Documentação para Desenvolvedores do Projeto Customer

Parte 1: Arquitetura Front-End

Introdução

A camada de front-end será responsável pela interface com o usuário, utilizando Razor Pages, HTML, CSS e JavaScript para criar uma experiência interativa e funcional. O padrão MVC será utilizado para separar as responsabilidades:

- **Model**: Define dados e regras de negócios.
- **View**: Exibe a interface ao usuário.
- **Controller**: Gerencia interações do usuário, processando entradas e atualizando Views ou Models conforme necessário.

Requisitos Funcionais (Front-End)

1. **Renderizar páginas** para cadastro, edição e listagem de clientes e logradouros.
2. **Consumir a API RESTful** para criar, atualizar, visualizar e excluir registros.
3. **Validar dados** no front-end antes de enviá-los à API.

Informações da API que será consumida pelo front-end

- **Título**: CustomerRegistrationApi
- **Versão**: 1.0

Endpoints

- **/api/Auth/login (POST)**:
  - **Descrição**: Realiza login.
  - **Request Body**: JSON com `username(adm)` e `password(123)`.
  - **Resposta**: Código 200 OK em caso de sucesso.
- **/api/Customer (POST)**:
  - **Descrição**: Cria um novo cliente.
  - **Request Body**: JSON com `name`, `email`, `addresses`, e `companyFile`.
  - **Resposta**: Código 200 OK em caso de sucesso.
- **/api/Customer (GET)**:
  - **Descrição**: Retorna todos os clientes.
  - **Resposta**: Código 200 OK em caso de sucesso.
- **/api/Customer/{email} (GET)**:
  - **Descrição**: Retorna um cliente específico pelo email.
  - **Resposta**: Código 200 OK em caso de sucesso.
- **/api/Customer/{email} (PUT)**:
  - **Descrição**: Atualiza um cliente específico pelo email.
  - **Request Body**: JSON com `name`, `email`, `addresses`, e `companyFile`.
  - **Resposta**: Código 200 OK em caso de sucesso.
- **/api/Customer/{email} (DELETE)**:
  - **Descrição**: Exclui um cliente específico pelo email.

    - **Resposta**: Código 200 OK em caso de sucesso.

Componentes (Schemas)

- **CompanyFileRequest**: Representa um arquivo relacionado à empresa.
    - **Campos obrigatórios**: `fileName`, `contentType`, `data`.
- **CustomerRequest**: Representa as informações de um cliente.
    - **Campos obrigatórios**: `name`, `email`, `addresses`, `companyFile`.
- **LoginRequest**: Representa as credenciais para login.
    - **Campos obrigatórios**: `username`, `password`.

Detalhes Técnicos

- **Framework**: ASP.NET Razor Pages.
- **Scripts**: Use JavaScript para chamadas AJAX e manipulação do DOM.
- **Estilização**: Utilize bibliotecas CSS como Bootstrap para facilitar o design responsivo.

Parte 2: Arquitetura Back-End

Introdução

O back-end será responsável pelo processamento das requisições, lógica de negócios e interação com o banco de dados. A API RESTful será desenvolvida utilizando **C#** e **.NET Core 6.0** ou superior.

Requisitos Funcionais (Back-End)

1. **Gerenciar clientes e logradouros** (criar, atualizar, visualizar, remover).
2. **Armazenar logotipo do cliente** como `VARBINARY` no banco de dados.
3. **Garantir integridade referencial** entre clientes e logradouros.

Diagrama de Classes - CustomerRegistrationApi

1. **Classe Address**: Representa um endereço.
    - **Atributos**: `Street` (string), `Id` (int?).
    - **Relacionamento**: Um cliente pode ter múltiplos endereços (1:N).
2. **Classe CompanyFile**: Representa um arquivo relacionado à empresa.
    - **Atributos**: `FileName`, `ContentType`, `Data`, `Id`.
    - **Relacionamento**: Um cliente tem um arquivo associado (1:1).
3. **Classe Customer**: Representa um cliente.
    - **Atributos**: `Name`, `Email`, `Addresses`, `CompanyFile`, `Id`.
    - **Relacionamentos**:
        - **1:N** com a classe `Address`.
        - **1:1** com a classe `CompanyFile`.

Relacionamentos entre as Classes

- **Customer - Address**: Relacionamento de 1:N, um cliente pode ter vários endereços.
- **Customer - CompanyFile**: Relacionamento de 1:1, um cliente tem um arquivo associado.

