📘 Documentación del Sistema de Inventario Digital - v1

Fecha de emisión: 2025-06-04

# 1. Introducción y Fuente de Verdad

Este documento establece la arquitectura oficial, tareas asignables, estándares y diseños del Sistema de Inventario Digital.  
Su propósito es alinear el trabajo entre múltiples desarrolladores humanos y asistencias GPT, bajo un único conjunto de reglas.

**⚠️ Nota oficial:** Esta documentación es la única fuente oficial para decisiones de diseño, codificación y arquitectura.  
Cualquier tarea que se realice fuera de este estándar debe ser previamente autorizada por el responsable del sistema (Richard Fica).

# 2. Contenido del Documento

1. Introducción y Fuente de Verdad  
2. Arquitectura del Sistema  
3. Modelos de Datos  
4. Endpoints API REST

# 3. Arquitectura del Sistema

🔄 Nota técnica:  
Este sistema se desarrollará en dos fases. En la \*\*fase 1\*\*, se utilizará \*\*SQL Server\*\* como motor de base de datos por motivos de entorno y compatibilidad. En la \*\*fase 2\*\*, se migrará la base de datos a \*\*MySQL\*\*, utilizando estructuras y convenciones compatibles desde el inicio.

El sistema está basado en una arquitectura cliente-servidor clásica con separación clara entre frontend (React.js), backend (Node.js con Express) y base de datos (MySQL). La organización por capas sigue el patrón MVC (Modelo - Vista - Controlador) en el backend.  
  
La comunicación se realiza mediante una API RESTful segura con autenticación basada en tokens JWT.

## 3.1 Estructura de carpetas - Backend

backend/  
├── config/ # Configuraciones (DB, JWT)  
├── controllers/ # Lógica de negocio y manejo de rutas  
├── middlewares/ # Funciones de validación y autenticación  
├── models/ # Definiciones mssql (SQL Server nativo con consultas directas) (tablas)  
├── routes/ # Archivos de rutas por módulo (auth, products, etc.)  
├── utils/ # Funciones auxiliares (hash, helpers)  
└── server.js # Punto de entrada del backend

## 3.2 Estructura de carpetas - Frontend

frontend/  
├── public/ # Archivos estáticos (favicon, index.html)  
├── src/  
│ ├── components/ # Componentes reutilizables (Header, Modal, etc.)  
│ ├── pages/ # Vistas (Login, Productos, Reportes)  
│ ├── services/ # Llamadas a la API  
│ └── App.js # Enrutamiento principal y lógica raíz  
└── package.json # Dependencias del proyecto frontend

# 4. Modelos de Datos (MySQL)

La base de datos del sistema está diseñada en MySQL siguiendo principios de normalización. A continuación se presentan las tablas principales y sus relaciones.

## Tabla: Role

- RoleId (PK, INT, AUTO\_INCREMENT)  
- Name (VARCHAR) – Ej: admin, operador  
- Description (TEXT)

## Tabla: User

- UserId (PK, INT, AUTO\_INCREMENT)  
- Username (VARCHAR)  
- Email (VARCHAR)  
- PasswordHash (TEXT)  
- RoleId (FK → Role.RoleId)

## Tabla: Category

- CategoryId (PK, INT, AUTO\_INCREMENT)  
- Name (VARCHAR)  
- Description (TEXT)

## Tabla: Product

- ProductId (PK, INT, AUTO\_INCREMENT)  
- Name (VARCHAR)  
- Description (TEXT)  
- CategoryId (FK → Category.CategoryId)  
- MinimumStock (INT)  
- CreatedAt (DATETIME)

## Tabla: StockMovement

- MovementId (PK, INT, AUTO\_INCREMENT)  
- ProductId (FK → Product.ProductId)  
- Type (VARCHAR, 2) – Código del tipo de movimiento (01, 02, 03, 04)  
- Quantity (INT)  
- Unit (VARCHAR)  
- Amount (DECIMAL)  
- DocumentNumber (VARCHAR)  
- LineNumber (INT)  
- AccountingDate (DATE)  
- UserId (FK → User.UserId)  
- Description (TEXT)  
- CreatedAt (DATETIME)

## Relaciones Clave

- Un Role puede estar asignado a muchos Users.  
- Un Category puede contener múltiples Products.  
- Cada Product puede tener múltiples movimientos de stock.

