

ProjetoEDOO

Sistema de Gerenciamento de Restaurante

Autores: Carlos Eduardo Falcão Teixeira, João Victor Acioly

Link github: <https://github.com/jvacioly/ProjetoEDOO>

1. Introdução

O projeto consiste em um sistema integrado que une **gestão de estoque** e **delivery** para restaurantes. Ele permite que clientes realizem pedidos online, acompanhem o status dos pedidos e que os administradores gerenciem o fluxo de pedidos e controle de estoque em tempo real.

2. Funcionalidades

- **Para Clientes:**

- ✓ Visualização do cardápio online
- ✓ Seleção de pratos com quantidade e observações
- ✓ Acompanhamento do status do pedido e histórico de pedidos

- **Para Restaurantes:**

- ✓ Recebimento e gestão dos pedidos
- ✓ Atualização do status dos pedidos (esperando, preparando, entregue, etc.)
- ✓ Controle automatizado do estoque
- ✓ Fluxo de caixa integrado ao sistema de pedidos

3. Tecnologias Utilizadas

Front-end: HTML, CSS, JavaScript

Back-end: C++ com JSON para armazenamento de dados

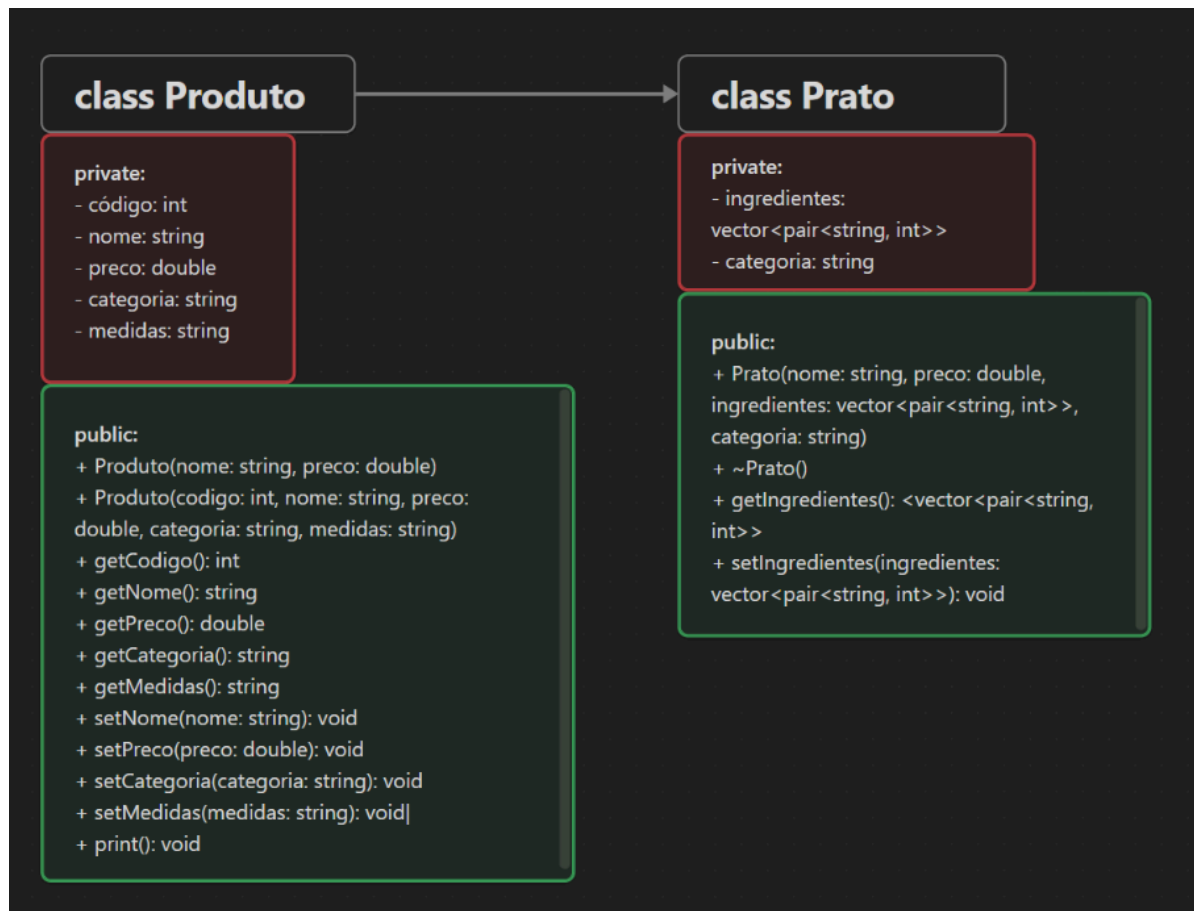
Comunicação: WebSockets (CivetWeb) para atualização em tempo real

