RetailChainDB: Sistema de Integración SQL Server - MongoDB

Esta documentación técnica presenta un sistema de integración entre SQL Server y MongoDB para gestionar una cadena minorista. El sistema aprovecha las fortalezas de ambas bases de datos para optimizar diferentes aspectos del negocio: SQL Server maneja datos transaccionales estructurados, mientras que MongoDB gestiona datos flexibles no estructurados.

1. Arquitectura del Sistema

1.1 Estructura General

El sistema se basa en una arquitectura de microservicios con dos componentes principales de almacenamiento:

- **SQL Server**: Almacena datos estructurados y transaccionales (productos, inventario, tiendas, empleados)
- **MongoDB**: Almacena datos no estructurados y relaciones flexibles (preferencias de clientes, detalles de reservas)

1.2 Capas de la Aplicación

- **Controllers**: Manejan las solicitudes HTTP y coordinan respuestas
- **Services**: Contienen la lógica de negocio y orquestan operaciones
- **Models**: Definen las estructuras de datos para ambas bases de datos
- **Database Clients**: Administran conexiones a las bases de datos

1.3 Diagrama de Componentes

```
RetailChainDB/
├── controllers/         # Controladores HTTP
├── services/            # Servicios de lógica de negocio
├── models/              # Modelos de datos SQL y MongoDB
├── databases/           # Conexiones a bases de datos
├── routes/              # Rutas API
├── utils/               # Utilidades comunes
├── config/              # Configuración de la aplicación
└── integration/         # Componentes de integración entre DB
```

2. Modelo de Datos

2.1 Esquema SQL Server

2.1.1 Esquemas Principales

- **Store**: Contiene información de ubicaciones geográficas y tiendas
- **HR**: Administra datos de empleados y estructura organizacional
- **Inventory**: Gestiona productos, categorías, proveedores y niveles de stock
- **Sales**: Controla ventas, clientes, promociones y programas de fidelidad
- **Audit**: Mantiene registros de auditoría y seguimiento de cambios

2.1.2 Tablas Clave

- `Store.Store`: Información de tiendas físicas
- `HR.Employee`: Datos de empleados
- `Inventory.Product`: Catálogo de productos
- `Inventory.StoreInventory`: Niveles de stock por tienda
- `Sales.Sale` y `Sales.SaleDetail`: Transacciones de venta

2.2 Esquema MongoDB

2.2.1 Colecciones Principales

- `Customer`: Información extendida de clientes
- `Reservation`: Reservas de productos para recogida posterior

2.2.2 Estructura de Documentos

Customer:

```
{
  sqlCustomerId: Number,        // ID del cliente en SQL Server
  firstName: String,
  lastName: String,
  email: String,
  preferences: {                // Preferencias flexibles
    preferredCategories: [String],
    preferredStores: [Number],
    communicationPreferences: {
      email: Boolean,
      sms: Boolean,
      pushNotifications: Boolean
    }
  },
  lastActivity: Date
}
```

Reservation:

```
{
  customerId: ObjectId,          // Referencia al cliente en MongoDB
  sqlCustomerId: Number,         // ID del cliente en SQL Server
  storeId: Number,               // ID de la tienda en SQL Server
  productId: Number,             // ID del producto en SQL Server
  quantity: Number,
  status: String,                // PENDING, CONFIRMED, COMPLETED, CANCELLED,
EXPIRED
  confirmationCode: String,
  expiryDate: Date,
  statusHistory: [{              // Historial de cambios
    status: String,
    date: Date,
    comment: String
  }]
}
```

3. Integración entre Bases de Datos

3.1 Estrategias de Sincronización

El sistema emplea varias estrategias para mantener la coherencia entre SQL Server y MongoDB:

1. **Identificadores Comunes**: Cada entidad en MongoDB mantiene referencias a los IDs correspondientes en SQL Server
2. **Sincronización Programada**: Scripts como `syncCustomer.js` sincronizan datos periódicamente
3. **Transacciones Distribuidas**: Para operaciones que afectan a ambas bases de datos

3.2 Flujo de Reserva (Caso de Uso Principal)

El flujo de reserva de productos ilustra la integración entre ambas bases de datos:

1. El cliente verifica disponibilidad (consulta SQL Server)
2. Se crea un documento de reserva en MongoDB
3. Se actualiza el inventario en SQL Server
4. Se genera un código de confirmación
5. El cliente puede confirmar, cancelar o completar la reserva, afectando ambas bases de datos

3.3 Manejo de Integridad

Para asegurar la integridad referencial:

- Los métodos en `reservation.service.js` realizan operaciones atómicas
- Se implementan verificaciones de consistencia antes de las actualizaciones
- Se utilizan patrones de compensación para manejar fallos

4. API RESTful

4.1 Endpoints Principales

4.1.1 Inventario

- `GET /api/v1/inventory`: Lista de inventario
- `GET /api/v1/inventory/stores/{storeId}`: Inventario por tienda
- `GET /api/v1/inventory/products/{productId}`: Inventario por producto
- `GET /api/v1/inventory/availability/{storeId}/{productId}`: Verifica disponibilidad
- `PUT /api/v1/inventory/update/{storeId}/{productId}`: Actualiza inventario
- `POST /api/v1/inventory/transfer`: Transfiere inventario entre tiendas

4.1.2 Reservas

- `POST /api/v1/reservation`: Crea reserva
- `GET /api/v1/reservation/{code}`: Obtiene detalles
- `PUT /api/v1/reservation/{code}/confirm`: Confirma reserva
- `PUT /api/v1/reservation/{code}/cancel`: Cancela reserva
- `PUT /api/v1/reservation/{code}/complete`: Completa reserva
- `GET /api/v1/reservation/active`: Lista reservas activas

4.2 Documentación Swagger

La API está documentada con Swagger, accesible en `/api-docs` cuando el servidor está en ejecución.

