Adicionando Funcionalidades Dinâmicas ao Cardápio e Agendamento com Banco de Dados (.NET MVC)

Olá! Nas etapas anteriores, planejamos os modelos de dados, a autenticação e criamos a estrutura visual das páginas principais. Agora, vamos dar vida ao cardápio e ao sistema de agendamento, conectando-os ao banco de dados MySQL que você configurará localmente. Usaremos o Entity Framework Core (EF Core) como nosso ORM (Object-Relational Mapper) para facilitar a comunicação entre nosso código C# e o banco de dados.

Pré-requisitos: * Você já deve ter os modelos `Usuario`, `Produto`, `Agendamento` e `ItemPedido` definidos (conforme `modelos_de_dados.md`). * O MySQL Server deve estar instalado e rodando. * O pacote NuGet `Pomelo.EntityFrameworkCore.MySql` (ou outro conector MySQL para EF Core) e `Microsoft.EntityFrameworkCore.Tools` (para migrations) devem estar instalados no projeto. `bash dotnet add package Pomelo.EntityFrameworkCore.MySql dotnet add package Microsoft.EntityFrameworkCore.Tools`

1. Configurando o DbContext

O `DbContext` é a ponte entre seus modelos C# e o banco de dados. Ele gerencia a conexão, rastreia as alterações nos objetos e salva essas alterações no banco.

- **1.1. Crie a classe `ApplicationDbContext.cs` (ex: na pasta `Data`):**

  ```csharp // Data/ApplicationDbContext.cs using Microsoft.EntityFrameworkCore; // Certifique-se de que os caminhos para seus modelos estão corretos // Exemplo: using SeuProjeto.Models;

  public class ApplicationDbContext : DbContext { public ApplicationDbContext(DbContextOptions options) : base(options) { }

  ```
  public DbSet<Usuario> Usuarios { get; set; }
  public DbSet<Produto> Produtos { get; set; }
  public DbSet<Agendamento> Agendamentos { get; set; }
  public DbSet<ItemPedido> ItensPedido { get; set; }
  ```
  ```

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    // Configurações adicionais do modelo, se necessário (chaves compostas, relações, etc.)
    // Exemplo: Chave primária composta para ItemPedido (se não usar ID autoincremental para ItemPedido)
    // modelBuilder.Entity<ItemPedido>()
    //    .HasKey(ip => new { ip.AgendamentoId, ip.ProdutoId });
    // No nosso caso, ItemPedido já tem um Id próprio, então não é estritamente necessário aqui,
    // mas é bom saber que você pode definir chaves compostas.

    // Configurar precisão para propriedades decimais (Preço em Produto, PrecoUnitario e ValorTotal em Agendamento)
    modelBuilder.Entity<Produto>()
        .Property(p => p.Preco)
        .HasColumnType("decimal(18,2)");

    modelBuilder.Entity<ItemPedido>()
        .Property(ip => ip.PrecoUnitario)
        .HasColumnType("decimal(18,2)");

    modelBuilder.Entity<Agendamento>()
        .Property(a => a.ValorTotal)
        .HasColumnType("decimal(18,2)");

    // Definindo relações (EF Core geralmente infere muitas delas, mas pode ser explícito)
    // Relação Usuario -> Agendamentos (Um-para-Muitos)
    modelBuilder.Entity<Usuario>()
        .HasMany(u => u.Agendamentos)
        .WithOne(a => a.Usuario)
        .HasForeignKey(a => a.UsuarioId)
        .OnDelete(DeleteBehavior.Cascade); // Ou Restrict, SetNull dependendo da sua regra de negócio

    // Relação Agendamento -> ItensPedido (Um-para-Muitos)
    modelBuilder.Entity<Agendamento>()
        .HasMany(a => a.ItensPedido)
        .WithOne(ip => ip.Agendamento)
        .HasForeignKey(ip => ip.AgendamentoId)
        .OnDelete(DeleteBehavior.Cascade);

    // Relação Produto -> ItensPedido (Um-para-Muitos)
    modelBuilder.Entity<Produto>()
        .HasMany(p => p.ItensPedido)
        .WithOne(ip => ip.Produto)
        .HasForeignKey(ip => ip.ProdutoId)
        .OnDelete(DeleteBehavior.Restrict); // Impedir exclusão de produto se
```

