

Search or jump to... Pull requests Issues Marketplace Explore

Watch 1 Star 1 Fork 4

Code Issues 0 Pull requests 0 Projects 0 Wiki Security Insights

BackEnd do projeto GUFOS

32 commits 1 branch 0 releases 1 contributor

Branch: master New pull request Create new file Upload files Find file Clone or download

paulobrandaooofficial Update README.md Latest commit 46d4cb0 10 days ago

Controllers	JWT Ok	10 days ago
Models	Entity Framework Ok	11 days ago
Properties	Projeto Gufos Iniciado	11 days ago
bin/Debug/netcoreapp3.0	JWT Ok	10 days ago
obj	JWT Ok	10 days ago
GUFOS_BackEnd.csproj	JWT Ok	10 days ago
Program.cs	Projeto Gufos Iniciado	11 days ago
README.md	Update README.md	10 days ago
Startup.cs	JWT Ok	10 days ago
appsettings.Development.json	Projeto Gufos Iniciado	11 days ago
appsettings.json	JWT Ok	10 days ago

README.md

## GUFOS - Agenda de Eventos - BackEnd C# - .Net Core 3.0

### Requisitos

- Visual Studio Code
- Banco de Dados funcionando - DDLs, DMLs e DQLs
- .NET Core SDK 3.0

### Criação do Projeto

Criamos nosso projeto de API com:

```
dotnet new webapi
```

### Entity Framework - Database First

Instalar o gerenciador de pacotes EF na máquina:

```
dotnet tool install --global dotnet-ef
```

Baixar Pacote SQL Server:

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

Baixar pacote de escrita de códigos do EF:

```
dotnet add package Microsoft.EntityFrameworkCore.Design
```

Dar um restore na aplicação para ler e aplicar os pacotes instalados:

```
dotnet restore
```

Testar se o EF está ok

```
dotnet ef
```

Criar os Models à partir da sua base de Dados -o = criar o diretório caso não exista -d = Incluir as DataNotations do banco

```
dotnet ef dbcontext scaffold "Server=DESKTOP-XVGT587\SQLEXPRESS;Database=Gufos;Trusted_Connection=True;" Microsoft.EF
```

## Controllers

Apagamos o controller que já vem com a base...

### CategoriaController

Criamos nosso primeiro Controller: CategoriaController

Herdamos nosso novo controller de ControllerBase

Definimos a "rota" da API logo em cima do nome da classe, utilizando:

```
[Route("api/[controller]")]
```

Logo abaixo dizemos que é um controller de API, utilizando:

```
[ApiController]
```

Damos CTRL + . para incluir:

```
using Microsoft.AspNetCore.Mvc;
```

Instanciamos nosso contexto da nossa Base de Dados:

```
GufosContext _contexto = new GufosContext();
```

Damos CTRL + . para incluir nossos models:

```
using GUFOS_BackEnd.Models;
```

Criamos nosso método GET:

```
// GET: api/Categoria/
[HttpGet]
public async Task<ActionResult<List<Categoria>>> Get()
{
    var categorias = await _context.Categoria.ToListAsync();

    if (categorias == null)
    {
        return NotFound();
    }

    return categorias;
}
```

Importamos com CTRL + . as dependências:

```
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;
using System.Collections.Generic;
```

Testamos o método GET de nosso controller no Postman:

```
dotnet run
https://localhost:5001/api/categoria
```

Deve ser retornado:

```
[
  {
    "categoriaId": 1,
    "titulo": "Desenvolvimento",
    "evento": []
  },
  {
    "categoriaId": 2,
    "titulo": "HTML + CSS",
    "evento": []
  },
  {
    "categoriaId": 3,
    "titulo": "Marketing",
    "evento": []
  }
]
```

Criamos nossa sobrecarga de método GET, desta vez passando como parâmetro o ID:

```

// GET: api/Categoria/5
[HttpGet("{id}")]
public async Task<ActionResult<Categoria>> Get(int id)
{
    var categoria = await _context.Categoria.FindAsync(id);

    if (categoria == null)
    {
        return NotFound();
    }

    return categoria;
}

```

Testamos no Postman: <https://localhost:5001/api/categoria/1>

Criamos nosso método POST para inserir uma nova categoria:

```

// POST: api/Categoria/
[HttpPost]
public async Task<ActionResult<Categoria>> Post(Categoria categoria)
{
    try
    {
        await _context.AddAsync(categoria);
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        throw;
    }

    return categoria;
}

