



Segurança em APIs ASP.NET com JWT e Bearer Authentication

Índice

[Índice](#)

[Introdução](#)

[Antes de começar](#)

[O que é autenticação?](#)

[Como garantir que eu sou eu?](#)

[Autenticação Externa](#)

[O que é autorização](#)

[Por onde começar?](#)

[Autenticação e autorização em APIs](#)

[Onde armazenar os Tokens?](#)

[Session Storage](#)

[Local Storage](#)

[Cookies](#)

[Domínios e Sub Domínios](#)

[Banco de Dados](#)

[O que é JWT?](#)

[Entendendo os Payloads e assinatura](#)

[Não armazene valores sensíveis no Payload](#)

[Tempo de vida do JWT](#)

[Refresh Token](#)

[Gerando Tokens com ASP.NET](#)

[Criando a API](#)

[Chave Privada](#)
[Criando um usuário](#)
[Como gerar um Token?](#)
[Assinando o Token](#)
[Token Descriptor](#)
[Expirando o Token](#)
[Gerando o Payload](#)
 [Entendendo os Claims](#)
 [Gerando um ClaimsIdentity](#)
[Token Service](#)
[Registrando o Token Service](#)
[Testando a API](#)
 [Testando via Postman](#)
 [Inspecionando o Token](#)
[Implementando autenticação na API](#)
 [Adicionando autenticação](#)
 [Adicionando autorização](#)
 [Adicionando suporte ao JWT](#)
 [Bearer Authentication](#)
 [Definindo o esquema de autenticação](#)
 [Challenge](#)
 [Lendo o Token](#)
 [Restringindo rotas](#)
 [Obtendo o usuário logado](#)
 [Acessando o HttpContext](#)
[Autorizando as rotas da API](#)
 [Entendendo Roles, Claims e Policies](#)
 [Criando Políticas](#)
 [Registrando as políticas](#)
 [Utilizando as políticas](#)
[Testando a API](#)
 [Importando a coleção](#)
 [Testando o login](#)
 [401 - Unauthorized](#)
 [403 - Forbidden](#)
[Código Fonte](#)

Introdução

Segurança sempre é um tema complexo e delicado, e quando se trata de APIs, estes pontos nos trazem ainda mais dúvidas e com certeza precisam de muito mais cuidado.

Neste eBook vamos aprender o que é autenticação, autorização, JWT, Bearer oAuth e implementar tudo isto em ASP.NET utilizando Minimal APIs.

Antes de começar

Antes de seguir neste eBook, é importante salientar que vamos partir do pre-suposto que você já conhece ASP.NET.

Caso necessário, você pode seguir nossos cursos nesta ordem abaixo, para aprender mais sobre Razor, ASP.NET e Minimal APIs.

Fundamentos do ASP.NET 6

Aprenda a construir APIs, implementar autenticação e autorização e desenvolvimento de serviços com o Framework mais amado pela comunidade.

 <https://balta.io/cursos/fundamentos-aspnet>



Uma visão geral sobre o ASP.NET Razor Pages

Descubra o que é como funciona um dos Frameworks Web mais maduros e utilizados do mercado.

 <https://balta.io/cursos/uma-visao-geral-do-aspnet-razor-pages>



O que é autenticação?

Autenticação é o processo que diz quem você é. Por exemplo, em um processo de autenticação interno ou externo, eu estou garantindo que sou o André Baltieri, através do E-mail xyz@balta.io.

