**UNIVERSIDADE POSITIVO**

# DOCUMENTAÇÃO TÉCNICA – API GESTÃO DE EVENTOS

Gabriel Henrique Vaz dos Santos | Vinicius Mulling | Joabe Ramos Leal | Eduardo José dos Santos S.

Repositório Github | Postman Workspace

**Curitiba**

**2025**

# Índice

1. **Objetivo do Projeto**

   O objetivo desse projeto foi desenvolver uma API de Gestão de Eventos, voltada tanto para Pessoas Jurídicas quanto Pessoas Físicas. A API contém recursos de autenticação para realizar a limitação entre requisições de acordo com o nível hierárquico (Role) do usuário (Admin > organizador > cliente).

   A aplicação para clientes possibilita operações do tipo: compra de ingressos, visualização dos eventos, atrações de cada evento, lista de organizadores e outras operações.

   Os organizadores podem: Cadastrar um novo evento, cadastrar uma atração no evento de própria autoria (organizador x não pode cadastrar atração no evento do organizador y), podem atualizar informações de clientes etc.

   Já o admin tem o acesso total na aplicação: Pode excluir organizadores/clientes, criar organizadores/clientes e entre muito mais.

## 2. Estrutura da Aplicação

A organização da aplicação foi baseada no modelo de “Arquitetura em Camadas”, onde é realizado um projeto específico para cada pasta e dentro do projeto da pasta “WebApi” é realizada a composição e relacionamentos entre as outras pastas como: “Application”, “Domain”, “Infrastructure” etc.
Para o projeto não fugir do escopo acadêmico, optamos por fazer algo parecido, com a seguinte estrutura:

**Application –** Localização dos DTOs (Transferidores de dados entre back-end e front-end) e arquivos de funcionalidades do sistema, como: Gerador de senha HASH (criptografia para banco de dados) e JWT (autenticação via token).

**Controllers –** Localização dos controladores do projeto, aqui fica o coração da API, contendo todos os ENDPOINTS e métodos de requisição.

**Data –** Localização do DbContext e das Migrations, basicamente, a pasta onde se localiza as configurações de Banco de Dados.

**Domain –** Localização das Entities (modelos) e Enums, toda alteração realizada em qualquer uma das entidades é necessária a realização de uma nova migration para atualizar o banco de dados.

### 2.1 Application

#### 2.1.1 DTOs (Data Transfer Object):

Padrão usado para transportar dados entre camadas da aplicação sem expor diretamente as entidades do banco. Eles ajudam a garantir segurança, performance e desacoplamento entre sistemas.

#### 2.1.2 HashGenerator:

Classe utilitária usada para gerar códigos criptográficos, geralmente aplicados em senhas, garantindo que elas sejam armazenadas de forma segura no banco de dados.

#### 2.1.3 JwtSettings:

Classe de configuração que armazena parâmetros importantes para a geração e validação de tokens JWT, como chave secreta, tempo de expiração e emissor do token.

### 2.2 Controllers

#### 2.2.1 AtracaoController:

Gerencia as operações relacionadas às atrações de um evento, como cadastro, listagem, atualização e exclusão.

#### 2.2.2 AuthController:

Responsável pelas funcionalidades de autenticação, como login, geração de tokens JWT e registro de usuários.

#### 2.2.3 ClienteController

Lida com as operações específicas dos clientes, permitindo visualização e gerenciamento dos seus dados (Endereço, CPF, Cidade, Estado etc.).

#### 2.2.4 EventoController:

Controla o cadastro, edição, listagem e exclusão de eventos organizados na plataforma.

#### 2.2.5 IngressoController:

Gerencia os ingressos vinculados a eventos, como criação, consulta por tipo e vinculação com clientes.

#### 2.2.6 OrganizadorController:

Cuida das ações relacionadas aos organizadores de eventos, como gerenciamento de perfil e eventos criados.

#### 2.2.7 UsuarioController

Trata as operações gerais dos usuários da plataforma, como listagem e controle de permissões.

### 2.3 Data

#### 2.3.1 Migrations:

Contém os arquivos de migração do Entity Framework, que definem as alterações na estrutura do banco de dados ao longo do tempo.

#### 2.3.2 AppDbContext:

Classe que representa o contexto do banco de dados, mapeando entidades para tabelas e configurando o acesso aos dados.

### 2.4 Domain

#### 2.4.1 Entities:

Modelos de dados que representam os objetos principais do sistema, como Evento, Cliente, Ingresso etc.

#### 2.4.2 Enums:

Enumerações utilizadas para categorizar dados fixos, como tipos de ingresso ou papéis de usuário (Roles).

### 2.5 Program:

Arquivo principal de entrada da aplicação, onde é feita a configuração e inicialização do servidor e dos serviços da API.