```
    estiver em um pedido
    }

}
```

- **1.2. Configure a String de Conexão em `appsettings.json`:**

  `json // appsettings.json { // ... outras configurações ... "ConnectionStrings": { "DefaultConnection": "Server=localhost;Port=3306;Database=NomeDoSeuBancoDeDados;Uid=seu_usuario_mysq`
  Importante: Substitua `NomeDoSeuBancoDeDados`, `seu_usuario_mysql` e `sua_senha_mysql` pelos seus dados reais. Crie este banco de dados no MySQL Workbench ou via linha de comando antes de prosseguir.

- **1.3. Registre o `DbContext` em `Program.cs`:**

  ```csharp // Program.cs // ... outras usings ... using Microsoft.EntityFrameworkCore; // using SeuProjeto.Data; // Caminho para seu ApplicationDbContext

  var builder = WebApplication.CreateBuilder(args);

  // Adicionar serviços ao contêiner. var connectionString = builder.Configuration.GetConnectionString("DefaultConnection"); builder.Services.AddDbContext(options => options.UseMySql(connectionString, ServerVersion.AutoDetect(connectionString)));

  // ... resto da configuração de autenticação, controllers, etc. ...

  var app = builder.Build(); // ... resto do pipeline ... app.Run(); ```

- **1.4. Crie as Migrations e Atualize o Banco:** Migrations são como um controle de versão para o seu esquema de banco de dados.

  1. Abra o terminal na pasta raiz do seu projeto.
  2. Execute: `dotnet ef migrations add InitialCreate` (ou outro nome para a migration inicial). Isso criará uma pasta `Migrations` no seu projeto com o código para criar as tabelas.
  3. Execute: `dotnet ef database update` Isso aplicará a migration ao seu banco de dados MySQL, criando as tabelas.
  ```

2. Funcionalidade do Cardápio (`CardapioController`)

Agora vamos modificar o `CardapioController` para buscar os produtos do banco.

- **2.1. Injete o `ApplicationDbContext` no `CardapioController.cs`:**

  ```csharp // Controllers/CardapioController.cs using Microsoft.AspNetCore.Mvc; using Microsoft.EntityFrameworkCore; // Para ToListAsync() e Include() // using SeuProjeto.Data; // Seu DbContext // using SeuProjeto.Models; // Seus modelos, incluindo Produto using System.Collections.Generic; // Para List using System.Threading.Tasks; // Para Task

  public class CardapioController : Controller { private readonly ApplicationDbContext _context; // Injete seu DbContext

  ```
  public CardapioController(ApplicationDbContext context) // Construtor para injeção de dependência
  {
      _context = context;
  }

  // GET: Cardapio ou Cardapio/Index
  public async Task<IActionResult> Index()
  {
      List<Produto> produtos = await _context.Produtos.ToListAsync(); // Busca todos os produtos do banco
      return View(produtos); // Passe os produtos para a View
  }

  // GET: Cardapio/Detalhes/5 (Opcional)
  public async Task<IActionResult> Detalhes(int? id)
  {
      if (id == null)
      {
          return NotFound(); // Retorna erro 404 se o ID não for fornecido
      }

      // Busca o produto pelo ID. Include() pode ser usado para carregar dados relacionados se houver.
      var produto = await _context.Produtos
          // .Include(p => p.AlgumaPropriedadeRelacionada) // Exemplo se Produto tivesse relação
          .FirstOrDefaultAsync(m => m.Id == id);

      if (produto == null)
      {
          return NotFound(); // Retorna erro 404 se o produto não for encontrado
      }
  ```
  ```

```
    return View(produto); // Passa o produto encontrado para a View Detalhes.cshtml
}
```

} `` **Explicação:** * O ApplicationDbContext é injetado através do construtor. O sistema de injeção de dependência do .NET Core cuida de fornecer a instância. * _context.Produtos.ToListAsync() : Busca todos os registros da tabela Produtos de forma assíncrona e os converte para uma lista de objetos Produto . * A View Views/Cardapio/Index.cshtml que criamos antes já está preparada para receber IEnumerable e exibir os dados. * A action Detalhes é um exemplo de como você pode buscar um item específico. Você precisaria criar uma View Views/Cardapio/Detalhes.cshtml` para exibir as informações desse produto.

3. Funcionalidades de Agendamento (`AgendamentoController`)

Esta é a parte mais complexa, pois envolve o usuário logado, seleção de múltiplos produtos e gravação de múltiplos registros relacionados.

- **3.1. Injete o `ApplicationDbContext` no `AgendamentoController.cs`:** Assim como fizemos no `CardapioController`.