4. Estrutura do Projeto

📁 **/backend** → Lógica do servidor e gerenciamento de pedidos

O backend do projeto foi planejado para ser modular e eficiente, com cada classe tendo sua responsabilidade clara. está estruturado a partir de quatro classes principais: Produto, Prato, Restaurante e Pedido. Cada uma delas tem uma função específica no sistema de gerenciamento do restaurante idealizado. A classe Produto representa os itens

do estoque, como ingredientes ou bebidas, com atributos como nome, preço e categoria. Ela serve de base para a classe Prato, que herda de Produto e adiciona as funcionalidades de gerenciar ingredientes, representando os pratos do menu.



A classe Pedido, por sua vez, representa os pedidos feitos pelos clientes. Ela armazena informações como os pratos solicitados, quantidades, endereço de entrega, forma de pagamento e status do pedido. O valor total é calculado automaticamente com base nos itens, e o pedido pode passar por diferentes status, como "confirmar", "preparando", "entregue" ou "cancelado". O sistema também permite adicionar observações e gerenciar os pratos do pedido.



Por fim, a classe Restaurante, a principal classe do sistema, ela integra todas as funcionalidades e se conecta ao banco de dados. Ela gerencia o estoque de produtos, o menu de pratos, os pedidos dos clientes e o fluxo de caixa. Os dados são armazenados em arquivos JSON, como estoque.json, pedidos.json e fluxo.json, garantindo persistência e facilidade na recuperação das informações. O restaurante também possui métodos para adicionar, editar e remover itens do estoque, além de registrar e acompanhar pedidos.


class Restaurante

private:

- estoque: json
- pedidos: json
- fluxo: json
- nome: string
- endereco: vector< string>
- contato: string
- descricao: string
- menu: vector

public:

- + Restaurante(nome: string, menu: vector)
- + Restaurante(menu: vector)
- + Restaurante(nome: string, endereco: vector, contato: string, descricao: string, menu: vector)
- + ~Restaurante()
- + getNome(): string
- + getMenu(): vector
- + carregarEstoque(): void
- + salvarEstoque() const: void
- + addEstoque(produto: Produto&, quantidade: int): void
- + editEstoque(codigoProduto: int, nome: string, categoria: string, medida: string, quantidade: double, remover: bool): void
- + apagarItem(codigoProduto: int): bool
- + checarEstoque(pedido: Pedido&): bool
- + mostrarEstoque() const: void
- + carregarPedidos(): void
- + salvarPedidos() const: void
- + registrarPedido(pedido: Pedido&): void
- + prepararPedido(IDpedido: string&): void
- + enviarPedido(IDpedido: string&): void
- + cancelarPedido(IDpedido: string&): void
- + finalizarPedido(IDpedido: string&): void
- + carregarFluxo(): void
- + salvarFluxo() const: void
- + registrarCompra(valor: double): void
- + registrarVenda(valor: double): void
- + adicionarCaixa(valor: double): void
- + mostrarMenu() const: void

