Sistema de Gerenciamento de Vinhos

Rafael Granja Henrique Menezes Larissa Goularte

Maio 2025

Contents

1 Introdução

O consumo de vinhos envolve processos complexos de cultivo, armazenamento e comercialização. Para apoiar decisões em cada etapa, é fundamental ter um sistema confiável que armazene informações sobre regiões, castas, quantidades de colheita e vendas.

Neste trabalho, construímos dois subsistemas complementares:

1. Um banco de dados relacional em SQL Server, responsável por guardar dados estruturados sobre vinhos.

2. Uma aplicação API em Node.js que expõe esses dados via endpoints REST, permitindo integrações com interfaces web ou outras ferramentas analíticas.

Na Seção 2, apresentamos a visão geral da arquitetura. Em seguida, detalhamos as configurações e scripts do banco em Seção 3, explicando cada parte do `schema.sql` e do `seed.sql`. Na Seção 4 mostramos como a API se conecta ao banco e responde a requisições simples. Para facilitar testes, adicionamos uma view de exemplo em Seção 5. Por fim, discutimos testes, resultados e melhores práticas em Seções 6 e 7.

2 Hierarquia do Sistema

A arquitetura proposta segue o padrão cliente-servidor:

- **Database System (SQL Server)**: Responsável pelo armazenamento persistente. Usamos Docker Compose para orquestração, garantindo funcionamento em diferentes sistemas operacionais.

- **API Application (Node.js)**: Servidor REST que escuta requisições HTTP na porta 3000 e devolve dados em JSON ou renderiza views via motor de templates.

O fluxo de comunicação é:

1. O cliente faz uma requisição HTTP para a API (por exemplo, `/hello`).

2. A API processa a rota, acessa o módulo de conexão com o SQL Server e executa queries.

3. Os resultados são retornados como objetos JavaScript e passados para o template ou serializados em JSON.

4. A resposta é enviada ao cliente como HTML renderizado ou JSON, conforme a rota.

3 System 1: Sistema de Base de Dados (SQL Server)

O sistema de banco de dados foi criado com scripts SQL executados em um container Docker. A seguir, expandimos os principais elementos do `schema.sql`.

3.1 Function: GuidToCreatedAt

Para rastrear quando cada registro foi criado sem depender de timestamps explícitos, usamos GUIDs e uma função que extrai a data de criação de seu componente:

Listing 1: Função para extrair data de criação de GUID

```
CREATE OR ALTER FUNCTION dbo.GuidToCreatedAt(@g
   UNIQUEIDENTIFIER)
RETURNS DATETIME2
AS
BEGIN
    DECLARE @bin VARBINARY(16) = CONVERT(VARBINARY(16),
       @g);
    RETURN DATEADD(SECOND,
        CAST(SUBSTRING(@bin, 1, 4) AS INT) / 300,
        '1970-01-01');
END;
```

3.2 Definições de Tabelas

Definimos tabelas principais e seus relacionamentos:

- `regiao`: Áreas geográficas com chave `id`.
- `casta`: Tipos de uva, relacionados a colheitas.
- `vinha`: Associa cada casta a uma região.
- `colheita`: Registra quantidade e qualidade por safra.
- `vinho`: Representa cada vinho produzido, com composição e relação de estoque.
- `composicaoVinho`: Descreve percentuais de cada casta em um vinho.
- `lote`: Lotes engarrafados, vinculados a vinhos e preços.
- `cliente`, `venda`, `detalheVenda`: Fluxo comercial e histórico de vendas.
- `precoCliente`: Tabelas de preços diferenciados por categoria de cliente.
- `utilizador`: Controle de acesso e auditoria de usuários.

3.3 Data Inicial (Seed)

O `seed.sql` insere dados iniciais:

- Regiões: Douro, Alentejo, Dão.
- Castas populares: Touriga, Arinto, Alvarinho.
- Safras das colheitas de 2022 e 2023.
- Preços diferenciados para clientes regulares e VIP.
- Vendas de exemplo para teste de integridade.

4 System 2: Aplicação API (Node.js)

A aplicação API é composta por módulos de configuração, conexão, rotas e views.

4.1 Configuration (config.js)

Armazena credenciais e parâmetros de conexão:

Listing 2: Configuração de conexão (config.js)

```
const config = {
  user: 'sa',
  password: 'Str0ngPassword@',
  server: 'localhost',
  database: 'vinhos',
  options: { encrypt: false, trustServerCertificate: true
     , enableArithAbort: true },
  port: 1433
};
module.exports = config;
```

4.2 Módulo de Conexão à Base de Dados (src/index.js)

Conexão e execução de queries usando `mssql`:

Listing 3: Módulo de conexão (src/index.js)

```
const sql = require('mssql');
const config = require('./config');

async function connectDB() {
  const pool = await sql.connect(config);
  console.log('Conex o estabelecida!');
  return pool;
}
```

```
async function executeQuery(query) {
  const pool = await connectDB();
  return await pool.request().query(query);
}

module.exports = { connectDB, executeQuery };
\end{lstisting}

\subsection{Rotas da API}
\paragraph{Route /hello (routes/hello.js)}
Retorna as regi es cadastradas:
\begin{lstlisting}[language=JavaScript, caption={Rota /
   hello (routes/hello.js)}]
const { executeQuery } = require('../src/index');

async function registerRoutes(fastify) {
  fastify.get('/hello', async (req, rep) => {
    const { recordset } = await executeQuery('SELECT TOP
       5 * FROM regiao');
    rep.view('hello', { recordset });
  });
}
module.exports = registerRoutes;
```

5 View de Exemplo (hello.ejs)

Template EJS que exibe as regiões:

Listing 4: Template de visão (views/hello.ejs)

```
<!DOCTYPE html>
<html>
<head><title>Regi es de Vinho</title></head>
<body>
  <h1>Regi es de Vinho</h1>
  <ul>
    <% recordset.forEach(reg => { %>
      <li><%= reg.nome %></li>
    <% }) %>
  </ul>
</body>
</html>
```

6 Resultados e Validações

Para validar, executamos:

- CRUD em tabelas.
- Chamadas simultâneas às diferentes rotas das APIs.
- Verificação de fechamento de conexões.

7 Discussões

Recomendações:

- Use variáveis de ambiente em produção.
- Inclua testes automatizados (Jest).
- Centralize logs e erros.

8 Contribuições dos Integrantes

- Henrique Menezes: definiu a modelagem e configuração do banco de dados.
- Rafael Granja: realizou a modelação de tabelas e elaboração de relatórios.
- Larissa Goularte: implementou as rotas da API e integração com views.

9 Scripts Adicionais

Arquivos auxiliares:

- `docker-compose.yml`: Orquestra o container do SQL Server para rodar o banco em qualquer sistema operacional.
- `init-db.sh`: Configura o banco no container e executa `schema.sql` e `seed.sql` automaticamente.