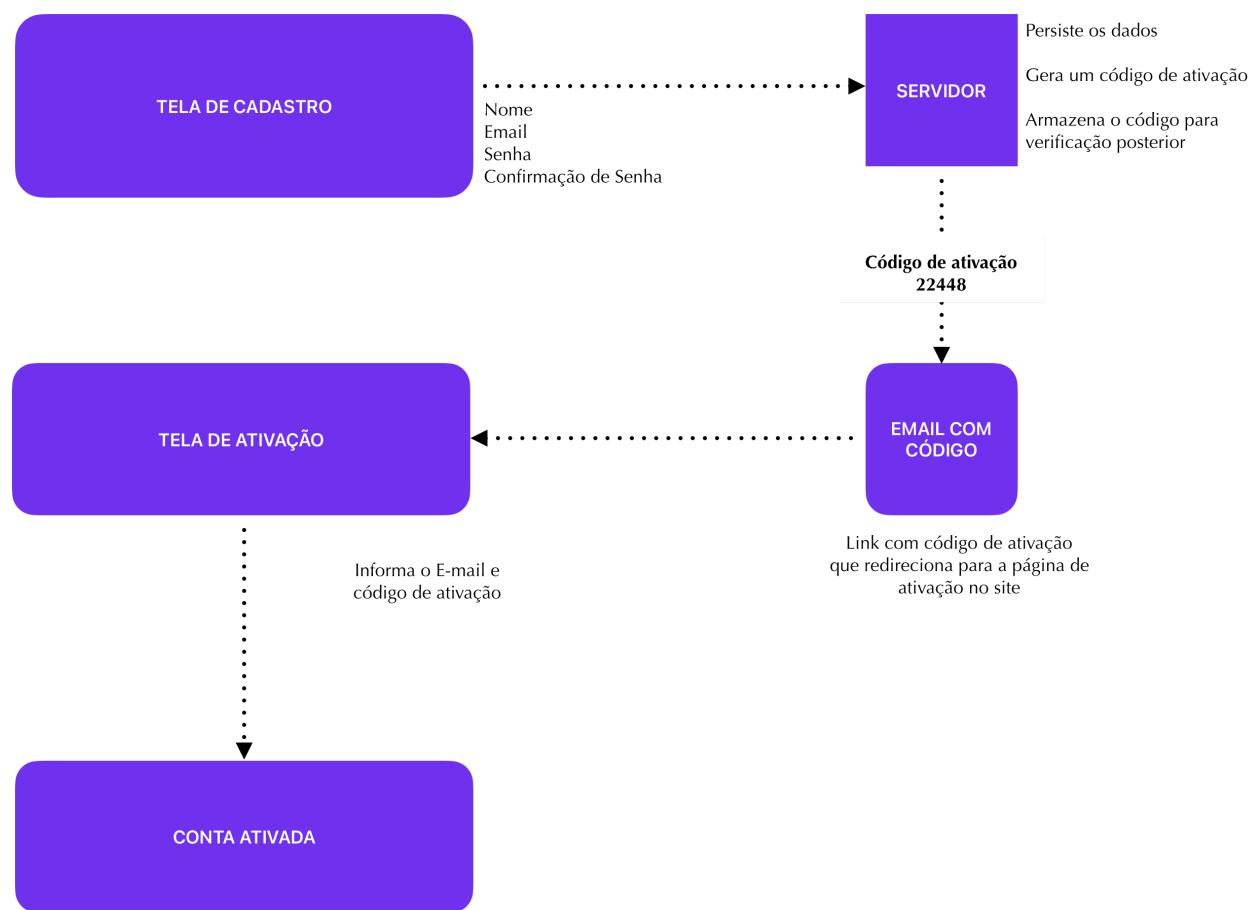
Este processo pode ser feito de diferentes maneiras, via usuário e senha, via E-mail e senha ou mesmo via redes sociais, os famosos “Login com Google”, “Login com Facebook”.

De qualquer forma, não importa como, o processo é sempre o mesmo, garantir que quem está dizendo que é “xyz@balta.io” é realmente o “André Baltieri”.

Como garantir que eu sou eu?

O primeiro passo que precisamos é garantir que uma pessoa está ligada a um E-mail ou um nome de usuário, e este processo é relativamente simples.

No caso da garantia de um E-mail ser de quem ele realmente disse que é, basta no processo de registro do usuário, enviar um E-mail com um código a ele.



Autenticação Externa

Outro tipo de autenticação muito comum é utilizando alguma rede social ou serviço externo. Neste modelo, você delega a responsabilidade para outro servidor, o que pode ser uma boa, já que o processo de verificação de contas do Google ou Microsoft por exemplo são bem mais complexos e completos do que possivelmente o seu será.

Neste formato, o que fazemos é no login, gerar um token de ativação e redirecionar o usuário para uma plataforma externa.

Após autenticado, o usuário retorna para nossa plataforma com um token e assim damos andamento na requisição.

Em suma, qualquer pessoa pode fornecer este serviço, basta realizar a implementação do OIDC (Open Id Connect), um protocolo aberto de autenticação.

Existem servidores OIDCs prontos como o Identity Server ou o Keycloak, ambos fornecem uma ótima implementação e são completos em recursos.

Resumindo, se o [balta.io](#) tivesse uma implementação do OIDC, você poderia adicionar um botão “Login com balta” em seu site.

Como o custo e risco de manter um OIDC próprio são altos, a recomendação é sempre começar do mais básico, implementando autenticação OAuth com JWT.

O que é autorização

Se autenticação diz quem você é, autorização diz o que você pode fazer, são os famosos Roles ou Perfis, e que no [ASP.NET](#) se estendem para políticas (Policies) e afirmações (Claims).