5. Implementación Técnica

5.1 Tecnologías Utilizadas

- **Backend**: Node.js, Express.js
- **Bases de Datos**: SQL Server, MongoDB
- **Conexiones**: mssql/msnodesqlv8, mongoose
- **Documentación**: Swagger/OpenAPI
- **Utilidades**: moment.js, chalk, crypto

5.2 Manejo de Conexiones

5.2.1 SQL Server

```
// En sqlClient.js
const connectToSQLServer = async () => {
  try {
    sqlPool = await sql.connect(sqlConfig);
    Logger.log(`SQL Server connection established successfully`, 'success');
    return sqlPool;
  } catch (error) {
    Logger.log(`SQL Server connection failed: ${error.message}`, "error");
    process.exit(1);
  }
}
```

5.2.2 MongoDB

```
// En mongoClient.js
const connectToMongoDB = async () => {
  try {
    await mongoose.connect(mongoURI, {
      dbName: "RetailChainDB",
    });
    Logger.log("MongoDB connection established successfully", "success");
  } catch (error) {
    Logger.log(`MongoDB connection failed: ${error.message}`, "error");
    process.exit(1);
  }
}
```

5.3 Modelos y Servicios

5.3.1 Modelos SQL

Los modelos SQL (`sqlModels.js`) implementan clases con métodos estáticos para operaciones comunes:

```
class Inventory {
  static async checkAvailability(storeId, productId) {
    // Consulta SQL para verificar disponibilidad
  }

  static async updateInventory(storeId, productId, quantity) {
    // Actualiza inventario y registra transacción
  }
}
```

5.3.2 Modelos MongoDB

Los esquemas de MongoDB (`mongoModels.js`) incluyen hooks y métodos estáticos:

```
const ReservationSchema = new mongoose.Schema({
  // Definición del esquema
});

// Método para actualizar reservas expiradas
ReservationSchema.statics.updateExpiredReservations = async function() {
  // Actualización de reservas expiradas
};
```

5.3.3 Servicios de Integración

Los servicios coordinan operaciones entre bases de datos:

```
// En reservation.service.js
const createReservation = async (data) => {
  // 1. Verifica disponibilidad en SQL Server
  const availability = await Inventory.checkAvailability(storeId, productId);

  // 2. Crea reserva en MongoDB
  const reservation = new Reservation({...});
  await reservation.save();

  // 3. Actualiza inventario en SQL Server
  await Inventory.updateInventory(storeId, productId, -quantity);

  return { /* datos de reserva */ };
}
```

6. Escalabilidad y Rendimiento

6.1 Optimizaciones

- **Índices**: Definidos en MongoDB y SQL Server para consultas frecuentes
- **Procesamiento por Lotes**: Implementado en scripts de sincronización
- **Estructura de Documentos**: Diseñada para minimizar búsquedas adicionales

6.2 Monitoreo

El sistema utiliza un logger personalizado (`logger.js`) que ofrece diferentes niveles de registro:

```
Logger.info("Información general");
Logger.error("Error crítico");
```

```
Logger.success("Operación exitosa");
Logger.debug("Información de depuración");
```

7. Seguridad

7.1 Variables de Entorno

Todas las credenciales y configuraciones sensibles están en variables de entorno:

```
SQL_SERVER = "ServerName\InstanceName"
SQL_DATABASE = "DatabaseName"
MONGO_URI = "mongodb://localhost:27017"
```

7.2 Validación de Entradas

Los controladores validan todas las entradas del usuario antes de procesarlas:

```
if (!storeId || !productId) {
  return res.status(400).json({
    success: false,
    message: 'Store ID and Product ID are required'
  });
}
```

8. Despliegue y Configuración

8.1 Requisitos Previos

- Node.js (v14+)
- SQL Server con controlador ODBC (msnodesqlv8)
- MongoDB (v4.4+)

8.2 Instrucciones de Instalación

1. Clonar el repositorio
2. Instalar dependencias: `npm install`
3. Crear archivo `.env` basado en `.env.EXAMPLE`
4. Crear bases de datos:
    - SQL Server: ejecutar scripts en carpeta `sql-server/schema`
    - MongoDB: ejecutar `nosql/data/seed-data.js`
5. Iniciar servidor: `npm run dev` (desarrollo) o `npm start` (producción)

9. Consideraciones y Mejoras Futuras

- Implementar caché para consultas frecuentes
- Mejorar manejo de transacciones distribuidas
- Expandir documentación API
- Implementar pruebas automatizadas
- Añadir métricas de rendimiento

10. Conclusión

Este sistema de integración aprovecha las fortalezas complementarias de SQL Server y MongoDB para crear una solución robusta de gestión minorista. SQL Server proporciona consistencia transaccional y relaciones estructuradas para los datos críticos del negocio, mientras que MongoDB ofrece flexibilidad y escalabilidad para datos en evolución como preferencias de clientes y flujos de reservas.

La arquitectura orientada a servicios facilita el mantenimiento y las extensiones futuras, mientras que la documentación completa de API con Swagger permite la integración con otros sistemas.