## Diagrama Entidad-Relación (Resumen)

[Role] 1 ────< N [User]  
[Category] 1 ────< N [Product]  
[Product] 1 ────< N [StockMovement]

# 5. Endpoints API REST

La siguiente es la definición estándar de los endpoints REST que el sistema utilizará. Se organizan por módulo y siguen las convenciones RESTful, utilizando JWT para rutas protegidas.

## 🔐 Autenticación

POST /api/auth/register → Registro de nuevo usuario  
POST /api/auth/login → Login con JWT  
GET /api/auth/profile → Obtener perfil del usuario autenticado (protegido)

## 👥 Usuarios y Roles

GET /api/users → Listado de usuarios  
POST /api/users → Crear usuario  
PUT /api/users/:id → Actualizar usuario  
DELETE /api/users/:id → Eliminar usuario  
  
GET /api/roles → Listar roles  
POST /api/roles → Crear rol

## 🗂️ Categorías de Productos

GET /api/categories → Listar categorías  
POST /api/categories → Crear categoría  
PUT /api/categories/:id → Editar categoría  
DELETE /api/categories/:id → Eliminar categoría

## 📦 Productos

GET /api/products → Listar productos  
POST /api/products → Crear producto  
PUT /api/products/:id → Editar producto  
DELETE /api/products/:id → Eliminar producto  
GET /api/products/search → Buscar productos por nombre o categoría

## 📈 Stock y Movimientos

GET /api/stock → Ver stock actual de todos los productos  
POST /api/stock/add → Agregar stock (entrada)  
POST /api/stock/remove → Reducir stock (salida)

## 📊 Reportes

GET /api/reports/low-stock → Productos bajo stock mínimo  
GET /api/reports/movements → Movimientos generales de stock  
GET /api/reports/inventory-summary → Resumen general de inventario

📅 Requisito de cierre de mes:  
El sistema debe permitir que un usuario con privilegios especiales (por ejemplo, rol 'Administrador de Inventario') pueda \*\*cerrar un mes contable\*\*. Una vez cerrado, no se deben permitir nuevos movimientos de stock (entradas, salidas o anulaciones) con fechas dentro de ese mes.  
  
Este cierre debe validarse tanto en backend como frontend y reflejarse en los formularios de registro.

# 6. Detalle de Tareas por Módulo

Cada tarea ha sido descompuesta en unidades independientes que pueden desarrollarse en paralelo.  
Incluye input requerido, entregables, dependencias, interfaz y estado de avance.

## Tarea 1.1 – Modelo User

📌 Descripción: Definir el modelo mssql (SQL Server nativo con consultas directas) para usuarios del sistema.

🔽 Input requerido: Nada (tarea inicial)

📤 Output esperado: Archivo `User.js` con campos: id, username, email, passwordHash, roleId

📁 Archivos involucrados: models/User.js

🔗 Depende de: —

🧩 Interfaz esperada: Definición de modelo mssql (SQL Server nativo con consultas directas) + relaciones

✅ Devuelve: —

👤 Asignado a: \_\_\_\_\_\_

📌 Estado: 🟡 Pendiente

## Tarea 1.2 – POST /api/auth/register

📌 Descripción: Endpoint para registrar un nuevo usuario con password cifrada.

🔽 Input requerido: Modelo User (1.1), Modelo Role (2.1)

📤 Output esperado: Usuario creado, con password cifrada y datos validados

📁 Archivos involucrados: routes/auth.js, controllers/authController.js

🔗 Depende de: 1.1, 2.1

🧩 Interfaz esperada: registerUser(req, res)

✅ Devuelve: 201 Created con userId, username, email

👤 Asignado a: \_\_\_\_\_\_

📌 Estado: 🟡 Pendiente

## Tarea 1.3 – POST /api/auth/login

📌 Descripción: Autenticación de usuario con generación de JWT.

