Integración de Componentes

El sistema funciona como un flujo continuo de datos:

Frontend (QR/Formulario) → API REST (Django + DRF) → Base de Datos (SQL) → Módulo ML (Predicciones) → Dashboard Visual

Este diseño modular permite añadir nuevas funciones o fuentes de datos sin afectar la estabilidad del sistema principal, garantizando escalabilidad y mantenibilidad a largo plazo.

2. Lógica de negocio del inventario

En esta sección se describe con detalle el comportamiento real del endpoint `/api/scan/`, que constituye el núcleo de la lógica de negocio del sistema. Dicho endpoint recibe una petición JSON desde el frontend y, en función del `movement_type`, aplica reglas específicas para registrar entradas, salidas y auditorías de inventario, siempre garantizando la consistencia de los datos a nivel de lotes.

2.1. Modelo de datos relevante

Aunque los modelos se describen de forma general en la Sección anterior, aquí se resumen los campos que intervienen directamente en la lógica de *scan*:

- **Product**:
  - Identificador interno (`id`) y campos descriptivos (`name`, `category`, `unit`, etc.).
  - Stock mínimo recomendado (`min_stock`).
  - Ubicación principal asociada (`location`).
- **Batch** (lote de producto):
  - `product`: referencia al `Product` al que pertenece.
  - `quantity`: cantidad disponible en ese lote (siempre ≥ 0).
  - `entry_date`: fecha de entrada del lote.
  - `expiration_date`: fecha límite de consumo en estado cerrado.
  - `opened_units`: número de unidades actualmente abiertas (normalmente 0 o 1).

- `opened_at`: marca temporal de cuándo se abrió la unidad.
- `open_expires_at`: fecha/hora de caducidad una vez abierto.
- `is_depleted`: bandera booleana que indica si el lote está agotado.
- `depleted_at`: fecha/hora en la que el lote quedó a cero.

- **Location**:
  - Identificador y relación jerárquica con otras ubicaciones (`parent`).
  - Campo derivado `full_path()` que devuelve la ruta completa (por ejemplo `Cocina >Despensa >Balda 1`).
- **Movement**:
  - `product`: producto afectado.
  - `location`: ubicación del movimiento.
  - `quantity`: cantidad con signo (+ entrada, - salida).
  - `movement_type`: uno de `IN`, `OUT`, `ADJ`, `AUD`, `AUDTOTAL`.
  - `metadata`: campo JSON para información adicional (por ejemplo, lotes consumidos en una salida).

2.2. Entrada de stock (`IN`)

El movimiento de tipo **IN** registra la entrada de producto en el sistema y la creación de uno o varios lotes. El cuerpo típico de la petición es:

```
{
  "payload": "PRD:<uuid>",        // opcional
  "movement_type": "IN",
  "quantity": 5,
  "location": "<uuid_de_location>",
  "new_product": {
    "name": "Tomate en lata",
    "category": "Conservas",
    "unit": "lata",
    "min_stock": 2,
    "expiration_date": "2028-06-04",
    "notes": "Pack de oferta"
  }
}
```

Existen dos escenarios principales:

Caso A: Entrada con `PRD:<uuid>` válido

1. El backend extrae el UUID del `payload` y busca un `Product` existente. Si no se encuentra, se devuelve error 404.
2. Se valida la `location` (UUID de `Location`); si no existe o no pertenece al *tenant* actual, se devuelve error 400/404.
3. Se crea un nuevo `Batch` asociado al producto y a la ubicación:
   - `quantity` se inicializa con la `quantity` recibida.

- `expiration_date` se toma de `new_product.expiration_date` si se proporciona; en caso contrario puede quedar en blanco.
- El lote comienza con `opened_units = 0` e `is_depleted = false`.

4. Se registra un `Movement` de tipo `IN` con cantidad positiva.
5. La respuesta JSON incluye un mensaje de confirmación y, opcionalmente, un `payload` QR actualizado para el producto.

