Documentação do Projeto SuperJU

Resumo do Projeto

O **SuperJU** é um sistema de controle de estoque e vendas que permite o cadastro de clientes, produtos, gestão de entrada de produto, gestão de vendas e a geração de relatórios detalhados de estoque e vendas. O sistema foi construído utilizando como frontend o projeto WebForms .NET 4.8 que se comunica com a API Rest .NET 8 feita no padrão MVC.

O Projeto Web **SuperJU.WEB** é responsável por trabalhar a parte visual com o operador do sistema, realizar validações de dados inputados na tela pelo operador e fazer comunicação para persistir/buscar informações através de requisições RESTpara a API.

A API **SuperJU.API** é reponsável por fornecer todos endpoints client rest necessários para a operação do projeto Web com as principais operações: CRUD produtos e cliente, operações lógicas de negócio na entrada de produto e saída na venda, e relatórios. O projeto API também segue o padrão de repostas HTTP para os seguintes status 200, 201, 204, 400, 404 e 500.

Tecnologias Utilizadas

- **C#** (.NET Framework 8.0)
- **ASP.NET WebForms 4.8**
- **API REST** no padrão **MVC**:
    - Arquitetura: Controller, Service, Repository
    - ADO.NET para acesso aos dados
- **Injeção de Dependências** para gerenciamento de serviços e repositórios
- **SQL Server** (banco de dados)
- **Docker** (SQL Server container)
- **DBeaver** (cliente para interação com o banco de dados)
- **HttpClient** para comunicação com a API
- **JSON** para formatação de dados entre cliente e API
- **xUnit** e **Moq** para testes unitários na camada de serviços da aplicação API

Funcionalidades

- **Gestão de Produtos**: Operações de pesquisa, cadastro e alteração de produtos.
- **Gestão de Clientes**: Operações de pesquisa, cadastro e alteração de produtos.
- **Gestão de Entrada de Produto**: Operação de pesquisa entradas, realizar entrada de produtos e visualização de entrada de produtos. Ao realizar entrada de mercadoria o sistema aumenta o estoque do produto bem como atualiza seu valor de custo e seu valor de venda.
- **Gestão de Pedidos**: Operação de pesquisa pedidos, realizar pedido e visualização de pedidos. Ao realizar uma venda o sistema diminui o estoque do produto.
- **Relatórios**: Geração de relatórios detalhados de vendas e estoque.

Estrutura de Banco de Dados

O projeto utiliza o **SQL Server** como banco de dados, com as seguintes tabelas principais:

- **PRODUTOS**: Gerencia os produtos disponíveis para venda.
- **CLIENTES**: Armazena informações dos clientes.
- **ENTRADAS_PRODUTO** e **ENTRADAS_PRODUTO_ITEM**: Registram as entradas de mercadorias e seus itens.
- **FORMAS_PAGAMENTO**: Contém as formas de pagamento disponíveis (ex.: Dinheiro, Cartão, Pix).
- **PEDIDOS** e **PEDIDOS_ITEM**: Controlam os pedidos e os itens de cada pedido.

Docker Compose e Scripts SQL Utilizados

O banco de dados do projeto é configurado utilizando o Docker e um script SQL, que cria as tabelas e inserem dados iniciais. Para subir o banco de dados, o Visual Studio 2022 deve ser utilizado com o Prompt de Comando do Desenvolvedor, apontando o diretório para a raiz do projeto. Os seguintes comandos devem ser executados para iniciar ou desligar o Docker relacionado ao banco de dados.

```
Comando Start Banco:
docker-compose up -d

Comando Desligar Docker:
docker-compose down
```