 /database → Arquivos JSON para armazenar estoque e pedidos

```

1  {
2      "70193839": {
3          "cep": "20202020",
4          "endereco": "rua vinte",
5          "forma_pagamento": "Dinheiro",
6          "horario_pedido": "23:51:54",
7          "itens_do_pedido": [
8              {
9                  "prato_nome": "Bruschetta",
10                 "preco_unitario": 18.99,
11                 "quantidade": 1
12             },
13             {
14                 "prato_nome": "Pizza Margherita",
15                 "preco_unitario": 49.99,
16                 "quantidade": 1
17             }
18         ],
19         "numero": "201",
20         "observacao": "",
21         "preco_total": 87.97,
22         "status": "finalizado",
23         "tipo_endereco": "residencia"
24     }
25 }
26

```

Formato que os objetos de pedidos são armazenados no arquivo pedidos.json, ele contém um atributo chamado itens_do_pedido que armazena um array de objetos da classe prato.

```

1  {
2      "Despesas": 14168.0,
3      "Lucro": -14080.03,
4      "Ranking de Pedidos": {},
5      "Receita": 87.97
6  }
7

```

Formato que os atributos do fluxo de caixa são armazenados no arquivo fluxo.json.

```

1      {
2          "Alface": {
3              "categoria": "Legumes e Verduras",
4              "codigo": 1526,
5              "medida": "g",
6              "preco": 0.04,
7              "quantidade": 1000
8          },
9          "Alho": {
10             "categoria": "Temperos e Especiarias",
11             "codigo": 1692,
12             "medida": "Un",
13             "preco": 0.2,
14             "quantidade": 60
15         },
16         "Arroz": {
17             "categoria": "Grãos",
18             "codigo": 2304,
19             "medida": "g",

```

Formato que os objetos da classe produto são armazenados no arquivo estoque.json

/Comunicação: Conexão do backend e frontend através de WebSocket (CivetWeb)

Para garantir uma atualização em tempo real do conteúdo do estoque e pedidos, optamos por implementar um servidor WebSocket. Essa escolha permite uma comunicação bidirecional entre o frontend e o backend, onde ambos os lados podem enviar e receber mensagens instantaneamente, diferente da requisição HTTP que é stateless. O servidor, desenvolvido em C++, opera em um loop contínuo, aguardando mensagens do frontend para estabelecer e manter a conexão. Quando a conexão é estabelecida, o servidor e o cliente podem trocar dados de forma eficiente e em tempo real.

Configuração do Servidor WebSocket

O servidor foi configurado utilizando a biblioteca **CivetWeb**, uma solução leve e eficiente para criação de servidores web e WebSocket. No server.cpp a função `setup_websocket_server()` é responsável por inicializar o servidor e configurar os handlers para lidar com eventos específicos da conexão WebSocket. O servidor escuta na porta ``8000`` e define a rota ``/ws`` para gerenciar as conexões WebSocket.

Os handlers são funções callback que são chamadas automaticamente pelo servidor WebSocket quando ocorrem eventos específicos. A comunicação é gerenciada por quatro handlers principais:

1. `websocket_connect_handler`: Chamado quando uma nova conexão WebSocket é estabelecida.
2. `websocket_ready_handler`: Chamado quando a conexão está pronta para comunicação.
3. `websocket_data_handler`: Chamada quando recebe uma mensagem. Esta função processa as mensagens recebidas e envia respostas ao cliente.
4. `websocket_close_handler`: Chamado quando a conexão é fechada

```
// Função para configurar e iniciar o servidor WebSocket
void setup_websocket_server() {
    const char *options[] = {
        "listening_ports", "8000",
        nullptr
    };

    struct mg_callbacks callbacks;
    memset(&callbacks, 0, sizeof(callbacks));

    struct mg_context *ctx = mg_start(&callbacks, user_data: nullptr, options);
    if (!ctx) {
        cerr << "Falha ao iniciar o servidor" << endl;
        return;
    }

    // Configurando a rota WebSocket para "/ws"
    mg_set_websocket_handler(ctx, uri: "/ws",
        websocket_connect_handler, // Quando conecta
        websocket_ready_handler,   // Quando pronto
        websocket_data_handler,    // Quando recebe dados
        websocket_close_handler,   // Quando desconecta
        cbdata: nullptr);

    cout << "Servidor rodando em http://localhost:8000" << endl;
    cout << "WebSocket disponível em ws://localhost:8000/ws" << endl;

    // Mantém o servidor rodando
    while (true) {
        this_thread::sleep_for(chrono::seconds(1));
    }

    mg_stop(ctx);
}
```