🔽 Input requerido: Usuario existente con contraseña válida

📤 Output esperado: Token JWT, perfil del usuario

📁 Archivos involucrados: routes/auth.js, controllers/authController.js

🔗 Depende de: 1.1

🧩 Interfaz esperada: loginUser(req, res)

✅ Devuelve: 200 OK con token y datos de usuario

👤 Asignado a: \_\_\_\_\_\_

📌 Estado: 🟡 Pendiente

## Tarea 1.4 – Middleware de autenticación

📌 Descripción: Crear middleware para proteger rutas con JWT.

🔽 Input requerido: Token JWT en cabecera Authorization

📤 Output esperado: req.user disponible en controladores

📁 Archivos involucrados: middlewares/authMiddleware.js

🔗 Depende de: 1.3

🧩 Interfaz esperada: verifyToken(req, res, next)

✅ Devuelve: 401 si no hay token válido, next() si todo bien

👤 Asignado a: \_\_\_\_\_\_

📌 Estado: 🟡 Pendiente

## Tarea 2.1 – Modelo Role

📌 Descripción: Definir el modelo mssql (SQL Server nativo con consultas directas) para roles de usuario.

🔽 Input requerido: Nada (puede ser paralelo con 1.1)

📤 Output esperado: Archivo `Role.js` con campos: id, name, description

📁 Archivos involucrados: models/Role.js

🔗 Depende de: —

🧩 Interfaz esperada: Definición de modelo mssql (SQL Server nativo con consultas directas) + relación con User

✅ Devuelve: —

👤 Asignado a: \_\_\_\_\_\_

📌 Estado: 🟡 Pendiente

## Tarea 2.2 – GET /api/users

📌 Descripción: Obtener listado de todos los usuarios con su rol asociado.

🔽 Input requerido: Modelo User y Role definidos

📤 Output esperado: Lista JSON de usuarios

📁 Archivos involucrados: routes/users.js, controllers/userController.js

🔗 Depende de: 1.1, 2.1

🧩 Interfaz esperada: getUsers(req, res)

✅ Devuelve: 200 OK con array de usuarios y roles

👤 Asignado a: \_\_\_\_\_\_

📌 Estado: 🟡 Pendiente

## Tarea 2.3 – POST /api/users

📌 Descripción: Crear un nuevo usuario desde el panel administrativo.

🔽 Input requerido: Datos de usuario (nombre, email, contraseña, roleId)

📤 Output esperado: Usuario creado (internamente usa lógica de 1.2)

📁 Archivos involucrados: routes/users.js, controllers/userController.js

🔗 Depende de: 1.2

🧩 Interfaz esperada: createUser(req, res)

✅ Devuelve: 201 Created con usuario creado

👤 Asignado a: \_\_\_\_\_\_

📌 Estado: 🟡 Pendiente

## Tarea 2.4 – PUT /api/users/:id

📌 Descripción: Actualizar datos de un usuario existente.

🔽 Input requerido: Datos editables del usuario (excepto contraseña opcional)

📤 Output esperado: Usuario actualizado

📁 Archivos involucrados: routes/users.js, controllers/userController.js

🔗 Depende de: 2.2

🧩 Interfaz esperada: updateUser(req, res)

✅ Devuelve: 200 OK con usuario actualizado

👤 Asignado a: \_\_\_\_\_\_

📌 Estado: 🟡 Pendiente

## Tarea 2.5 – DELETE /api/users/:id

📌 Descripción: Eliminar usuario (borrado lógico o físico según convención).

🔽 Input requerido: ID del usuario

📤 Output esperado: Confirmación de borrado

📁 Archivos involucrados: routes/users.js, controllers/userController.js

🔗 Depende de: 2.2

🧩 Interfaz esperada: deleteUser(req, res)

✅ Devuelve: 200 OK con mensaje de éxito

👤 Asignado a: \_\_\_\_\_\_

📌 Estado: 🟡 Pendiente

## Tarea 3.1 – Modelo Category

📌 Descripción: Definir el modelo mssql (SQL Server nativo con consultas directas) para categorías de productos.