### 2.6 Relacionamento de entidades:

- Organizador se relaciona com Evento (1 organizador pode ter muitos eventos)
- Cliente se relaciona com Ingresso (1 cliente pode ter muitos ingressos)
- Evento se relaciona com Ingresso (1 evento pode ter muitos ingressos)
- Organizador e Cliente se relacionam com Usuário (todo Org e Cliente são usuários)

## 3. Endpoints

Abaixo segue todas as requisições em cada endpoint dos controllers.
Link do workspace no Postman com exemplos de requisições: Clique aqui

### 3.1 AuthController

**Método POST http://localhost:5031/api/auth/login:**
Método utilizado para logar no sistema. O retorno dessa requisição é um token de autenticação, qual será necessário para a maioria dos outros métodos, dependendo do seu nível de Role (admin, organizador ou cliente).

**Método POST http://localhost:5031/api/auth/register:**
Método utilizado para registrar um usuário no sistema. O retorno dessa requisição é a criação de um cliente na table "Cliente" no banco de dados.

**Método GET http://localhost:5031/api/auth/usuarios:**
Método utilizado para listar todos os usuários cadastrados no banco de dados.

### 3.2 AtracaoController

**Método POST http://localhost:5031/api/atracao:**
Método utilizado para criar uma atração dentro de um Evento já existente. Somente organizadores podem criar atrações para os próprios eventos. (Não pode associar uma atração para um evento que não seja de sua autoria)

**Método GET http://localhost:5031/api/atracao ou /api/atracao/{id}:**
Métodos utilizados para listar todas as atrações, ou, se especificar um id de uma atração na requisição ele retorna somente a atração em específico.

**Método PUT http://localhost:5031/api/atracao/{id}:**
Método utilizado para atualizar as informações de uma atração existente, somente o organizador dono do evento e atração pode modificar ela. (Ou o admin do sistema)

**Método DELETE http://localhost:5031/api/atracao/{id}:**
Método utilizado para excluir uma atração em específico.

### 3.3 ClienteController

**Método GET http://localhost:5031/api/cliente:**
Método utilizado para listar todos os clientes.

**Método PUT http://localhost:5031/api/cliente/{id}:**
Método utilizado para o usuário atualizar os próprios dados que não foram informados na hora do registro, como: Endereço, CPF etc.

### 3.4 EventoController

**Método POST http://localhost:5031/api/evento:**
Método utilizado para ORGANIZADORES criarem eventos. É necessário estar autenticado como Organizador para conseguir criar um evento, pois o sistema puxa o Id do usuário logado para associar o evento ao usuário.

**Método GET http://localhost:5031/api/evento ou /evento/{id}:**
Método utilizado para listar todos os eventos existentes ou um evento específico.

**Método PUT http://localhost:5031/api/evento/{id}:**
Método utilizado para um Organizador atualizar o próprio evento.

**Método DELETE http://localhost:5031/api/evento/{id}:**
Método utilizado para um Organizador apagar o próprio evento. (ou admin apagar evento de qualquer organizador)

### 3.5 IngressoController

**Método POST http://localhost:5031/api/ingresso:**
Método utilizado para criar um ingresso, necessário estar autenticado com qualquer tipo de conta para isso.

**Método GET http://localhost:5031/api/ingresso ou /ingresso/{id}:**
Método utilizado para listar todos os ingressos ou um ingresso específico.

**Método PUT http://localhost:5031/api/ingresso/{id}:**
Método utilizado para atualizar um ingresso específico.

**Método DELETE http://localhost:5031/api/ingresso/{id}:**
Método utilizado para deletar um ingresso específico.

### 3.6 OrganizadorController

**Método POST http://localhost:5031/api/Organizador:**
Método para criar um Organizador.

**Método GET http://localhost:5031/api/Organizador ou /Organizador/{id}:**
Método para listar todos os organizadores ou um organizador em específico.

**Método PUT http://localhost:5031/api/Organizador/{id}:**
Método para atualizar os dados de um organizador em específico.

**Método DELETE http://localhost:5031/api/Organizador/{id}:**
Método para deletar um organizador em específico.

### 3.7 UsuarioController

**Método PUT http://localhost:5031/api/Usuario:**
Método utilizado para criar um usuário. (Somente em casos específicos onde é necessário criar um usuário de forma manual, porque ele já é criado automaticamente no AuthController)

**Método GET http://localhost:5031/api/Usuario ou /Usuario/{id}:**
Método utilizado para listar todos os usuários ou um usuário específico.

**Método PUT http://localhost:5031/api/Usuario/{id}:**
Método utilizado para atualizar os dados de um usuário em específico.