Funcionamento e Tratamento de Dados do WebSocket

No código, o WebSocket é configurado para enviar e receber mensagens de texto ('MG_WEBSOCKET_OPCODE_TEXT'), que são tratadas como strings JSON, através de uma função da biblioteca CivetWeb chamada `mg_websocket_write`

```
mg_websocket_write(conn, MG_WEBSOCKET_OPCODE_TEXT, data: json_data.c_str(), data_len: json_data.size());
```

O tratamento das mensagens é realizado no `websocket_data_handler`. As mensagens recebidas são interpretadas através de flags procurando palavras chaves e assim as respostas são enviadas de volta ao cliente. O formato JSON é utilizado para estruturar os dados, utilizando a biblioteca **nlohmann/json** para manipulação.

- Solicitar Estoque: Envia os dados do estoque em formato JSON.
- Solicitar Fluxo: Envia os dados do fluxo de pedidos.
- Solicitar Pedidos: Envia os dados dos pedidos atuais.
- Editar: Atualiza o estoque com base nas informações recebidas.
- Adicionar: Adiciona ou remove itens do estoque.
- Pedido: Registra um novo pedido com base nos dados recebidos.
- Alterar Status: Atualiza o status de um pedido (preparando, caminho, cancelado, finalizado).



```
// Handler para mensagens recebidas via WebSocket
int websocket_data_handler(mg_connection *conn, int bits, char *data, size_t data_len, void *cbdata) {
    if (data && data_len > 0) {
        string request(data, data_len);

        // AÇÕES DE ESTOQUE
        if (request == "Solicitar Estoque") {
            // Envia os dados do estoque para o cliente
            send_estoque_json(conn);
        }
        else if (request == "Solicitar Fluxo") {
            // Envia os dados do fluxo para o cliente
            send_fluxo_json(conn);
        }
        else if (request == "Solicitar Pedidos") {
            // Envia os dados do fluxo para o cliente
            send_pedidos_json(conn);
        }
        else if (request.find("@editar") != string::npos) {
            json jsonInfos = json::parse([&] request);
            string nome = jsonInfos.value(key: "nome", default_value: "");
            string categoria = jsonInfos.value(key: "categoria", default_value: "");
            string medida = jsonInfos.value(key: "medida", default_value: "");
            int codigoProduto = jsonInfos.contains(key: "codigo") ? stoi(str(jsonInfos["codigo"]).get<string>()) : 0;
            string acao;
            try {
                acao = jsonInfos.value(key: "add/rem", default_value: "");
            } catch (const json::exception &e) {
                acao = "";
            }
            if (acao == "adicionar") {
                double quantidade = jsonInfos.contains(key: "quantAdicionada") ? stod(str(jsonInfos["quantAdicionada"]).get<string>()) : 0.0;
                restaurante->editEstoque(codigoProduto, nome, categoria, medida, quantidade, remover: false);
            }
            else if (acao == "remover") {
                double quantidade = jsonInfos.contains(key: "quantRemover") ? stod(str(jsonInfos["quantRemover"]).get<string>()) : 0.0;
            }
        }
    }
}
```