Enquanto a autenticação segue em diversas vezes uma padronização, a autorização não tem necessariamente uma regra. Eu mesmo já fiz sistemas onde ao invés de Roles utilizávamos Tags.

De qualquer forma, a ideia aqui é, sabendo que o “xyz@balta.io” é o “André Baltieri”, o que ele pode fazer dentro deste sistema?

Note que estamos falando DESTE sistema, pois a autorização varia muito, inclusive entre módulos, páginas e até mesmo botões.

Podem haver páginas no sistema que eu posso ver, mas não posso editar e a autorização precisa tratar tudo isto.

Na maioria dos casos também, a autorização é CUMULATIVA ou seja, eu posso ter vários perfis como “admin”, “employee”, “sales” e cada um deles ter funções distintas que são acumuladas.

Por onde começar?

Existem modelos prontos como o Keycloak e Identity Server que são implementações do OIDC, mas também existem Frameworks como o [ASP.NET](#) Identity que nos permite criar um sistema de usuários em poucos minutos.

Porém, minha recomendação é pelo menos para fins de estudo, que você implemente manualmente um sistema de autenticação, para conhecer as entranhas e saber tudo o que está acontecendo.

Desta forma, te convida a implementar um modelo de autenticação híbrido, contendo autenticação interna e externa, implementando OAuth com [ASP.NET](#) e Blazor.

Autenticação e autorização em APIs

Como você já deve imaginar, tudo começa na API, visto que a segurança no lado do cliente é sempre fraca, todo processo deve rodar no servidor.

Armazenar um usuário e seus perfis é uma tarefa relativamente simples, incluindo ler este dados e enviar para tela, o problema está no armazenamento destes dados do outro lado.

Deixa eu te explicar melhor, em APIs nós nunca ficamos autenticados ou autorizados, a cada requisição este processo é feito. Isto se repete para toda requisição.

Existe um motivo para isto, e até um tempo atrás, utilizávamos sessão para manter estes dados em memória e o usuário permanecer conectado.

Com a distribuição das aplicações em diferentes servidores, manter o usuário conectado não é algo viável, pois os servidores não compartilham memória.

Então imagina que você acessou o site do [balta.io](#) agora e fez o login, o servidor armazenou seus dados de login em memória e você está agora visualizando uma aula com 10 minutos de duração.

Após terminar de ver a aula, você clica no botão concluir, porém, o servidor que você se autenticou previamente está ocupado, com muita carga. Neste momento entra em ação o Load Balancer ou balanceador de carga.

Ele rapidamente identifica que existe outro servidor do [balta.io](#) e que está livre, desocupado, então manda sua requisição para lá.

Como os servidores não compartilham memória, logo, você não está autenticado neste servidor e sua requisição falha com o erro 401.

Mudando o cenário para autenticação que temos hoje, onde a cada requisição você precisa se autenticar, este erro deixa de acontecer.

Neste modelo, geramos um Token de acesso, baseado em uma chave privada que só o servidor tem (Ela tem que ser comum entre os servidores) e então a cada requisição, o Frontend envia este token.

Com o Token em mãos, como temos a chave privada, conseguimos descriptar ele e obter os valores do usuário (E quaisquer outros valores que ele tenha).

Você pode também armazenar os Tokens para uma maior validação, mas isto implica em pelo menos uma requisição no banco de dados a cada requisição autenticada que sua API recebe.

Onde armazenar os Tokens?

Não faz sentido!!!! Foi a primeira coisa que pensei quando vi que os tokens devem ser armazenados pelo Frontend e enviados a cada requisição, mas deixa eu te explicar melhor este processo.

Se precisamos enviar o token a cada requisição, já que não ficamos autenticados nas APIs, precisamos armazená-los em algum lugar.

Porém, no Frontend só existem quatro possíveis lugares para que ele seja armazenado:

Session Storage

Session Storage é um local que os browsers disponibilizam para armazenarmos chave/valor como string. Ou seja, podemos armazenar algo como:

Chave	Valor
TOKEN	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZ
NOME	ANDRE BALTIERI

Tudo o que armazenamos no Session Storage vive apenas enquanto a aba estiver aberta, uma vez fechada, tudo se perde.

Este modelo é amplamente utilizado quando temos cenários mais críticos em segurança, como os casos dos bancos. Imagina esquecer de fazer logout e o app do seu banco permanecer logado mesmo quando você reinicia seu computador?