```

Testamos no Postman, passando em RAW , do tipo JSON:

```
{
    "titulo": "Teste"
}
```

Criamos nosso método PUT para atualizar os dados:

```

// PUT: api/Categoria/5
[HttpPut("{id}")]
public async Task<IActionResult> Put(long id, Categoria categoria)
{
    if (id != categoria.CategoriaId)
    {
        return BadRequest();
    }

    _context.Entry(categoria).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        var categoria_valido = await _context.Categoria.FindAsync(id);

        if (categoria_valido == null)
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return NoContent();
}

```

Testamos no Postman, no método PUT, pela URL <https://localhost:5001/api/categoria/4> passando:

```
{
    "categoriaId": 4,
    "titulo": "Design Gráfico"
}
```

Por último, incluímos nosso método DELETE , para excluir uma determinada Categoria:

```

// DELETE: api/Categoria/5
[HttpDelete("{id}")]
public async Task<ActionResult<Categoria>> Delete(int id)
{
    var categoria = await _context.Categoria.FindAsync(id);
    if (categoria == null)
    {
        return NotFound();
    }

    _context.Categoria.Remove(categoria);
    await _context.SaveChangesAsync();

    return Ok();
}

```

```

        }

        _context.Categoria.Remove(categoria);
        await _context.SaveChangesAsync();

        return categoria;
    }
}

```

Testamos pelo Postman, pelo método DELETE, e com a URL: <https://localhost:5001/api/categoria/4>  
Deve-se retornar o objeto excluído:

```
{
    "categoriaId": 4,
    "titulo": "Design Gráfico",
    "evento": []
}
```

## LocalizacaoController

Copiar ControllerCategoria e alterar com CTRL + F  
Testar os métodos REST

## EventoController

Copiar ControllerCategoria e alterar com CTRL + F  
Testar os métodos REST  
Notamos que no método GET não retorna a árvore de objetos *Categoria* e *Localizacao*  
Para incluirmos é necessário adicionar em nosso projeto o seguinte pacote:

```
dotnet add package Microsoft.AspNetCore.Mvc.NewtonsoftJson
```

Depois em nossa Startup.cs, dentro de ConfigureServices, no lugar de services.AddControllers():

```
services.AddControllersWithViews().AddNewtonsoftJson(opt => opt.SerializerSettings.ReferenceLoopHandling = Reference
```

Damos CTRL + . para incluir a dependência:

```
using Newtonsoft.Json;
```

Após isso precisamos mudar nosso controller para receber os atributos, no método GET ficará assim:

```
var eventos = await _context.Evento.Include(c => c.Categoria).Include(l => l.Localizacao).ToListAsync();
```

No método GET com parâmetro ficará assim:

```
var evento = await _context.Evento.Include(c => c.Categoria).Include(l => l.Localizacao).FirstOrDefaultAsync(e => e.
```

Adicionar os Controllers restantes

## SWAGGER - Documentação da API

Instalar o Swagger:

```
dotnet add Gufos_BackEnd.csproj package Swashbuckle.AspNetCore -v 5.0.0-rc4
```

Registramos o gerador do Swagger dentro de ConfigureServices, definindo 1 ou mais documentos do Swagger:

```
services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo { Title = "API", Version = "v1" });
    // Mostrar o caminho dos comentários dos métodos Swagger JSON and UI.
    var xmlFile = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
    var xmlPath = Path.Combine(AppContext.BaseDirectory, xmlFile);
    c.IncludeXmlComments(xmlPath);
});
```

Colocar na Startup com CTRL + .

```
using Microsoft.OpenApi.Models;
using System.Reflection;
using System.IO;
```

Colocar dentro do csproj para gerar a documentação com base nos comentários dos métodos:

```
<PropertyGroup>
  <GenerateDocumentationFile>true</GenerateDocumentationFile>
  <NoWarn>$(NoWarn);1591</NoWarn>
</PropertyGroup>
```

Em Startup.cs , no método Configure , habilita o middleware para atender ao documento JSON gerado e à interface do usuário do Swagger:

```
// Habilitamos efetivamente o Swagger em nossa aplicação.
app.UseSwagger();
// Especificamos o endpoint da documentação
app.UseSwaggerUI(c =>
{
    c.SwaggerEndpoint("/swagger/v1/swagger.json", "API V1");
});
```

Rodar a aplicação e testar em: <https://localhost:5001/swagger/>

## JWT - Autenticação com Json Web Token

Instalar pacote JWT

```
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer --version 3.0.0
```