**Método DELETE http://localhost:5031/api/Usuario/{id}:**
Método utilizado para excluir um usuário em específico.

## 4. Modelagem de Dados – Entities e Enums

O Projeto foi dividido em diversos componentes para a modularização do código, promovendo melhor manutenção e compreensão do conteúdo. Nós poderíamos também ter aplicado o uso de uma EntityBase, uma classe padrão para evitar a repetição de atributos, mas não levamos em consideração pois achávamos que seria um projeto de pequeno escopo.

A maioria conta com atributos padrões, como: **Id, nome, descrição, categoria etc.** (com pequenas diferenças entre entidades)

### 4.1 Atracao – Entity:

Entidade com atributos padrão, com vínculo de relacionamento com a entidade Evento. Todo evento precisa de uma atração, um evento pode ter várias atrações e uma atração pertence à um evento.

### 4.2 Cliente – Entity:

Entidade com atributos padrão, informações adicionais de pessoas físicas e com vínculo de relacionamento com a entidade Ingresso. Todo cliente pode ter vários ingressos para vários eventos diferentes.

### 4.3 Evento – Entity:

Entidade com atributos padrão e com vínculo de relacionamento com organizador, todo evento PRECISA de um Organizador para existir. Também possuí vínculo com a entidade Ingresso, onde um evento pode ter vários ingressos e um ingresso pode estar vinculado a um evento.

### 4.4 Ingresso – Entity:

Entidade com atributos padrão e com vínculo de relacionamento com Cliente, Evento e TipoIngresso. Todo ingresso PRECISA de um Cliente e Evento para existir. A relação com TipoIngresso é o tipo de ingresso vendido, como: Adulto, estudante etc.

### 4.5 Organizador – Entity:

Entidade com atributos padrão, informações adicionais de pessoas jurídicas e com vínculo de relacionamento com a entidade Usuario. Todo Organizador é um usuário, mas nem todo Usuário é um Organizador.

### 4.6 Usuario – Entity:

Entidade com atributos padrão, utilizada para construir o usuário que irá se autenticar no sistema. Possui vínculo com o enum Roles, onde é definido se será Cliente, Organizador ou Admin. O construtor padrão o define como “Cliente”, então todo usuário criado a princípio, será um cliente.

### 4.7 Roles – Enum:

Enum utilizado para atribuir diferentes valores para o mesmo contexto, onde pode ser definido Admin, Organizador ou Cliente.

### 4.8 TipoIngresso – Enum:

Enum utilizado para atribuir diferentes tipos de ingresso em sua criação, podendo ser 9 tipos diferentes.

5. **Justificativa Técnica**

   A modelagem das entidades principais — como **Evento, Cliente, Organizador e Ingresso** — foi elaborada com base em requisitos reais de sistemas de gestão de eventos. Cada entidade foi pensada para refletir operações comuns nesse tipo de aplicação, como organização de eventos, emissão e compra de ingressos, e controle de usuários com diferentes permissões de acesso.

   Essa modelagem orientou a construção de toda a estrutura da aplicação, desde a criação das tabelas no banco de dados até a definição das regras de negócio e rotas de cada controller. O uso de enums como Roles e TipoIngresso garantiu consistência e padronização nas funcionalidades que exigem validação de papéis e tipos categorizados.

   A separação em camadas (**Application, Domain, Data, Controllers**) foi adotada seguindo os princípios da *arquitetura em camadas*, promovendo uma organização clara e escalável do código. Essa abordagem facilita a manutenção da aplicação e favorece futuras expansões, como a implementação de novas funcionalidades (ex: pagamento de ingressos, sistema de avaliação de eventos, notificações, entre outras).

   A escolha por utilizar DTOs no transporte de dados entre o front-end e o back-end foi motivada pela necessidade de garantir segurança (não exposição direta das entidades do domínio), melhor desempenho (apenas dados necessários sendo trafegados) e flexibilidade (adaptação dos formatos para diferentes contextos).

   Adicionalmente, foram implementadas práticas essenciais de segurança, como autenticação com JWT e criptografia de senhas com hash, visando aproximar o sistema de um ambiente real de produção.

## 6. Conclusão

O desenvolvimento da API de Gestão de Eventos proporcionou uma aplicação robusta e escalável, com funcionalidades bem definidas para diferentes tipos de usuários: administradores, organizadores e clientes. A aplicação foi construída com base em boas práticas de desenvolvimento, como a separação em camadas, uso de DTOs, autenticação via JWT e integração com banco de dados utilizando o Entity Framework.

Além de atender aos requisitos do projeto acadêmico, a solução oferece uma base sólida para expansão futura, permitindo a inclusão de novas funcionalidades, como pagamento online, integração com sistemas externos e interface front-end. O projeto também contribuiu para o aprimoramento técnico do grupo em relação ao uso de APIs RESTful, arquitetura de software e segurança da informação.

Por fim, a API demonstra a importância de uma estrutura bem planejada e modularizada para garantir manutenibilidade, clareza e eficiência no desenvolvimento de aplicações reais.