# ChatBot.Solution - README

03/08/2025, 22:51:46

# **Sistema de Chat com Bot Inteligente**

## **Visão Geral**

O ChatBot.Solution é um sistema moderno de chat em tempo real que simula interações com um bot de atendimento. Construído com .NET 9 e arquitetura de ponta, demonstra práticas atuais de desenvolvimento de software, incluindo Vertical Slice Architecture, CQRS, e padrões de design robustos.

## **Backend - Arquitetura e Funcionamento**

### **Visão Geral do Backend**

O backend do ChatBot é a espinha dorsal da aplicação, responsável por gerenciar as sessões de chat, o processamento de mensagens, a lógica do bot e a persistência de dados. Projetado com princípios de arquitetura moderna, como Vertical Slice Architecture e CQRS (Command Query Responsibility Segregation), busca alta coesão, baixo acoplamento e excelente manutenibilidade.

### **Estrutura de Pastas**

A solução está organizada em projetos bem definidos, seguindo a abordagem de **monorepo modularizado** por camadas e funcionalidades:

ChatBot.Solution/
├── src/ # Código Fonte da Aplicação
│ ├── ChatBot.Api/ # Camada de Apresentação (Web API)
│ ├── ChatBot.Application/ # Camada de Aplicação (Lógica de Negócio, CQRS, Slices Verticais)
│ ├── ChatBot.Domain/ # Camada de Domínio (Regras de Negócio e Entidades)
│ ├── ChatBot.Infrastructure/ # Camada de Infraestrutura (Implementações de Persistência e Serviços Externos)
│ └── ChatBot.Shared/ # Contratos Compartilhados (DTOs, Enums, Constantes)
├── tests/ # Projetos de Teste (Unit, Integration, Architecture)
│ ├── ChatBot.ArchitectureTests/
│ ├── ChatBot.IntegrationTests/
│ └── ChatBot.UnitTests/
└── docs/ # Documentação adicional

### **Diagrama Arquitetural**

A arquitetura adota uma abordagem híbrida, combinando aspectos de Clean Architecture com Vertical Slice Architecture, gerenciada pelo padrão Mediator.

+----------------------------------------------------------------------------------+
| ChatBot.Solution |
| |
| +---------------------+ +------------------------+ +------------------+
| | ChatBot.Api | | ChatBot.Application | | ChatBot.Domain |
| | (Presentation Layer)|<-----|(Application Layer) |<-----|(Core Business) |
| | (HTTP Endpoints) | | (CQRS, MediatR, Slices)| | (Entities, Rules)|
| +----------+----------+ +----------+-------------+ +--------^---------+
| | ^ |
| | (Uses Interfaces) | |
| V | |
| +----------+----------+ +------------+-----------+ +--------+---------+
| | ChatBot.Shared | | ChatBot.Infrastructure|------>| External Systems|
| | (Common Contracts) |<---- | (Implementation Layer) | | (LLMs, Databases)|
| +---------------------+ | (Repositories, Services)| +------------------+
| +------------------------+
+----------------------------------------------------------------------------------+

**Principais Relações:**

* **ChatBot.Api** (Camada de Apresentação): Depende de ChatBot.Application e ChatBot.Shared
* **ChatBot.Application** (Camada de Aplicação): Depende de ChatBot.Domain e ChatBot.Shared
* **ChatBot.Infrastructure** (Camada de Infraestrutura): Depende de ChatBot.Domain e ChatBot.Application
* **ChatBot.Domain** (Camada de Domínio): É independente, apenas utiliza ChatBot.Shared
* **ChatBot.Shared** (Contratos Compartilhados): É a camada mais básica, utilizada por todas as outras

## **Explicação Detalhada das Camadas**

### **1. ChatBot.Api (Web API Layer - Camada de Apresentação)**

Esta camada é o ponto de entrada para a aplicação, expondo as funcionalidades via HTTP (RESTful) e gerenciando a comunicação em tempo real via SignalR.