🔽 Input requerido: Nada (modelo independiente)

📤 Output esperado: Archivo `Category.js` con campos: id, name, description

📁 Archivos involucrados: models/Category.js

🔗 Depende de: —

🧩 Interfaz esperada: Definición de modelo mssql (SQL Server nativo con consultas directas)

✅ Devuelve: —

👤 Asignado a: \_\_\_\_\_\_

📌 Estado: 🟡 Pendiente

## Tarea 3.2 – CRUD /api/categories

📌 Descripción: Crear todos los endpoints REST para la gestión de categorías.

🔽 Input requerido: Modelo Category (3.1)

📤 Output esperado: Endpoints GET, POST, PUT, DELETE funcionando

📁 Archivos involucrados: routes/categories.js, controllers/categoryController.js

🔗 Depende de: 3.1

🧩 Interfaz esperada: getCategories, createCategory, updateCategory, deleteCategory

✅ Devuelve: 200/201/204 según acción

👤 Asignado a: \_\_\_\_\_\_

📌 Estado: 🟡 Pendiente

## Tarea 3.3 – GET /api/categories (uso en frontend)

📌 Descripción: Endpoint optimizado para usar en select de productos (solo id y nombre).

🔽 Input requerido: Modelo Category (3.1)

📤 Output esperado: Lista simple con id y name

📁 Archivos involucrados: routes/categories.js, controllers/categoryController.js

🔗 Depende de: 3.1

🧩 Interfaz esperada: getCategoryList(req, res)

✅ Devuelve: 200 OK con [{id, name}]

👤 Asignado a: \_\_\_\_\_\_

📌 Estado: 🟡 Pendiente

## Tarea 4.1 – Modelo Product

📌 Descripción: Definir el modelo mssql (SQL Server nativo con consultas directas) para productos.

🔽 Input requerido: Modelo Category (3.1)

📤 Output esperado: Archivo `Product.js` con campos: id, name, description, categoryId, minimumStock, createdAt

📁 Archivos involucrados: models/Product.js

🔗 Depende de: 3.1

🧩 Interfaz esperada: Definición de modelo mssql (SQL Server nativo con consultas directas) + FK con Category

✅ Devuelve: —

👤 Asignado a: \_\_\_\_\_\_

📌 Estado: 🟡 Pendiente

## Tarea 4.2 – CRUD /api/products

📌 Descripción: Crear todos los endpoints REST para productos.

🔽 Input requerido: Modelo Product (4.1)

📤 Output esperado: Endpoints GET, POST, PUT, DELETE funcionando

📁 Archivos involucrados: routes/products.js, controllers/productController.js

🔗 Depende de: 4.1

🧩 Interfaz esperada: getProducts, createProduct, updateProduct, deleteProduct

✅ Devuelve: 200/201/204 según acción

👤 Asignado a: \_\_\_\_\_\_

📌 Estado: 🟡 Pendiente

## Tarea 4.3 – GET /api/products/search

📌 Descripción: Crear endpoint para búsqueda por nombre o categoría.

🔽 Input requerido: Modelo Product (4.1), modelo Category (3.1)

📤 Output esperado: Lista filtrada de productos

📁 Archivos involucrados: routes/products.js, controllers/productController.js

🔗 Depende de: 4.1, 3.1

🧩 Interfaz esperada: searchProducts(req, res)

✅ Devuelve: 200 OK con array filtrado

👤 Asignado a: \_\_\_\_\_\_

📌 Estado: 🟡 Pendiente

## Tarea 5.1 – Modelo StockMovement

📌 Descripción: Definir el modelo mssql (SQL Server nativo con consultas directas) para entradas y salidas de stock.

🔽 Input requerido: Modelo Product (4.1)

📤 Output esperado: Archivo `StockMovement.js` con campos extendidos: tipo, monto, unidad, documento, línea, fecha contable, usuario.