Estruturas de Banco de Dados

- **Tabela CompanyFiles**:
  - **Colunas**: `Id`, `FileName`, `ContentType`, `Data` (VARBINARY).
- **Tabela Customers**:
  - **Colunas**: `Id`, `Name`, `Email`, `CompanyFileId`.
- **Tabela Addresses**:
  - **Colunas**: `Id`, `Street`, `CustomerId`.

Relacionamentos

- **Customers** têm 1:N com **Addresses** e 1:1 com **CompanyFiles**.
- **Ações de Migration**: Criação de tabelas, chaves estrangeiras, índices e ações de exclusão em cascata.

Configuração do Banco de Dados (Em Memória ou SQL Server)

Para configurar seu projeto para rodar com banco de dados em memória durante o desenvolvimento e permitir migração para SQL Server em produção, utilizamos o padrão **Adapter Pattern**.

Passos para Configuração

1. **Banco de Dados em Memória**:
   - Durante o desenvolvimento, configure para usar um banco de dados em memória.
2. **Banco de Dados SQL Server**:
   - Quando necessário, altere a string de conexão para usar o SQL Server.
3. **Exemplo de Injeção de Dependência**:

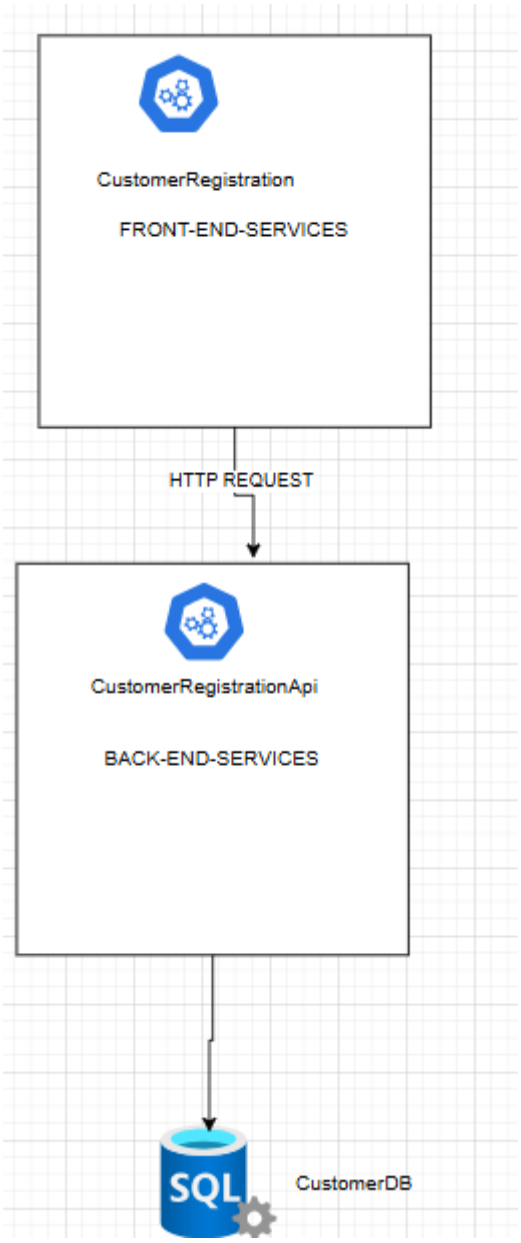
   // Registra o repositório com a injeção de dependência para memoria

   builder.Services.AddSingleton<ICustomerRepository, CustomerRepository>();