**Responsabilidades:**

* Receber requisições HTTP e transformá-las em comandos ou queries
* Enviar comandos/queries para a camada de aplicação via MediatR
* Retornar respostas HTTP adequadas
* Gerenciar comunicação em tempo real via SignalR

**Componentes Principais:**

* **Controllers:** Endpoints da API RESTful (ChatController, BotController, UsersController)
* **Hubs:** Comunicação em tempo real via SignalR (ChatHub.cs)
* **Middleware:** Processamento transversal (ExceptionHandlingMiddleware)
* **Filters:** Validação adicional e autorização
* **Extensions:** Configuração do pipeline da aplicação

### **2. ChatBot.Application (Application Layer - Camada de Aplicação)**

Esta é a camada de orquestração e a casa dos “Vertical Slices”. Contém a lógica de aplicação e coordena as operações entre o domínio e a infraestrutura.

**Responsabilidades:**

* Implementar CQRS através do MediatR
* Organizar código por funcionalidade (Vertical Slices)
* Coordenar operações entre domínio e infraestrutura
* Aplicar comportamentos transversais (logging, validação, transações)

**Componentes Principais:**

* **Features (Vertical Slices):** Organização por funcionalidade (Chat, Bot, Users)
* **Commands:** Operações que modificam estado
* **Queries:** Operações que buscam dados
* **Handlers:** Processam comandos e queries
* **Validators:** Validações FluentValidation
* **Events:** Eventos de aplicação

**Common:** Componentes compartilhados

* **Behaviors:** Pipeline MediatR (LoggingBehavior, ValidationBehavior, TransactionBehavior, PerformanceBehavior)
* **Exceptions:** Exceções customizadas
* **Interfaces:** Contratos base (ICommand, IQuery, IUnitOfWork)
* **Models:** Modelos genéricos (Result<T>)

### **3. ChatBot.Domain (Domain Layer - Camada de Domínio)**

Esta é a camada mais interna e o coração do negócio. Contém as entidades, regras de negócio e eventos de domínio. É totalmente independente de outras camadas.

**Responsabilidades:**

* Definir entidades e regras de negócio
* Manter a pureza das regras de domínio
* Definir contratos para repositórios e serviços
* Gerenciar eventos de domínio

**Componentes Principais:**

* **Entities:** Conceitos do negócio com identidade (User, ChatSession, Message, BotResponse)
* **ValueObjects:** Conceitos sem identidade (Email, MessageContent)
* **Enums:** Enumerações do domínio (SessionStatus, MessageType, BotResponseType)
* **Interfaces:** Contratos transversais (IAuditable, ISoftDeletable, IDomainEvent)
* **Repositories:** Interfaces para persistência
* **Services:** Interfaces para serviços de domínio
* **Events:** Eventos de domínio (MessageSentDomainEvent, ChatSessionEndedDomainEvent)

### **4. ChatBot.Infrastructure (Infrastructure Layer - Camada de Infraestrutura)**

Esta camada contém as implementações concretas dos contratos definidos nas camadas Domain e Application. Lida com sistemas externos.

**Responsabilidades:**

* Implementar repositórios e serviços
* Gerenciar acesso a dados (Entity Framework Core)
* Comunicar com sistemas externos
* Implementar factories e adapters

**Componentes Principais:**

* **Data:** Acesso e persistência (ChatBotDbContext, Configurations, Interceptors, Migrations, UnitOfWork)
* **Repositories:** Implementações concretas dos repositórios
* **Services:** Implementações de serviços (SignalRChatService)
* **Factories:** Implementações de factories (BotResponseStrategyFactory)
* **External:** Adapters para serviços externos

### **5. ChatBot.Shared (Shared Contracts - Contratos Compartilhados)**

Esta camada contém definições de tipos e utilitários utilizados por múltiplas camadas.