  ```csharp // Controllers/AgendamentoController.cs using Microsoft.AspNetCore.Mvc; using Microsoft.AspNetCore.Authorization; using Microsoft.EntityFrameworkCore; using System.Security.Claims; // Para pegar o ID do usuário logado // using SeuProjeto.Data; // using SeuProjeto.Models; // E ViewModels using System.Threading.Tasks; using System.Linq; using System.Collections.Generic; using System;

  [Authorize] // Exige que o usuário esteja logado public class AgendamentoController : Controller { private readonly ApplicationDbContext _context;

  ```
  public AgendamentoController(ApplicationDbContext context)
  {
      _context = context;
  }
  ```

  ```

- **3.2. Modifique a Action `Criar` (GET) para carregar produtos:**

  ```csharp // GET: Agendamento/Criar public async Task Criar() { // Carregar a lista de produtos para o usuário selecionar ViewBag.ProdutosDisponiveis = await _context.Produtos.Where(p => p.Nome != null).ToListAsync(); // Exemplo de filtro simples

  ```
  var model = new AgendamentoViewModel(); // ViewModel que criamos antes
  model.DataHoraAgendamento =
  DateTime.Now.Date.AddDays(1).AddHours(12); // Sugestão: próximo dia ao
  meio-dia
  return View(model);
  ```

  } ``A View `Views/Agendamento/Criar.cshtml` já usa `ViewBag.ProdutosDisponiveis` para listar os produtos.

- **3.3. Modifique a Action `Criar` (POST) para salvar o agendamento:**

  ```csharp // POST: Agendamento/Criar [HttpPost] [ValidateAntiForgeryToken] public async Task Criar(AgendamentoViewModel viewModel) { string userId = User.FindFirstValue(ClaimTypes.NameIdentifier); // Pega o ID do usuário logado (Google ID)

  ```
  // Validação adicional: Verificar se o usuário existe no nosso banco (opcional, mas bom)
  var usuario = await _context.Usuarios.FindAsync(userId);
  if (usuario == null)
  {
      ModelState.AddModelError("", "Usuário não encontrado. Por favor, faça login novamente.");
  }

  // Validar se pelo menos um item foi selecionado
  if (viewModel.ItensSelecionados == null || !
  viewModel.ItensSelecionados.Any(i => i.Quantidade > 0))
  {
      ModelState.AddModelError("", "Você precisa selecionar pelo menos um item para o pedido.");
  }

  if (ModelState.IsValid && usuario != null)
  {
      Agendamento novoAgendamento = new Agendamento
      {
          UsuarioId = userId,
          Usuario = usuario, // Associar o objeto usuário
          DataHoraAgendamento = viewModel.DataHoraAgendamento,
  ```

```
        Observacoes = viewModel.Observacoes,
        Status = "Pendente", // Status inicial
        DataHoraCriacao = DateTime.UtcNow,
        ValorTotal = 0 // Será calculado abaixo
    };

    foreach (var itemSelecionado in viewModel.ItensSelecionados.Where(i =>
i.Quantidade > 0))
    {
        var produto = await
_context.Produtos.FindAsync(itemSelecionado.ProdutoId);
        if (produto != null)
        {
            var itemPedido = new ItemPedido
            {
                Agendamento = novoAgendamento, // EF Core associará o
AgendamentoId automaticamente
                ProdutoId = produto.Id,
                Produto = produto, // Associar o objeto produto
                Quantidade = itemSelecionado.Quantidade,
                PrecoUnitario = produto.Preco // Preço no momento da compra
            };
            novoAgendamento.ItensPedido.Add(itemPedido);
            novoAgendamento.ValorTotal += (produto.Preco *
itemSelecionado.Quantidade);
        }
        else
        {
            // Tratar caso um produto selecionado não seja encontrado
(improvável se a lista estiver correta)
            ModelState.AddModelError("", $"Produto com ID
{itemSelecionado.ProdutoId} não encontrado.");
            // Recarregar produtos e retornar à view
            ViewBag.ProdutosDisponiveis = await _context.Produtos.ToListAsync();
            return View(viewModel);
        }
    }

    _context.Agendamentos.Add(novoAgendamento); // Adiciona o
agendamento e seus itens (devido à navegação)
    await _context.SaveChangesAsync(); // Salva tudo no banco de dados

    TempData["Sucesso"] = "Agendamento criado com sucesso!";
    return RedirectToAction(nameof(MeusAgendamentos));
}