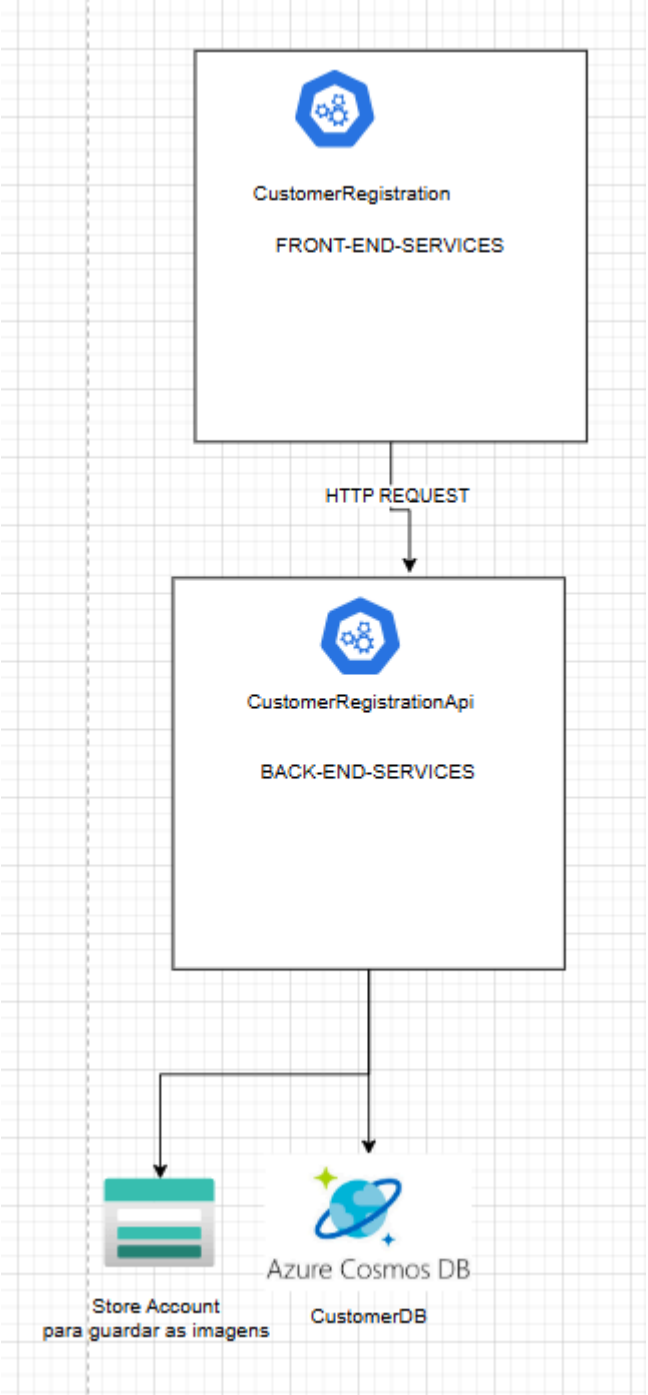
   // Registra o repositório com a injeção de dependência para sqlserver

   //builder.Services.AddScoped<ICustomerRepository, SqlServerCustomerRepository>();

Arquitetura de componente da solução



Sugestão de Melhoria para arquitetura para segunda versão



A migração do SQL Server para Cosmos DB e o uso do Azure Blob Storage para imagens trazem escalabilidade global, baixa latência e redução de custos. O Cosmos DB, com seu modelo NoSQL baseado em documentos JSON, proporciona flexibilidade para armazenar e consultar dados dinâmicos, enquanto elimina limitações de esquemas rígidos do SQL Server. Já o Azure Blob Storage separa o armazenamento de imagens dos dados, otimizando performance e reduzindo custos de infraestrutura.

Padrões de arquiteturas do sistema

1. Padrões Arquiteturais:

MVC (Model-View-Controller): Muito usado em aplicações web para separar a lógica da interface do usuário e a manipulação dos dados.

Clean Architecture: Separação de responsabilidades em camadas, como:

Core: Entidades e Regras de Negócio.

Application: Casos de Uso.

Infrastructure: Implementações de persistência, APIs externas.

Presentation: Controladores e endpoints.

2. Padrões de Design:

Repository Pattern: Uma camada de abstração entre a lógica de negócio e o banco de dados.

Singleton: Para gerenciar instâncias únicas, geralmente usado com o IoC Container para classes como serviços de configuração.

Domain-Driven Design (DDD):é uma abordagem de design de software focada no domínio do negócio. Ele organiza o sistema em torno de conceitos do domínio, utilizando uma **linguagem ubíqua** compartilhada entre desenvolvedores e especialistas. DDD divide o sistema em **contextos delimitados (Bounded Contexts)**

Error Handling Without Exceptions: é um padrão de projeto para lidar com erros sem o uso de mecanismos de exceção nativos da linguagem para melhorar performance;

3. Boas Práticas e Padrões do ASP.NET Core:

Dependency Injection (DI): Todas as classes e serviços configurados via IoC Container (geralmente em Program.cs ou Startup.cs).

Swagger/OpenAPI: Para documentação de APIs.

Configuration: Uso de arquivos de configuração como appsettings.json e injeção de IConfiguration para acessar informações.

4. Outros Aspectos Técnicos:

AutoMapper: Para mapear entre entidades e DTOs.

Testes Unitários e Mocking: Testes unitários com frameworks como **xUnit** com uso de Moq e AutoFIxture para criar mocks e fixtures.