**Responsabilidades:**

* Definir DTOs para comunicação entre camadas
* Fornecer constantes e enums compartilhados
* Oferecer métodos de extensão utilitários

**Componentes Principais:**

* **DTOs:** Objetos de transferência de dados organizados por domínio
* **Constants:** Constantes globais e mensagens
* **Enums:** Enums compartilhados
* **Extensions:** Métodos de extensão utilitários

## **Fluxo de Requisição: Do Endpoint ao Banco de Dados**

Vamos detalhar o caminho completo de uma mensagem, desde a requisição HTTP até a persistência e notificação em tempo real:

### **Exemplo: Envio de Mensagem**

[Cliente] → [API] → [Application] → [Domain] → [Infrastructure] → [Database]
↓ ↓ ↓ ↓ ↓ ↓
HTTP POST → Controller → MediatR → Entities → EF Core → PostgreSQL
↑ ↑ ↑ ↑ ↑ ↑
[Resposta] ← [JSON] ← [Result<T>] ← [Events] ← [SignalR] ← [Notifications]

#### **1. Requisição HTTP na API (`ChatBot.Api`)**

POST /api/chat/send-message
Content-Type: application/json
{
"chatSessionId": "guid",
"userId": "guid",
"content": "Olá, preciso de ajuda!"
}

* O ChatController.SendMessage recebe a requisição
* Cria um SendMessageCommand com os dados
* Delega ao IMediator (MediatR)

#### **2. Pipeline MediatR na Camada de Aplicação (`ChatBot.Application`)**

**Pipeline Behaviors (executados em ordem):**

1. LoggingBehavior: Registra início da requisição e parâmetros

2. ValidationBehavior: Executa SendMessageCommandValidator (FluentValidation)

3. PerformanceBehavior: Monitora tempo de execução

4. TransactionBehavior: Inicia transação de banco para comandos

#### **3. Execução do Handler (`SendMessageCommandHandler`)**

public async Task<Result<SendMessageResponse>> Handle(SendMessageCommand request, CancellationToken cancellationToken)
{
// 1. Validar se a sessão de chat existe
var chatSession = await \_chatSessionRepository.GetByIdAsync(request.ChatSessionId);
// 2. Validar se o usuário existe
var user = await \_userRepository.GetByIdAsync(request.UserId);
// 3. Criar nova mensagem
var message = new Message
{
ChatSessionId = request.ChatSessionId,
UserId = request.UserId,
Content = request.Content,
Type = MessageType.Text,
IsFromBot = false,
SentAt = DateTime.UtcNow
};
// 4. Adicionar evento de domínio
message.AddDomainEvent(new MessageSentDomainEvent(
message.Id,
message.ChatSessionId,
message.UserId,
message.Content,
message.SentAt,
message.IsFromBot
));
// 5. Persistir mensagem
await \_messageRepository.AddAsync(message);
await \_unitOfWork.SaveChangesAsync();
// 6. Retornar resultado
return Result<SendMessageResponse>.Success(new SendMessageResponse
{
MessageId = message.Id,
ChatSessionId = message.ChatSessionId,
Content = message.Content,
SentAt = message.SentAt
});
}

#### **4. Interação com Infraestrutura (`ChatBot.Infrastructure`)**

**Persistência no Banco:**

* UnitOfWork.SaveChangesAsync() coordena com ChatBotDbContext
* **Interceptors do EF Core:**
* AuditableEntityInterceptor: Preenche CreatedAt, CreatedBy, UpdatedAt, UpdatedBy
* SoftDeleteInterceptor: Garante soft delete ao invés de exclusão física

Transação é commitada pelo TransactionBehavior

#### **5. Publicação de Eventos de Domínio**

Após o SaveChangesAsync, o TransactionBehavior publica todos os Domain Events:

// MessageSentEventHandler é invocado
public async Task Handle(MessageSentDomainEvent notification, CancellationToken cancellationToken)
{
// Notifica clientes via SignalR em tempo real
await \_signalRChatService.SendMessageToChatSession(
notification.ChatSessionId,
notification.Content,
notification.IsFromBot,
notification.UserId,
notification.MessageId,
notification.SentAt
);
}

#### **6. Retorno da Resposta**

HTTP 200 OK
Content-Type: application/json
{
"messageId": "guid",
"chatSessionId": "guid",
"userId": "guid",
"content": "Olá, preciso de ajuda!",
"sentAt": "2024-01-30T10:30:00Z"
}