📁 Archivos involucrados: models/StockMovement.js

🔗 Depende de: 4.1

🧩 Interfaz esperada: Definición de modelo mssql (SQL Server nativo con consultas directas) + FK con Product

✅ Devuelve: —

👤 Asignado a: \_\_\_\_\_\_

📌 Estado: 🟡 Pendiente

## Tarea 5.2 – POST /api/stock/add

📌 Descripción: Endpoint para registrar una entrada de stock.

🔽 Input requerido: productId, cantidad, descripción

📤 Output esperado: Registro tipo '01' en StockMovement con unidad, monto, documento y fecha contable

📁 Archivos involucrados: routes/stock.js, controllers/stockController.js

🔗 Depende de: 5.1

🧩 Interfaz esperada: addStock(req, res)

✅ Devuelve: 201 Created con detalle de movimiento

👤 Asignado a: \_\_\_\_\_\_

📌 Estado: 🟡 Pendiente

## Tarea 5.3 – POST /api/stock/remove

📌 Descripción: Endpoint para registrar una salida de stock.

🔽 Input requerido: productId, cantidad, descripción

📤 Output esperado: Registro tipo '03' en StockMovement con unidad, monto, documento y fecha contable

📁 Archivos involucrados: routes/stock.js, controllers/stockController.js

🔗 Depende de: 5.1

🧩 Interfaz esperada: removeStock(req, res)

✅ Devuelve: 201 Created con detalle de movimiento

👤 Asignado a: \_\_\_\_\_\_

📌 Estado: 🟡 Pendiente

## Tarea 5.4 – GET /api/stock

📌 Descripción: Mostrar stock actual por producto, sumando entradas y salidas.

🔽 Input requerido: Datos agregados desde StockMovement

📤 Output esperado: Lista de productos con stock actual

📁 Archivos involucrados: routes/stock.js, controllers/stockController.js

🔗 Depende de: 5.1, 4.1

🧩 Interfaz esperada: getCurrentStock(req, res)

✅ Devuelve: 200 OK con listado de productos y stock actual

👤 Asignado a: \_\_\_\_\_\_

📌 Estado: 🟡 Pendiente

**✳️ Tarea 5.5 – Cierre de Mes**

📌 **Descripción:** Permitir que usuarios con rol especial puedan cerrar un mes contable. Una vez cerrado, no se permiten movimientos de stock en ese mes.

🔽 **Input requerido:** Fecha del mes a cerrar, rol autorizado

📤 **Output esperado:** Registro en tabla ClosedPeriods, validaciones aplicadas en todos los endpoints de movimiento

📁 **Archivos involucrados:**

* models/ClosedPeriod.js
* routes/stock.js
* controllers/stockController.js
* middlewares/monthLock.js

🔗 **Depende de:** 5.1, 5.2, 5.3

🧩 **Interfaz esperada:**

* POST /api/periods/close para cerrar un mes
* Middleware de validación validateMonthOpen(accountingDate)

✅ **Devuelve:**

* 201 Created al cerrar mes exitosamente
* 403 Forbidden si se intenta mover stock en mes cerrado

👤 **Asignado a:** \_\_\_\_\_\_

📌 **Estado:** 🟡 Pendiente

## Tarea 6.1 – GET /api/reports/low-stock

📌 Descripción: Listar productos cuyo stock actual esté por debajo del mínimo definido.

🔽 Input requerido: Modelos Product (4.1) y StockMovement (5.1)

📤 Output esperado: Listado de productos con stock bajo

📁 Archivos involucrados: routes/reports.js, controllers/reportController.js

🔗 Depende de: 4.1, 5.1, 5.4

🧩 Interfaz esperada: getLowStockProducts(req, res)

✅ Devuelve: 200 OK con array de productos bajo stock mínimo

👤 Asignado a: \_\_\_\_\_\_

📌 Estado: 🟡 Pendiente

## Tarea 6.2 – GET /api/reports/movements

📌 Descripción: Reportar todos los movimientos de stock, filtrables por rango de fechas.