// Se chegou aqui, algo falhou, recarregue o formulário com os erros
ViewBag.ProdutosDisponiveis = await _context.Produtos.ToListAsync(); //
Recarregar produtos
return View(viewModel);
```

} `` **Explicação Chave:** * User.FindFirstValue(ClaimTypes.NameIdentifier) : Pega o ID do usuário logado (que, na nossa configuração de autenticação Google, é o ID único fornecido pelo Google). * Criamos o Agendamento e, em seguida, iteramos sobre viewModel.ItensSelecionados (que o JavaScript da View preencheu). * Para cada item selecionado, buscamos o Produto no banco para garantir que ele existe e para pegar o preço atual. * Criamos um ItemPedido e o adicionamos à coleção novoAgendamento.ItensPedido . * _context.Agendamentos.Add(novoAgendamento); e await _context.SaveChangesAsync(); : O EF Core é inteligente o suficiente para, ao adicionar o Agendamento , também identificar e adicionar os ItemPedido relacionados que estão na coleção ItensPedido`, devido às propriedades de navegação que definimos nos modelos.

- **3.4. Modifique a Action `MeusAgendamentos` para buscar do banco:**

```csharp // GET: Agendamento/MeusAgendamentos public async Task MeusAgendamentos() { string userId = User.FindFirstValue(ClaimTypes.NameIdentifier); var agendamentosDoUsuario = await _context.Agendamentos .Where(a => a.UsuarioId == userId) .Include(a => a.ItensPedido) // Inclui os ItemPedido relacionados .ThenInclude(ip => ip.Produto) // Para cada ItemPedido, inclui o Produto relacionado .OrderByDescending(a => a.DataHoraCriacao) .ToListAsync();

    return View(agendamentosDoUsuario);
```

} `` **Explicação:** * .Include(a => a.ItensPedido) : Diz ao EF Core para carregar também a coleção de ItemPedido para cada Agendamento . * .ThenInclude(ip => ip.Produto) : Para cada ItemPedido carregado, também carrega o objeto Produto associado. Isso é crucial para que na View MeusAgendamentos.cshtml você possa exibir o nome do produto (ex: ip.Produto.Nome). * A View Views/Agendamento/ MeusAgendamentos.cshtml` que criamos antes já está preparada para exibir essas informações.

4. Testando as Funcionalidades

1. **Verifique o Banco:** Após rodar `dotnet ef database update` , confira no MySQL Workbench se as tabelas `Usuarios` , `Produtos` , `Agendamentos` e `ItensPedido` foram criadas corretamente.

2. **Popule a Tabela `Produtos` :** Adicione alguns produtos manualmente no banco de dados através do MySQL Workbench para ter o que listar no cardápio.
    - Exemplo de INSERT: `INSERT INTO Produtos (Nome, Descricao, Preco, UrlImagem, Categoria) VALUES ('Pizza Margherita', 'Molho de tomate, mussarela e manjericão', 30.00, '/images/pizza.jpg', 'Pizzas');`
3. **Execute o Aplicativo:**
4. **Cardápio:** Navegue até a página de cardápio. Você deve ver os produtos que inseriu no banco.
5. **Login:** Faça login com sua conta Google.
6. **Agendamento:**
    - Vá para a página de "Agendar Pedido".
    - Selecione alguns produtos e quantidades.
    - Escolha data/hora e adicione observações.
    - Clique em "Finalizar Agendamento".
    - Você deve ser redirecionado para "Meus Agendamentos" com uma mensagem de sucesso.
7. **Meus Agendamentos:** Verifique se o agendamento recém-criado aparece na lista, com os itens corretos.
8. **Verifique o Banco Novamente:** Confira as tabelas `Agendamentos` e `ItensPedido` para ver se os dados foram salvos corretamente.

Este é um passo grande e crucial! Teste cada parte com calma. Se encontrar erros, o console de depuração do Visual Studio (ou os logs do terminal se estiver usando `dotnet run`) e as mensagens de erro do ASP.NET Core serão seus melhores amigos para identificar problemas.

Lembre-se que a interface do usuário para selecionar produtos no agendamento (`Criar.cshtml`) pode ser melhorada com mais JavaScript para uma experiência mais rica (ex: atualizar o total dinamicamente sem precisar de um script tão verboso, adicionar/remover itens de forma mais visual). O exemplo fornecido é funcional, mas básico.