O script também inclui a criação da database superju do login superju_user e as permissões necessárias para operar no banco, e configura as principais tabelas utilizadas pelo sistema, como PRODUTOS, CLIENTES, ENTRADAS_PRODUTOS, PEDIDOS, entre outras. O script também realiza as inserções iniciais nas tabelas, como as formas de pagamento.

```
USE [master];

GO
IF NOT EXISTS (SELECT * FROM sys.sql_logins WHERE name = 'superju_user')
BEGIN
    CREATE LOGIN [superju_user] WITH PASSWORD = 'superju_user123', CHECK_POLICY = OFF;
    ALTER SERVER ROLE [sysadmin] ADD MEMBER [superju_user];

END

GO
IF DB_ID('superju') IS NULL
BEGIN
    CREATE DATABASE [superju];
END

GO
USE [superju];
```

```sql
GO
IF OBJECT_ID('dbo.FORMAS_PAGAMENTO') IS NULL
BEGIN
    CREATE TABLE [dbo].[FORMAS_PAGAMENTO] (
        [Id] [int] IDENTITY(1,1) NOT NULL,
        [Nome] [nvarchar] (50) NOT NULL,
        [Descricao] [nvarchar] (100) NOT NULL,
        CONSTRAINT [PK_FORMAS_PAGAMENTO] PRIMARY KEY ([Id])
    );
    INSERT INTO [dbo].[FORMAS_PAGAMENTO] VALUES ('Dinheiro','Pagamento em dinheiro');
    INSERT INTO [dbo].[FORMAS_PAGAMENTO] VALUES ('Cartão','Pagamento em cartão');
    INSERT INTO [dbo].[FORMAS_PAGAMENTO] VALUES ('Pix','Pagamento em pix');
END
```

Comunicação entre WebForms e API

A comunicação entre o **ASP.NET WebForms** e a **API REST** foi estabelecida utilizando **HttpClient**, e a URL base da API foi configurada no arquivo web.config. Abaixo estão os detalhes dessa conexão.

Configuração no web.config

A URL base da API foi definida no web.config como uma chave de configuração, permitindo flexibilidade para alterações sem precisar recompilar o código:

```xml
<configuration>
    <appSettings>
        <add key="SUPER_JU_API_URL" value="http://localhost:5083"/>
    </appSettings>
```

Requisições para a API utilizando HttpClient

As requisições para a API são feitas utilizando a classe HttpClient, com a URL configurada acessada através de ConfigurationManager.AppSettings.Get().

```
private static readonly string SUPER_JU_API_URL = ConfigurationManager.AppSettings.Get("SUPER_JU_API_URL");

private static readonly string URL_CLIENTE_PESQUISA = SUPER_JU_API_URL + "/clientes";
private static readonly string URL_CLIENTE_POR_ID = SUPER_JU_API_URL + "/clientes/{0}";
private static readonly string URL_CLIENTE_CADASTRAR = SUPER_JU_API_URL + "/clientes";
private static readonly string URL_CLIENTE_EDITAR = SUPER_JU_API_URL + "/clientes/{0}";

private static readonly string URL_PRODUTO_PESQUISA = SUPER_JU_API_URL + "/produtos";
private static readonly string URL_PRODUTO_POR_ID = SUPER_JU_API_URL + "/produtos/{0}";
private static readonly string URL_PRODUTO_CADASTRAR = SUPER_JU_API_URL + "/produtos";
private static readonly string URL_PRODUTO_EDITAR = SUPER_JU_API_URL + "/produtos/{0}";
private static readonly string URL_PRODUTO_RELATORIO = SUPER_JU_API_URL + "/produtos/relatorio";

private static readonly string URL_ENTRADA_PRODUTO_PESQUISA = SUPER_JU_API_URL + "/produtos/entrada";
private static readonly string URL_ENTRADA_PRODUTO_POR_ID = SUPER_JU_API_URL + "/produtos/entrada/{0}";
private static readonly string URL_ENTRADA_PRODUTO_CADASTRAR = SUPER_JU_API_URL + "/produtos/entrada";

private static readonly string URL_PEDIDO_PESQUISA = SUPER_JU_API_URL + "/pedidos";
private static readonly string URL_PEDIDO_POR_ID = SUPER_JU_API_URL + "/pedidos/{0}";
private static readonly string URL_PEDIDO_CADASTRAR = SUPER_JU_API_URL + "/pedidos";
private static readonly string URL_PEDIDO_RELATORIO = SUPER_JU_API_URL + "/pedidos/relatorio";

private static readonly string URL_PEDIDO_FORMA_PAGAMENTO_BUSCA_TODOS = SUPER_JU_API_URL + "/pedidos/forma-pagamento";
```

Por exemplo, para realizar uma pesquisa de cliente:

```
public static List<ClienteResponse> ClientePesquisa(int? id, string nome)
{
    Dictionary<string, string> values = new Dictionary<string, string>();
    if (id != null)
    {
        values.Add("id", id.ToString());
    }
    if (!string.IsNullOrEmpty(nome))
    {
        values.Add("nome", nome);
    }

    HttpClient httpClient = GetDefaultHttpClient();
    var responseMessage = httpClient.GetAsync(URL_CLIENTE_PESQUISA + GetRequestParam(values)).Result;
    if (responseMessage.StatusCode == HttpStatusCode.NotFound)
    {
        return null;
    }
    responseMessage.EnsureSuccessStatusCode();
    return JsonConvert.DeserializeObject<List<ClienteResponse>>(responseMessage.Content.ReadAsStringAsync().Result);
}
```