E portanto, as funções `send_pedidos_json`, `send_fluxo_json` e `send_estoque_json` são responsáveis por enviar dados ao cliente. Elas leem os arquivos JSON (`pedidos.json`, `fluxo.json` e `estoque.json`), e os convertem em strings e enviam via WebSocket pela a função mencionada antes (`mg_websocket_write`).

```
// Função que envia os dados de estoque em formato JSON
void send_estoque_json(struct mg_connection *conn) {
    string caminho_arquivo = BASE_DIR + "estoque.json";
    ifstream file(caminho_arquivo);
    json estoque;
    file >> estoque;

    string json_data = estoque.dump();
    mg_websocket_write(conn, MG_WEBSOCKET_OPCODE_TEXT, data: json_data.c_str(), data_len: json_data.size());
}
```

 **/frontend** → Código do site e interface do usuário

Para o frontend vamos tomar como exemplo o Javascript do estoque.

Primeiramente, a conexão é salva como um novo objeto WebSocket apontando para a porta 'ws://localhost:8000/ws' na constante `socket`.

```
// Criando a conexão WebSocket com o servidor C++
const socket :WebSocket = new WebSocket( url: 'ws://localhost:8000/ws');
```

O objeto tem um método chamado `.onopen` que é chamado quando a conexão WebSocket é estabelecida. Assim, mostrando no console a mensagem confirmando a conexão e em seguida através do método `.send`, duas strings são enviadas solicitando uma o estoque e outra o fluxo. Isso permite que a página seja carregada com os dados iniciais necessários.

```
// Função para quando a conexão for aberta
socket.onopen = function() :void {
    console.log("Conexão WebSocket estabelecida");
    // Não é necessário enviar nada no momento, mas pode ser feito para solicitar dados
    socket.send( data: 'Solicitar Estoque');
    socket.send( data: 'Solicitar Fluxo');
};
```

E com o método `.onmessage`, que é acionado quando recebe uma mensagem do servidor, os dados da mensagem são processados usando `JSON.parse` para converter em um objeto Javascript. Posteriormente, uma função criada para processar estes dados será utilizada. Essa função irá apagar os elementos atualmente presentes na tela e transformar o objeto recebido, dependendo de qual arquivo JSON foi enviado, em um array de objetos. Em seguida utilizando o método `forEach` do Javascript, os elementos serão recriados com as informações contidas em cada objeto do array.

```
// Função para tratar mensagens recebidas do servidor (JSON com dados do estoque)
socket.onmessage = function(event :MessageEvent ) :void {
    // Convertendo a string JSON para um objeto JavaScript
    const dados = JSON.parse(event.data);

    if (dados.hasOwnProperty( v: "Despesas" )) {
        processarFluxo(dados)
    } else {
        processarEstoque(dados)
    }
}
```

Já para enviar os dados, transformamos os dados dos formulários HTML em um objeto e depois transformamos este objeto em uma string JSON. Uma vez com a string JSON pronta, enviamos esta string pelo método `.send` novamente.

```
const jsonData :string = JSON.stringify(formObject)

socket.send(jsonData)
```

Exemplo de objeto com o dados do formulário ao adicionar item ao estoque:

```
▼ Object 1
  acao: "adicionar"
  categoria: "Frutas"
  medida: "Un"
  nome: "Maça"
  preco: "0.99"
  quantM: "20"
  ► [[Prototype]]: Object
```

5. Atividade de cada membro

Carlos Eduardo Falcão Teixeira:

- Responsável por desenvolver toda interface, criar e configurar o servidor WebSocket com a biblioteca Civetweb e parcialmente responsável pela conexão entre frontend e backend com a troca de strings JSON.
- Total de 41 commits
- 5.736 linhas adicionadas
- 591 linhas removidas

João Victor Acioly:

- Responsável por desenvolver o esquema de classes, arquivos C++ e a interação com o banco de dados json, incluindo a implementação das funcionalidades e a integração entre os componentes. Também foi responsável pela conexão entre frontend e backend.
- Total de 29 commits
- 27.919 linhas adicionadas
- 1.429 linhas removidas