Caso B: Entrada sin `PRD` (alta de producto nuevo)

Si el `payload` no contiene un `PRD:<uuid>` válido, el sistema interpreta la operación como alta de un nuevo producto:

1. Se valida que `new_product.name` y `new_product.unit` no estén vacíos. Si faltan, se devuelve error 400.
2. Se crea un nuevo `Product` usando los campos de `new_product` (nombre, categoría, unidad, stock mínimo, notas, etc.), asociado a la `location` indicada.
3. Se crea el lote inicial (`Batch`) con la `quantity` recibida y la fecha de caducidad opcional.
4. Se registra un `Movement` de tipo `IN`.
5. La respuesta incluye un `payload` QR del tipo `PRD:<uuid>` que identifica de forma permanente al nuevo producto.

En ambos casos, la lógica garantiza que el stock total de un producto es la suma de las cantidades de todos sus lotes activos (`quantity` ≥ 0 y `is_depleted = false`).

2.3. Salida de stock (`OUT`) con consumo FIFO

El movimiento de tipo **OUT** descuenta unidades del stock de un producto. La salida normal (sin marcar como abierto) se caracteriza por:

- Requerir un `payload` del tipo `PRD:<uuid>` válido.
- Aplicar una estrategia FIFO basada en la fecha de caducidad y la fecha de entrada del lote.

El algoritmo simplificado es:

1. Identificar el `Product` a partir del UUID del `payload`; si no existe, error 404.
2. Calcular el stock total disponible sumando `quantity` de todos los lotes del producto que:
   - No estén agotados (`is_depleted = false`).
   - Tengan cantidad positiva (`quantity >0`).
3. Si la cantidad solicitada (`need`) es mayor que el stock disponible, se devuelve error 400 con:
   - `error = ïnsufficient_stock"`
   - `available`: stock disponible.
   - `requested`: cantidad solicitada.
4. Se inicia una transacción atómica y se bloquean los lotes con `select_for_update()`, ordenándolos por:

   a) `expiration_date ASC` (nulos al final),
   b) `entry_date ASC`,
   c) `id ASC`.

5. Se recorre la lista de lotes en ese orden y se va descontando `take = min(quantity_lote, remaining)` en cada uno, actualizando:
   - `quantity` del lote.
   - `is_depleted` y `depleted_at` si la cantidad llega a 0.
6. Para cada lote afectado se almacena en una lista de trazabilidad:

```
{
  "batch_id": <id>,
  "prev_qty": cantidad_antes,
  "taken": unidades_extraidas,
  "new_qty": cantidad_despues,
  "expiration_date": "AAAA-MM-DD"
}
```

7. Si al final de la transacción queda `remaining >0`, se considera que ha habido una condición de carrera y se devuelve error 409 (`concurrency_race`).
8. En caso de éxito, se crea un `Movement` de tipo `OUT` con cantidad negativa e incluyendo en `metadata` la lista de lotes consumidos.
9. La respuesta JSON incluye:
   - `ok = true`
   - Datos básicos del producto y de la ubicación.
   - `requested`, `stock_remaining`.
   - `consumed_batches`: lista de lotes afectados.

2.4. Marcaje de unidades abiertas (`OUT` con `mark_open`)

Además del consumo de stock, el endpoint soporta una operación especial para marcar una unidad como "abierta" sin descontar cantidad. Esta operación se utiliza para envases que, una vez abiertos, tienen una vida útil limitada.

El frontend envía, junto al `movement_type = "OUT"`, dos campos adicionales:

- `mark_open` (booleano): indica que la intención es abrir un lote.
- `open_days` (entero opcional): número de días de vida útil tras abrir.

La lógica aplicada es:

1. Se localiza el producto a partir del `payload` (`PRD:<uuid>`).
2. Se selecciona el lote con stock disponible más cercano a caducar: primer `Batch` con `quantity >0` e `is_depleted = false`, ordenado por `expiration_date` y `entry_date`.
3. Si no existen lotes con stock, se devuelve error 400 (`"No hay stock para abrir."`).

4. Si el lote ya tiene `opened_units >0`, se rechaza la operación con error 400 para evitar tener varias aperturas simultáneas sobre el mismo lote.

5. Se marca el lote como abierto:

   - `opened_units = 1`
   - `opened_at = now()`
   - `open_expires_at = now() + open_days` si se proporcionó `open_days`; en caso contrario puede quedar en nulo.

6. La operación de `mark_open` no descuenta stock ni crea necesariamente un `Movement` de consumo; su objetivo es guardar el estado "abierto hasta fecha X" para ese lote.