🔽 Input requerido: Query params: desde, hasta

📤 Output esperado: Listado de movimientos con fecha, tipo, producto, cantidad

📁 Archivos involucrados: routes/reports.js, controllers/reportController.js

🔗 Depende de: 5.1

🧩 Interfaz esperada: getStockMovements(req, res)

✅ Devuelve: 200 OK con array de movimientos

👤 Asignado a: \_\_\_\_\_\_

📌 Estado: 🟡 Pendiente

## Tarea 6.3 – GET /api/reports/inventory-summary

📌 Descripción: Resumen general del stock por producto: entradas, salidas y saldo.

🔽 Input requerido: Agregado desde StockMovement por producto

📤 Output esperado: Lista con nombre del producto, total entradas, salidas, stock actual

📁 Archivos involucrados: routes/reports.js, controllers/reportController.js

🔗 Depende de: 5.1, 5.4

🧩 Interfaz esperada: getInventorySummary(req, res)

✅ Devuelve: 200 OK con resumen inventario

👤 Asignado a: \_\_\_\_\_\_

📌 Estado: 🟡 Pendiente

# 7. Diseño de Interfaces (Wireframes)

A continuación se presentan los diseños funcionales de las vistas principales del sistema, en formato wireframe de texto (ASCII).   
Estos diseños permiten a los desarrolladores y diseñadores seguir una misma visión funcional, sin ambigüedad en campos ni flujos.

## WF1 – Pantalla de Login

+-----------------------------------+  
| INICIAR SESIÓN |  
+-----------------------------------+  
| Usuario: [\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_] |  
| Clave: [\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_] |  
| |  
| [ Ingresar ] ¿Olvidó su clave?|  
+-----------------------------------+

## WF2 – Panel Principal

+--------------------+-------------------------------+  
| Menú | Área de trabajo |  
|--------------------|-------------------------------|  
| • Productos | [Vista seleccionada] |  
| • Categorías | |  
| • Movimientos | |  
| • Reportes | |  
| | |  
+--------------------+-------------------------------+

## WF3 – Gestión de Productos

+--------------------------------------------------------+  
| Productos |  
+--------------------------------------------------------+  
| [ + Nuevo Producto ] [ Buscar: \_\_\_\_\_\_\_\_ ] [🔍] |  
+--------------------------------------------------------+  
| Nombre | Categoría | Stock | Acciones |  
|--------------|------------|-------|--------------------|  
| Teclado MIDI | Electrónica| 12 | [✏️][🗑️] |  
| Micrófono | Audio | 5 | [✏️][🗑️] |  
+--------------------------------------------------------+

## WF4 – Stock y Movimientos

+-------------------------------------------------------------+  
| Movimientos de Inventario Detallados |  
+-------------------------------------------------------------+  
| Producto: [\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_] |  
| Tipo Movimiento: [01-Ingreso / 02-Anulación / |  
| 03-Egreso / 04-Anulación] |  
| Cantidad: [\_\_\_\_] Unidad: [\_\_\_\_\_\_\_\_\_\_] |  
| Monto: [\_\_\_\_\_\_\_\_\_\_\_] $ |  
| Fecha Contable: [\_\_\_\_\_\_/\_\_\_\_\_\_/\_\_\_\_\_\_] |  
| Documento N°: [\_\_\_\_\_\_\_\_\_\_] Línea: [\_\_] |  
| Descripción: [\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_] |  
| |  
| [ Registrar Movimiento ] |  
+-------------------------------------------------------------+  
| Historial |  
+-------------------------------------------------------------+  
| Fecha | Producto | Tipo | Cant | Doc | Usuario |  
|-------------|---------------|--------|------|-----|---------|  
| 2025-06-01 | Teclado MIDI | 01 | 5 |OC01 | admin |  
+-------------------------------------------------------------+

