(CP5) sobre **Autenticação e Autorização com JWT em ASP.NET Core Web API**

A startup **SafeScribe** está desenvolvendo uma plataforma inovadora para gestão de notas e documentos sensíveis voltada para equipes corporativas. A segurança e o controle de acesso são os pilares do produto. Eles precisam de um backend robusto que garanta que apenas usuários autenticados tenham acesso ao sistema e que suas permissões sejam aplicadas de forma rigorosa.

A sua missão é construir o núcleo da API RESTful da SafeScribe, implementando um sistema de autenticação e autorização seguro utilizando JSON Web Tokens (JWT).

Requisitos Técnicos:

- **Framework:** .NET 8 (ou superior)
- **Tipo de Projeto:** ASP.NET Core Web API
- **Autenticação:** JWT (JSON Web Tokens) com o pacote `Microsoft.AspNetCore.Authentication.JwtBearer`.

Entrega e outras regras:

- **20/10/2025**
- **Grupo até 3 pessoas**
- **Link do Projeto no GITHUB**
- **Nome e RM dos membros no README**

Tarefas e Requisitos do Projeto:

Tarefa 1: Estrutura, Modelagem e Configuração da Segurança

O primeiro passo é definir os modelos de dados e configurar o serviço de autenticação JWT na aplicação.

1. **Modelagem das Entidades:**
    a. Crie uma classe `User` com propriedades essenciais: `Id`, `Username`, `PasswordHash` (nunca guarde a senha em texto plano!) e `Role` (função/perfil).
    b. Crie uma classe `Note` com propriedades como `Id`, `Title`, `Content`, `CreatedAt` e `UserId` (para identificar o dono da nota).
2. **Definição das Funções (Roles):**
    a. Crie um `enum` ou uma classe estática para representar as funções dos utilizadores no sistema. Pelo menos três funções devem existir:
        i. `Leitor`: Pode apenas visualizar as suas próprias notas.
        ii. `Editor`: Pode criar e editar as suas próprias notas.
        iii. `Admin`: Possui controlo total, podendo visualizar, editar e apagar as notas de qualquer utilizador.
3. **Configuração do JWT:**
    a. Adicione as configurações do JWT (Chave Secreta, Emissor, Audiência) ao ficheiro `appsettings.json`.
    b. Em `Program.cs`, configure os serviços de autenticação e autorização. Use `AddAuthentication()` e `AddJwtBearer()` para definir como o token JWT deve ser validado.

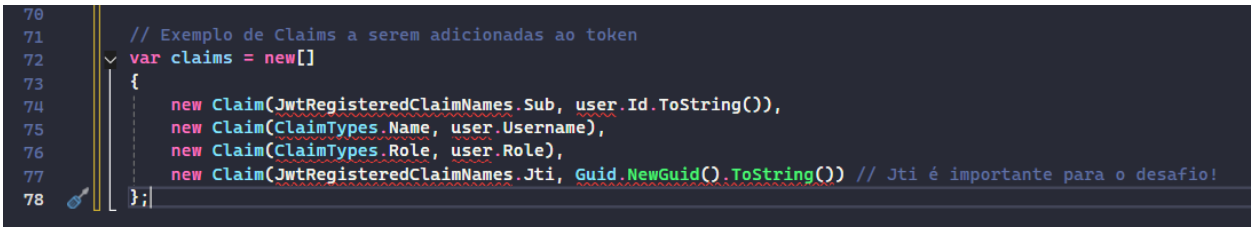
```
// Exemplo da configuração em Program.cs
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = builder.Configuration["Jwt:Issuer"],
            ValidAudience = builder.Configuration["Jwt:Audience"],
            IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(builder.Configuration["Jwt:Key"]))
        };
    });
```

c. **Documentação:** Explique no código (usando comentários) o que cada opção do `TokenValidationParameters` significa.

Tarefa 2: Serviço de Autenticação e Geração de Token

Crie um serviço dedicado para gerir a lógica de registo, login e criação de tokens.

1. **Crie a Interface e a Implementação do Serviço (DIP):**
    a. Defina uma interface `ITokenService` com métodos para as operações de autenticação.
    b. Implemente a classe `TokenService` que dependerá das configurações (`IConfiguration`) para ler os dados do `appsettings.json`.
2. **Implemente os Métodos do Serviço:**
    a. **Registo de Utilizador:** Crie um método que receba os dados de um novo utilizador, gere um hash da senha (recomende o uso de uma biblioteca como `BCrypt.Net`) e o salve (pode ser numa base de dados em memória para simplificar, como o Entity Framework In-Memory).
    b. **Login e Geração de Token:** Crie um método que valide as credenciais do utilizador. Se forem válidas, gere um token JWT contendo "claims" (reivindicações) essenciais, como o `UserId` e a `Role` do utilizador.



```
// Exemplo de Claims a serem adicionadas ao token
var claims = new[]
{
    new Claim(JwtRegisteredClaimNames.Sub, user.Id.ToString()),
    new Claim(ClaimTypes.Name, user.Username),
    new Claim(ClaimTypes.Role, user.Role),
    new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()) // Jti é importante para o desafio!
};
```

Tarefa 3: Construção dos Endpoints da API RESTful com Autorização

Exponha a lógica através de controllers, aplicando as regras de segurança.