Adicionar a configuração do nosso Serviço de autenticação:

```
// JWT
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
{
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,
        ValidIssuer = Configuration["Jwt:Issuer"],
        ValidAudience = Configuration["Jwt:Issuer"],
        IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(Configuration["Jwt:Key"]))
    };
});
```

Importar com CTRL + . as dependências:

```
using Microsoft.IdentityModel.Tokens;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using System.Text;
```

Adicionamos em nosso appsettings.json :

```
{
  "Jwt": {
    "Key": "GufosSecretKey",
    "Issuer": "gufos.com"
  }
}
```

Em Startup.cs , no método Configure , usamos efetivamente a autenticação:

```
app.UseAuthentication();
```

Criamos o Controller *LoginController* e herdamos da  *ControllerBase*  
Colocamos a rota da API e dizemos que é um controller de API :

```
[Route("api/[controller]")]
[ApiController]
public class LoginController : ControllerBase
{}
```

Criamos nossos métodos:

```
// Chamamos nosso contexto do banco
```

```

GufosContext _context = new GufosContext();

// Definimos uma variável para percorrer nossos métodos com as configurações obtidas no appsettings.json
private IConfiguration _config;

// Definimos um método construtor para poder passar essas configs
public LoginController(IConfiguration config)
{
    _config = config;
}

// Chamamos nosso método para validar nosso usuário da aplicação
private Usuario AuthenticateUser(Usuario login)
{
    var usuario = _context.Usuario.FirstOrDefault(u => u.Email == login.Email && u.Senha == login.Senha);

    if (usuario != null)
    {
        usuario = login;
    }

    return usuario;
}

// Criamos nosso método que vai gerar nosso Token
private string GenerateJSONWebToken(Usuario userInfo)
{
    var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_config["Jwt:Key"]));
    var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256);

    // Definimos nossas Claims (dados da sessão) para poderem ser capturadas
    // a qualquer momento enquanto o Token for ativo
    var claims = new[] {
        new Claim(JwtRegisteredClaimNames.NameId, userInfo.Nome),
        new Claim(JwtRegisteredClaimNames.Email, userInfo.Email),
        new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString())
    };

    // Configuramos nosso Token e seu tempo de vida
    var token = new JwtSecurityToken(_config["Jwt:Issuer"],
        _config["Jwt:Issuer"],
        claims,
        expires: DateTime.Now.AddMinutes(120),
        signingCredentials: credentials);

    return new JwtSecurityTokenHandler().WriteToken(token);
}

```

// Usamos essa anotação para ignorar a autenticação neste método, já que é ele quem fará isso

```

[AllowAnonymous]
[HttpPost]
public IActionResult Login([FromBody]Usuario login)
{
    IActionResult response = Unauthorized();
    var user = AuthenticateUser(login);

    if (user != null)
    {
        var tokenString = GenerateJSONWebToken(user);
        response = Ok(new { token = tokenString });
    }

    return response;
}

```

Importamos as dependências:

```

using System;
using System.IdentityModel.Tokens.Jwt;
using System.Linq;
using System.Text;
using GUFOS_BackEnd.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Configuration;
using Microsoft.IdentityModel.Tokens;

```

Testamos se está sendo gerado nosso Token pelo Postman, no método POST

Pela URL: <https://localhost:5001/api/login>  
E com os seguintes parâmetros pela RAW :

```
{
    "nome": "Administrador",
    "email": "adm@adm.com",
    "senha": "123",
}
```

O retorno deve ser algo do tipo:

```
{
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJJuYW1laWQiOiJQYXVsbyIsImVtYWlsIjoiYWRtQGFkbS5jb20iLCJqdGkiOiIwY
```

Após confirmar, vamos até <https://jwt.io/>

Colamos nosso Token lá e em Payload devemos ter os seguintes dados:

```
{  
  "nameid": "Administrador",  
  "email": "adm@adm.com",  
  "jti": "d1e13b73-5f8f-423c-97e2-835f55bbfb0e",  
  "exp": 1571157573,  
  "iss": "gufos.com",  
  "aud": "gufos.com"  
}
```

Pronto! Agora é só utilizar a anotação `[Authorize]` em baixo da anotação REST de cada método que desejar colocar autenticação!

No Postman devemos gerar um token pela rota de login e nos demais endpoints devemos adicionar o token gerado na aba *Authorization* escolhendo a opção *Bearer Token*