## WF5 – Reportes

+--------------------------------------------------------+  
| Reportes de Inventario |  
+--------------------------------------------------------+  
| [ Reporte: ▼ ] Desde: [\_\_\_\_] Hasta: [\_\_\_\_] [🔍] |  
+--------------------------------------------------------+  
| Tipo de Reporte seleccionado: |  
| [ ] Productos bajo stock mínimo |  
| [ ] Movimientos por fecha |  
| [ ] Resumen de inventario |  
+--------------------------------------------------------+  
| Resultados abajo con tabla o gráfico según caso |  
+--------------------------------------------------------+

# 8. Contratos de Datos y Convenciones de Respuesta

Esta sección define los formatos estándar de datos entre backend y frontend, y las convenciones HTTP de respuesta.  
Permite una integración robusta y predecible entre tareas que comparten datos.

## 8.1 Formato de Usuario (User)

{  
 "userId": 12,  
 "username": "admin123",  
 "email": "admin@empresa.cl",  
 "role": {  
 "roleId": 1,  
 "name": "admin"  
 }  
}

## 8.2 Formato de Producto (Product)

{  
 "productId": 7,  
 "name": "Teclado MIDI",  
 "description": "Controlador de 49 teclas",  
 "category": {  
 "categoryId": 2,  
 "name": "Música"  
 },  
 "minimumStock": 5,  
 "createdAt": "2025-06-04T12:00:00Z"  
}

## 8.3 Movimiento de Stock (StockMovement)

{  
 "movementId": 15,  
 "productId": 7,  
 "type": "01",  
 "quantity": 10,  
 "unit": "unidad",  
 "amount": 52000.50,  
 "documentNumber": "OC-1001",  
 "lineNumber": 1,  
 "accountingDate": "2025-06-04",  
 "userId": 12,  
 "description": "Ingreso por compra OC-1001 línea 1",  
 "createdAt": "2025-06-04T15:30:00Z"  
}

## 8.4 Convención de Formato de Fechas

Todas las fechas siguen el formato ISO 8601: `YYYY-MM-DDTHH:mm:ssZ`

## 8.5 Convenciones de Respuesta HTTP

• 200 OK - Respuesta exitosa con datos  
• 201 Created - Recurso creado exitosamente  
• 204 No Content - Eliminación o acción sin contenido de retorno  
• 400 Bad Request - Error de validación o formato de datos  
• 401 Unauthorized - Token inválido o ausente  
• 404 Not Found - Recurso no encontrado  
• 409 Conflict - Duplicado o conflicto de datos

# 9. Flujo Integrado de Usuario

Este flujo describe cómo se encadenan las acciones entre módulos para lograr un caso de uso completo.  
Permite verificar que los endpoints y tareas colaboran correctamente entre sí.

## Ejemplo: Flujo desde Login hasta Reporte

1. Usuario accede a `POST /api/auth/login` con credenciales válidas → recibe JWT.  
2. JWT se usa para autenticar llamadas posteriores.  
3. Usuario accede a `GET /api/products` → lista productos disponibles.  
4. Selecciona producto y registra `POST /api/stock/add` o `remove`.  
5. Puede consultar `GET /api/stock` para ver stock actualizado.  
6. Finalmente, accede a `GET /api/reports/inventory-summary` para ver resumen total.

# 10. Matriz de Conversación entre Tareas

Esta tabla resume las relaciones funcionales entre tareas, indicando qué módulos generan datos que otros consumen.  
Facilita la trazabilidad y evita inconsistencias entre desarrolladores.

|  |  |  |  |
| --- | --- | --- | --- |
| Tarea Origen | Provee a Tarea | Formato Compartido | Comentario |
| 1.1 - Modelo User | 1.2, 1.3, 2.2 | { userId, username, email, roleId } | Modelo base de usuario |
| 2.1 - Modelo Role | 1.2, 2.3, GET /roles | { roleId, name } | Select en formularios |
| 3.1 - Modelo Category | 4.1, 4.3 | { categoryId, name } | Clasificación de productos |
| 4.1 - Modelo Product | 5.1, 5.2, 5.3, 6.1, 6.3 | { productId, name } | Usado en stock y reportes |
| 5.1 - Modelo StockMovement | 5.4, 6.2, 6.3 | { movementId, productId, type, quantity } | Base de movimientos |

# 11. Flujos Adicionales de Usuario

Además del flujo general principal, se incluyen aquí otros flujos comunes entre módulos que garantizan interacción e integración entre tareas.