1. **Crie o `AuthController`:**
    a. Este controller terá endpoints públicos para os utilizadores se registarem e fazerem login.
    b. `POST /api/v1/auth/registrar`: Endpoint para criar um novo utilizador.
    c. `POST /api/v1/auth/login`: Endpoint que recebe credenciais e, se válidas, retorna o token JWT.
2. **Crie o `NotasController`:**
    a. Este controller deve ser protegido por defeito. Adicione o atributo `[Authorize]` no topo da classe.
    b. `POST /api/v1/notas` - Criar Nota:
        i. Requer autorização de `Editor` ou `Admin`. Use `[Authorize(Roles = "Editor,Admin")]`.
        ii. **Lógica de Segurança:** O `UserId` da nova nota deve ser obtido a partir da claim do token do utilizador autenticado, e não de um dado enviado no corpo da requisição. Isso previne que um utilizador crie uma nota em nome de outro.
    c. `GET /api/v1/notas/{id}` - Obter Nota:
        i. Requer autorização.
        ii. **Lógica de Segurança:** Um utilizador com a função `Leitor` ou `Editor` só pode aceder às notas que ele mesmo criou. Um `Admin` pode aceder

a qualquer nota. A lógica de verificação deve ser implementada dentro do método.

d. **`PUT /api/v1/notas/{id}` - Atualizar Nota:**
    i. Requer autorização.
    ii. **Lógica de Segurança:** Similar à obtenção, um `Editor` só pode atualizar as suas próprias notas. Um `Admin` pode atualizar qualquer uma.
e. **`DELETE /api/v1/notas/{id}` - Apagar Nota:**
    i. Requer autorização estrita. Apenas um `Admin` pode apagar notas. Use `[Authorize(Roles = "Admin")]`.

3. **Use DTOs (Data Transfer Objects):**
    a. Crie DTOs como `UserRegisterDto`, `LoginRequestDto`, `NoteCreateDto` para separar os modelos da API dos modelos de domínio, melhorando a segurança e a organização.

Desafio Final (Lógica de Segurança Avançada):

Para testar a fundo a compreensão do ciclo de vida dos tokens e do pipeline do ASP.NET Core, implemente um mecanismo de **logout com blacklist de tokens**.

- **Problema:** Por natureza, um JWT é "stateless". Uma vez emitido, ele é válido até a sua expiração, mesmo que o utilizador queira fazer "logout". O desafio é invalidar um token ativo antes do seu tempo de expiração.
- **Endpoint:** `POST /api/v1/auth/logout`
    - Este endpoint deve ser protegido (`[Authorize]`).
    - **Funcionalidade:** Ao ser chamado, ele deve pegar o token JWT que foi usado na requisição e adicioná-lo a uma "lista negra" (blacklist), garantindo que ele não possa ser reutilizado.
- **Requisitos de Implementação:**

- **Crie um Serviço de Blacklist:**
  - Crie uma interface `ITokenBlacklistService` e uma implementação `InMemoryTokenBlacklistService`.
  - Este serviço deve ser registado como **Singleton** no container de injeção de dependência.
  - Ele deve ter métodos como `AddToBlacklistAsync(string jti)` e `IsBlacklistedAsync(string jti)`. O "jti" (JWT ID) é uma claim única em cada token, perfeita para este propósito.
- **Crie um Middleware Personalizado:**
  - Crie uma classe de middleware (`JwtBlacklistMiddleware`).
  - Este middleware deve ser adicionado ao pipeline de requisições em `Program.cs`, **depois** do middleware de autenticação (`app.UseAuthentication()`).
  - **Lógica do Middleware:** Para cada requisição autenticada, o middleware deve extrair a claim `jti` do token, injetar o `ITokenBlacklistService` e verificar se o token está na blacklist. Se estiver, o middleware deve interromper a requisição e retornar um status `401 Unauthorized`.

Este desafio força o aluno a ir além da configuração padrão, exigindo a criação de serviços com gestão de estado (mesmo que em memória) e a manipulação do pipeline de requisições, tornando a solução muito mais robusta e difícil de ser gerada por uma IA genérica.

Critérios de Avaliação:

- **Configuração de Autenticação e JWT (30%):**
  - O JWT está corretamente configurado em `Program.cs` e `appsettings.json`?
  - A geração de tokens inclui as claims necessárias (UserId, Role, Jti)?
  - As senhas dos utilizadores são armazenadas de forma segura usando hash?
- **Implementação de Endpoints e Lógica de Autorização (40%):**
  - Os atributos `[Authorize]` e `[Authorize(Roles = "...")]` são aplicados corretamente nos controllers e endpoints?
  - A lógica de negócio dentro dos métodos respeita as regras de permissão (ex: um editor só pode alterar as suas próprias notas)?
  - A identidade do utilizador é obtida corretamente a partir das claims do token para operações críticas?
- **Qualidade do Código e Arquitetura (20%):**
  - Uso claro do Princípio de Inversão de Dependência (DIP) com serviços e interfaces?
  - Código limpo, bem organizado e com nomes significativos?
  - Utilização de DTOs para separar as camadas da aplicação?
  - Tratamento de erros adequado (retorno de códigos de status HTTP como 401, 403, 404)?
- **Implementação do Desafio Final (Logout) (10%):**
  - O serviço de blacklist foi criado e registado como singleton?
  - O middleware personalizado foi implementado e interceta corretamente as requisições?
  - A funcionalidade de logout efetivamente impede que um token seja reutilizado?