## **Tecnologias e Padrões de Projeto**

### **Tecnologias Escolhidas**

#### **.NET 9**

* **Performance**: Alto desempenho para aplicações em tempo real
* **Produtividade**: Ambiente robusto com vasta documentação
* **Modernidade**: Incorpora últimas tendências e otimizações

#### **Entity Framework Core (Code-First)**

* **ORM Moderno**: Simplifica interação com banco de dados
* **Code-First**: Esquema gerado a partir dos modelos de domínio
* **Migrations**: Evolução automatizada do banco de dados

#### **PostgreSQL**

* **Confiabilidade**: Banco de dados robusto e maduro
* **Performance**: Excelente para aplicações transacionais
* **Recursos Avançados**: Suporte a JSON, índices avançados, etc.

#### **SignalR**

* **Tempo Real**: Comunicação bidirecional instantânea
* **Escalabilidade**: Suporte a múltiplas conexões simultâneas
* **Flexibilidade**: Fallback automático para diferentes protocolos

### **Padrões de Projeto Implementados**

#### **1. Vertical Slice Architecture**

**Uso**: Organização principal da camada Application

**Benefícios**:

* **Organização por Funcionalidade**: Código agrupado por “slice” de negócio
* **Baixo Acoplamento**: Cada slice é autônomo
* **Alta Coesão**: Código relacionado vive junto
* **Manutenibilidade**: Mudanças localizadas a um slice específico

#### **2. Mediator Pattern (MediatR)**

**Uso**: Comunicação entre Controllers e Handlers

**Benefícios**:

* **Desacoplamento**: Remetentes não conhecem receptores
* **CQRS Simplificado**: Separação clara entre Commands e Queries
* **Pipeline Behaviors**: Comportamentos transversais configuráveis

#### **3. Factory Method**

**Uso**: Criação de estratégias de resposta do bot

public interface IBotResponseStrategyFactory
{
IBotResponseStrategy CreateStrategy(string messageContent);
}
public class BotResponseStrategyFactory : IBotResponseStrategyFactory
{
public IBotResponseStrategy CreateStrategy(string messageContent)
{
if (messageContent.ToLower() == "sair")
return new ExitCommandStrategy();
if (ContainsKeywords(messageContent))
return new KeywordBasedResponseStrategy();
return new RandomResponseStrategy();
}
}

**Benefícios**:

* **Flexibilidade**: Criação dinâmica de estratégias
* **Extensibilidade**: Novas estratégias sem modificar código existente
* **Princípio Open/Closed**: Aberto para extensão, fechado para modificação

#### **4. Strategy Pattern**

**Uso**: Diferentes lógicas de resposta do bot

public interface IBotResponseStrategy
{
Task<string> GenerateResponseAsync(string userMessage);
}
public class RandomResponseStrategy : IBotResponseStrategy
{
public async Task<string> GenerateResponseAsync(string userMessage)
{
var responses = new[] { "Interessante!", "Entendi.", "Continue..." };
return responses[Random.Next(responses.Length)];
}
}
public class ExitCommandStrategy : IBotResponseStrategy
{
public async Task<string> GenerateResponseAsync(string userMessage)
{
return "Obrigado por usar nosso chat! Até logo!";
}
}

**Benefícios**:

* **Algoritmos Intercambiáveis**: Diferentes comportamentos encapsulados
* **Seleção Dinâmica**: Estratégia escolhida em tempo de execução
* **Manutenibilidade**: Cada estratégia é independente

#### **5. Observer Pattern (Domain Events)**

**Uso**: Reação a eventos de domínio

// Evento de Domínio
public record MessageSentDomainEvent : IDomainEvent
{
public Guid MessageId { get; init; }
public Guid ChatSessionId { get; init; }
public string Content { get; init; }
public DateTime OccurredOn { get; } = DateTime.UtcNow;
}
// Observer (Event Handler)
public class MessageSentEventHandler : INotificationHandler<MessageSentDomainEvent>
{
public async Task Handle(MessageSentDomainEvent notification, CancellationToken cancellationToken)
{
// Reage ao evento enviando notificação via SignalR
await \_signalRChatService.SendMessageToChatSession(/\* ... \*/);
}
}