## Flujo: Creación de usuario desde el panel admin

1. Admin accede a `POST /api/users` (tarea 2.3)  
2. Elige `roleId` desde `GET /api/roles` (tarea 2.1)  
3. Usuario se crea vía `createUser` reutilizando la lógica de `1.2`  
4. Usuario luego accede a `POST /api/auth/login`

## Flujo: Registro de nuevo producto hasta visualización en reporte

1. Se crea producto con `POST /api/products` (tarea 4.2)  
2. Se le hace un ingreso de stock con `POST /api/stock/add` (tarea 5.2)  
3. Puede visualizarse con `GET /api/stock` (tarea 5.4)  
4. Y aparecerá en `GET /api/reports/inventory-summary` (tarea 6.3)

## 8.6 Códigos de Tipo de Movimiento

• 01 - Ingreso por compra  
• 02 - Anulación de ingreso  
• 03 - Egreso por consumo  
• 04 - Anulación de egreso

Para permitir el desarrollo paralelo sin depender de módulos aún no implementados, se define un conjunto de datos simulados ("mock data") que representan respuestas estándar del sistema.

Estos archivos deben ubicarse en la carpeta `/mock/` dentro del repositorio, y estar disponibles para todos los desarrolladores e IAs participantes.

Ejemplos incluidos:

📁 /mock/

- mockUser.json

{

"userId": 1,

"username": "admin",

"email": "admin@inventario.cl",

"role": "Administrador",

"password": "1234",

"token": "JWT123456789"

}

- mockProduct.json

{

"productId": 101,

"name": "Laptop HP",

"description": "Laptop de alto rendimiento",

"categoryId": 5,

"stock": 20,

"price": 799990,

"unit": "unidad",

"barcode": "1234567890123"

}

- mockCategory.json

{

"categoryId": 5,

"name": "Electrónica",

"description": "Productos electrónicos y computacionales"

}

- mockStockMovement.json

{

"movementId": 555,

"productId": 101,

"type": "SALIDA",

"quantity": 3,

"documentNumber": "EG-2025-001",

"accountingDate": "2025-06-04T10:30:00Z",

"motive": "Entrega a cliente",

"createdBy": "admin"

}

- mockReportData.json

{

"reportId": 1,

"generatedAt": "2025-06-04T12:00:00Z",

"filter": {

"category": "Electrónica",

"dateRange": ["2025-06-01", "2025-06-04"]

},

"results": [

{

"productId": 101,

"name": "Laptop HP",

"entries": 10,

"exits": 5,

"currentStock": 5

}

]

}

## 8.7 Conjuntos de Datos Mock para Desarrollo Paralelo

- mockClosedPeriod.json

{

"closedPeriods": [

{

"id": 1,

"month": "2025-05",

"closedBy": "admin",

"closedAt": "2025-06-01T00:00:00Z"

}

]

}

# Adaptación del sistema sin Sequelize

El sistema ha sido adaptado para prescindir completamente del ORM Sequelize. En su lugar, se utiliza el paquete oficial 'mssql' de Node.js para conectarse a SQL Server y ejecutar consultas nativas. Esto mejora el control, la transparencia de las transacciones, y elimina posibles errores como recursiones cíclicas o conflictos de modelos pluralizados.

Cambios clave:

* - Se eliminan todos los modelos ORM.
* - Todas las operaciones CRUD se realizan con queries SQL puras usando `mssql` y promesas.
* - Las relaciones entre tablas se manejan mediante `JOIN` explícitos.
* - Las transacciones se manejan manualmente usando `BEGIN TRANSACTION`, `COMMIT`, y `ROLLBACK`.
* - Se mantiene la lógica de cierre de periodo, validación por mes, y registro de usuario.