Descrição dos Componentes

- **ConfigurationManager**: Utilizado para obter a URL da API a partir do web.config.
- **HttpClient**: Responsável por realizar as requisições HTTP (GET, POST, etc.) para a API.
- **JsonConvert**: Usado para converter o conteúdo da resposta JSON da API em objetos C#.

ADO.NET

O ADO.NET foi utilizado como tecnologia de acesso ao banco de dados. Ele fornece uma interface eficiente para a comunicação com o banco de dados SQL Server, permitindo a execução de comandos SQL diretamente, o que garante um controle refinado sobre as operações de leitura e escrita no banco.

Utilizado nas operações de CRUD (Create, Read, Update, Delete) tanto para os módulos de Produtos quanto para Clientes. A interação com o banco de dados é feita diretamente através de SqlConnection, SqlCommand, e SqlDataReader, permitindo o controle manual sobre as consultas SQL e transações.

```
string sql = @"SELECT Id, Nome, Descricao, Quantidade, ValorCusto, ValorVenda FROM PRODUTOS WHERE Id = @Id";

List<SqlParameter> parameters = new List<SqlParameter>();

parameters.Add(new SqlParameter("@Id", id));

using (SqlConnection connection = new SqlConnection(connectionString))
{
    try
    {
        connection.Open();
        SqlCommand command = new SqlCommand(sql, connection);

        command.Parameters.AddRange(parameters.ToArray());

        SqlDataReader dataReader = command.ExecuteReader();
        while (dataReader.Read())
        {
            Produto produto = new Produto()
            {
                Id = dataReader.GetInt32(dataReader.GetOrdinal("Id")),
                Nome = dataReader.GetString(dataReader.GetOrdinal("Nome")),
                Descricao = dataReader.GetString(dataReader.GetOrdinal("Descricao")),
                Quantidade = dataReader.GetInt32(dataReader.GetOrdinal("Quantidade")),
                ValorCusto = dataReader.GetDecimal(dataReader.GetOrdinal("ValorCusto")),
                ValorVenda = dataReader.GetDecimal(dataReader.GetOrdinal("ValorVenda"))
            };
            return produto;
        }
    }
```

Injeção de Dependências

A aplicação faz uso do padrão **Injeção de Dependências** na API para gerenciar a criação e o ciclo de vida das classes de serviço e repositório, facilitando a manutenção e escalabilidade do projeto.

A configuração foi realizada no projeto através da biblioteca nativa do .NET para injeção de dependências, que permite a vinculação dos **Services** e **Repositories** na inicialização da API. Isso garante a desacoplamento entre as camadas e facilita a realização de testes unitários.

Exemplo de configuração no Program.cs:

```
builder.Services.AddScoped<IClienteRepository, ClienteRepository>();
builder.Services.AddScoped<IEntradaProdutoItemRepository, EntradaProdutoItemRepository>();
builder.Services.AddScoped<IEntradaProdutoRepository, EntradaProdutoRepository>();
builder.Services.AddScoped<IFormaPagamentoRepository, FormaPagamentoRepository>();
builder.Services.AddScoped<IPedidoItemRepository, PedidoItemRepository>();
builder.Services.AddScoped<IPedidoRepository, PedidoRepository>();
builder.Services.AddScoped<IProdutoRepository, ProdutoRepository>();
builder.Services.AddScoped<IClienteService, ClienteService>();
builder.Services.AddScoped<IPedidoService, PedidoService>();
builder.Services.AddScoped<IProdutoService, ProdutoService>();
```

Testes Unitários

A camada de serviços da API foi testada utilizando **xUnit** e **Moq**. Esses testes garantem que as regras de negócio foram implementadas corretamente e funcionam como esperado, sem a necessidade de uma conexão real com o banco de dados ou outras dependências externas.

xUnit

O **xUnit** é um framework de testes para .NET, utilizado para criar e rodar os testes unitários. Ele oferece uma maneira eficiente e clara de verificar o comportamento do código.