7. La respuesta confirma la apertura e incluye el identificador del lote y la fecha de caducidad tras abrir.

2.5. Auditoría de ubicación (`AUD`)

La auditoría parcial permite inspeccionar el estado del inventario en una ubicación concreta. El cliente envía:

```
{
  "movement_type": "AUD",
  "location": "<uuid_de_location>"
}
```

La API realiza:

1. Validación de la ubicación; si no existe, error 404.

2. Obtención de todos los `Product` asignados a esa ubicación (posiblemente agrupando por árbol de ubicaciones si se usa jerarquía).

3. Para cada producto:

   a) Se recuperan los lotes con sus campos clave: `id`, `quantity`, `expiration_date`, `opened_units`, `open_expires_at`.

   b) Se filtran los lotes con cantidad > 0.

   c) Se calcula la cantidad total (`total_quantity` = suma de `quantity`).

   d) Se determina la `nearest_expiration` como la fecha mínima de caducidad entre los lotes no vacíos.

4. La respuesta incluye, para esa ubicación:

   - `location`: ruta completa de la ubicación.
   - `total_products`: número de productos distintos.
   - `items`: lista de productos con sus totales y lotes.

El frontend utiliza esta información para renderizar una tabla en la que, para cada producto, se muestran:

- Cantidad total.
- Caducidad más próxima.
- Detalle de lotes (incluyendo si alguno está marcado como abierto y hasta cuándo es válido).

2.6. Auditoría total (AUDTOTAL)

La auditoría total recorre *todas* las ubicaciones registradas y genera un resumen global del inventario.

1. Se obtienen todas las `Location` del *tenant*, ordenadas por nombre o ruta.
2. Para cada ubicación se recopilan los productos y se aplica la misma lógica que en la auditoría parcial:
   - Cálculo de `total_quantity` por producto.
   - Cálculo de `nearest_expiration`.
   - Recopilación de lotes con posible indicador de apertura (`opened_units`, `open_expires_at`).
3. Se construye una estructura de respuesta del tipo:

```
{
  "ok": true,
  "total_locations": N,
  "inventory": [
    {
      "location": "Cocina > Balda",
      "total_products": 1,
      "items": [ ... ]
    },
    {
      "location": "Cocina > Despensa",
      "total_products": 2,
      "items": [ ... ]
    }
  ]
}
```

4. El frontend muestra esta información agrupada por ubicación, con tablas análogas a las de `AUD`, permitiendo una vista global del inventario y de los lotes cercanos a caducar.

2.7. Reglas de consistencia y manejo de errores

Para mantener la integridad del sistema se aplican varias reglas:

- **Stock no negativo:** el campo `Batch.quantity` está protegido por una restricción de base de datos que asegura `quantity` $\geq$ `0`. La lógica de negocio nunca descuenta más unidades de las disponibles.
- **Lotes agotados:** cuando un lote llega a cantidad 0 se marca `is_depleted = true` y se registra `depleted_at`. Estos lotes pueden seguir apareciendo en auditorías, pero no se usan para consumos futuros.
- **Un solo lote abierto por producto y lote:** la operación `mark_open` comprueba que `opened_units = 0` antes de marcar un lote como abierto. Si ya hay una unidad abierta, se devuelve error 400 para evitar estados ambiguos.
- **Orden FIFO**: el consumo de stock respeta el orden `expiration_date` $\rightarrow$ `entry_date`, lo que minimiza el riesgo de caducidades no controladas.

- **Transacciones atómicas**: las operaciones de salida se realizan dentro de una transacción y usando `select_for_update()` para prevenir condiciones de carrera en entornos concurrentes.
- **Validación temprana**: formatos de UUID, existencia de ubicaciones y productos, y tipos de movimiento se validan al inicio de la petición, devolviendo códigos de error adecuados (`400 Bad Request`, `404 Not Found`, `409 Conflict`).

Con esta lógica, el sistema garantiza que todas las operaciones de escaneo, entradas, salidas y auditorías se reflejen de forma consistente en la base de datos, manteniendo siempre un rastro claro de qué lotes se han consumido, cuáles siguen disponibles y cuáles están abiertos con una caducidad específica tras su apertura.