**Benefícios**:

* **Desacoplamento**: Produtores de eventos não conhecem consumidores
* **Extensibilidade**: Novos handlers podem ser adicionados facilmente
* **Responsabilidade Única**: Cada handler tem uma responsabilidade específica

#### **6. Unit of Work**

**Uso**: Gerenciamento de transações

public interface IUnitOfWork
{
Task<int> SaveChangesAsync(CancellationToken cancellationToken = default);
Task BeginTransactionAsync(CancellationToken cancellationToken = default);
Task CommitTransactionAsync(CancellationToken cancellationToken = default);
Task RollbackTransactionAsync(CancellationToken cancellationToken = default);
}

**Benefícios**:

* **Atomicidade**: Operações tratadas como uma única transação
* **Consistência**: Garante integridade dos dados
* **Controle**: Gestão explícita de transações

#### **7. Command Pattern (CQRS)**

**Uso**: Separação entre operações de comando e consulta

// Command (modifica estado)
public record SendMessageCommand : ICommand<Result<SendMessageResponse>>
{
public Guid ChatSessionId { get; init; }
public Guid UserId { get; init; }
public string Content { get; init; }
}
// Query (busca dados)
public record GetChatHistoryQuery : IQuery<Result<ChatHistoryResponse>>
{
public Guid ChatSessionId { get; init; }
public int PageSize { get; init; } = 50;
}

**Benefícios**:

* **Separação de Responsabilidades**: Commands vs Queries
* **Otimização**: Diferentes estratégias para leitura e escrita
* **Clareza**: Intenções explícitas nas operações

### **Princípios SOLID Aplicados**

#### **Single Responsibility Principle (SRP)**

* Cada classe tem uma única responsabilidade
* Handlers focados em uma única operação
* Separação clara entre camadas

#### **Open/Closed Principle (OCP)**

* Sistema aberto para extensão via Strategy Pattern
* Novos handlers adicionados sem modificar código existente
* Pipeline behaviors configuráveis

#### **Liskov Substitution Principle (LSP)**

* Implementações de interfaces são substituíveis
* Strategies podem ser trocadas transparentemente

#### **Interface Segregation Principle (ISP)**

* Interfaces específicas e focadas
* Clientes dependem apenas do que usam

#### **Dependency Inversion Principle (DIP)**

* Dependências de abstrações, não de implementações
* Injeção de dependência em todas as camadas

## **Endpoints da API**

### **Chat Endpoints**

#### **Iniciar Sessão de Chat**

POST /api/chat/start-session
Content-Type: application/json
{
"userId": "guid-opcional",
"userName": "Nome do Usuário",
"initialMessageContent": "Olá!"
}

#### **Enviar Mensagem**

POST /api/chat/send-message
Content-Type: application/json
{
"chatSessionId": "guid",
"userId": "guid",
"content": "Mensagem do usuário"
}

### **SignalR Hub**

#### **Conexão**

const connection = new signalR.HubConnectionBuilder()
.withUrl("/chathub")
.build();
// Receber mensagens
connection.on("ReceiveMessage", (chatSessionId, content, isFromBot, userId, messageId, sentAt) => {
// Atualizar interface do chat
});

## **Configuração e Execução**

### **Pré-requisitos**

* .NET 9 SDK
* PostgreSQL 12+
* Visual Studio 2022 ou VS Code

### **Configuração do Banco de Dados**

1. **Configurar Connection String** em appsettings.json:

{
"ConnectionStrings": {
"DefaultConnection": "Host=localhost;Database=ChatBotDb;Username=postgres;Password=sua\_senha"
}
}

1. **Executar Migrations**:

dotnet ef database update --project src/ChatBot.Infrastructure --startup-project src/ChatBot.Api

### **Executar a Aplicação**

cd src/ChatBot.Api
dotnet run

A API estará disponível em:

* HTTP: http://localhost:5000
* HTTPS: https://localhost:5001
* Swagger: https://localhost:5001/swagger

*Esta é a primeira parte da documentação, focada no Backend. As próximas seções abordarão Frontend, Testes e Deploy.*