Moq

O **Moq** foi utilizado para criar *mocks* nas classes de repository ao testar os serviços. Com o uso de *mocks*, foi possível simular o comportamento dos repositorys, testando isoladamente a lógica de negócio na camada de serviço sem a necessidade de realizar chamadas na camada de repository aonde é acessado o banco de dados.

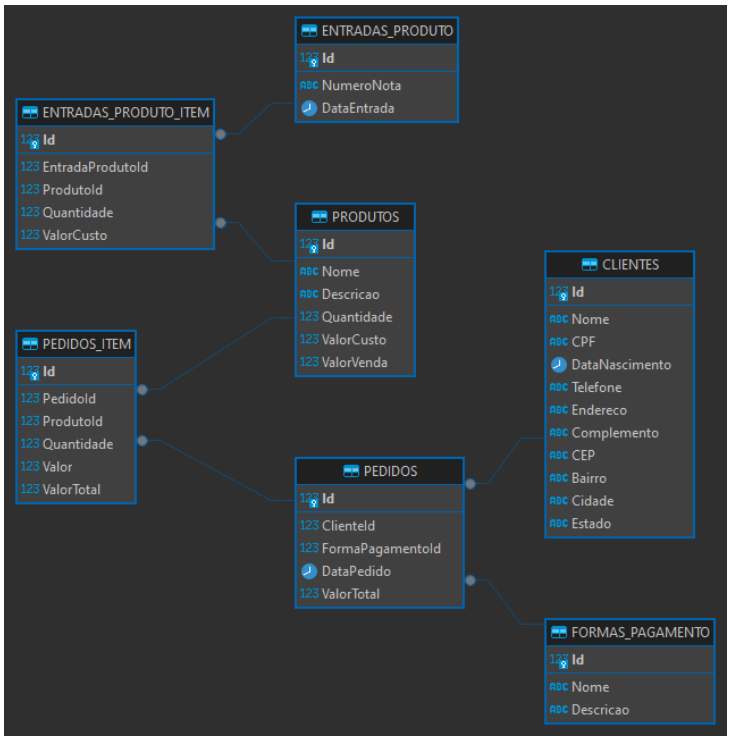
Exemplo de um teste com **Moq** e **xUnit:**

```
[Fact]
0 references
public void Retorna_Sucesso_Cliente_Busca_Por_Id()
{
    //Arrange
    Mock<IClienteRepository> clienteRepositoryMock = new Mock<IClienteRepository>();
    Cliente cliente = new Cliente
    {
        Id = 1,
        Nome = "Teste 1",
        CPF = "11111111111",
        DataNascimento = DateTime.Now,
        Telefone = "34988334833",
        Endereco = "Rua Teste, 33",
        Complemento = null,
        CEP = "44333111",
        Bairro = "Bairro Teste",
        Cidade = "Cidteste",
        Estado = "MG"
    };
    clienteRepositoryMock.Setup(repo => repo.BuscaPorId(It.IsAny<int>())).Returns(value: cliente);
    ClienteService clienteService = new ClienteService(clienteRepositoryMock.Object);

    //Act
    var response = clienteService.BuscarPorId(1);

    //Assert
    Assert.NotNull(response);
}
```

Diagrama do Sistema



Pacotes NuGet Adicionados

Os seguintes pacotes NuGet foram utilizados no projeto:

- **Newtonsoft.Json**: Para manipulação de JSON.
- **Microsoft.AspNet.WebApi.Client**: Para suporte a APIs REST.
- **xunit**: Para testes unitários.
- **Moq**: Para mocking em testes.

Bibliotecas

- **jQuery**: A biblioteca jQuery, versão 3.7.0.
- **jquery.inputmask**: Este plugin permite aplicar máscaras de entrada em campos de formulário, melhorando a experiência do usuário.
- **jquery.maskMoney**: Utilizado para formatar valores monetários em campos de entrada, garantindo que os usuários insiram dados no formato correto.

Conclusão

O projeto **SuperJU** integra um sistema completo de controle de estoque e vendas com uma interface WebForms que se comunica eficientemente com uma API RESTful através de requisições HTTP. A configuração flexível da URL da API no web.config, juntamente com o uso de **HttpClient**, permite uma arquitetura limpa e escalável. Além disso, a API foi testada utilizando **xUnit** e **Moq**, garantindo que a lógica de negócio está corretamente implementada e funcionando de forma confiável.

GitHub

Todo o código do projeto pode ser encontrado em: GitHub do SuperJU.