Em adicional, os Browsers disponibilizam uma API muito simples para trabalhar com Session Storage.

```
sessionStorage.setItem('chave', 'valor'); // Salva um valor  
sessionStorage.getItem('chave'); // Lê um valor salvo  
sessionStorage.removeItem('chave'); // Remove um item  
sessionStorage.clear(); // Limpa todos os dados
```

Local Storage

O Local Storage age exatamente igual ao Session Storage, exceto pelo fato do seu tempo de vida que persiste infinitamente.

Isto significa que uma vez que você salvou um valor no Local Storage, ele permanece lá até ser removido ou você limpar os dados do Browser.

```
localStorage.setItem('chave', 'valor'); // Salva um valor  
localStorage.getItem('chave'); // Lê um valor salvo  
localStorage.removeItem('chave'); // Remove um item  
localStorage.clear(); // Limpa todos os dados
```

O Local Storage é o item mais indicado para salvar nossas informações. Previamente eram os Cookies, mas com alguns problemas de vazamento e compartilhamento de Cookies, esta recomendação mudou.

Cookies

Os Cookies são automaticamente anexados nos cabeçalhos das requisições e alguns modelos de autenticação como as do ASP.NET MVC e ASP.NET Razor Pages trabalham com Cookies.

É importante lembrar que ambos modelos citados acima são diferentes do que estamos implementando aqui, por isto Cookies fazem sentido para o cenário.

De qualquer forma, tratar quando e como queremos compartilhar o Token de autenticação pelo Local Storage é a melhor opção para nosso cenário.

Domínios e Sub Domínios

É importante lembrar também que o armazenamento local (Session e Local) são baseados nos domínios e/ou sub domínios, o que significa que informações persistidas nas sessões do site balta.io por exemplo, não serão visíveis nas sessões dentro do site microsoft.com.

Desta forma, não temos como compartilhar informações entre storages de diferentes domínios ou sub domínios.

No caso dos Cookies, existem políticas que permitem estas trocas de informações, recursos como Same Site e troca de origem, desde que atribuídos de forma correta e consciente, tornam o Cookie uma ótima opção para Single Sign On por exemplo.

Banco de Dados

Sim, existe um banco de dados que roda dentro do Browser, chamado IndexDb e usando o Blazor WASM conseguimos até rodar o SQLite no Browser.

De qualquer forma, as restrições são as mesmas do Local Storage, eles duram até serem removidos mas com a vantagem de armazenar mais informações (Tamanho em disco).

Como no nosso caso, precisamos apenas de uma chave e valor e o Local Storage nos oferece até 200MB (Isto varia de acordo com o Browser), podemos novamente ficar com o Local Storage que é mais fácil, leve e simples.

O que é JWT?

Então você está me dizendo que eu vou armazenar um Token em um local onde o usuário (Ou outra pessoa) pode ir lá e visualizar?

Isto mesmo, os Tokens são como chaves de acesso, com informações e uma duração, ou seja, se alguém obter seu Token, ele pode “impersonar” ou fingir que é você.

Por isto a segurança física é a primeira e mais importante etapa que temos. Se alguém tem acesso ao seu Browser, fisicamente (ou remotamente) ele pode ver seu Token.

Na verdade, este seria o menor dos seus problemas, já que os dados de navegação são armazenados localmente, ou seja, todas as suas sessões estão em um arquivo.

Basta copiar este arquivo da sua máquina para a minha e pronto, estarei logado com todas as suas sessões.

Por este motivo frequentemente somos recomendados a não clicar em links suspeitos, visto que uma simples cópia expõe todas as suas informações.

Mas voltando aos Tokens, embora você possa armazenar uma chave/valor no Local Storage, é legal encriptar estas informações, correto?

Desta forma, se alguém roubar seu Token, não verá nada além de uma Hash como esta abaixo:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF
```

Como este processo é algo comum entre as aplicações e APIs, criou-se um padrão chamado **JWT** (Pronuncia-se JÓT), que é a sigla para **Json Web Token**.

Ao desencriptar este Token, temos como resultado os seguintes JSON:

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}  
  
{  
  "sub": "1234567890",  
  "name": "John Doe",  
  "iat": 1516239022  
}
```

Se você notar, o Token contém “.” para segmentar suas regiões e o mesmo é dividido em três partes principais.

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9 . ey
JzdWIi0iIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG91IiwiaWF0IjoxNTE2MjM5MDIyfQ . SfIKx
wRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5c

A primeira parte, em vermelho, é chamada de Header ou cabeçalho, que define o algoritmo utilizado na encriptação e o tipo do Token, no nosso caso, JWT.

Quando desencriptamos ela, temos o seguinte JSON como resultado:

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

Podemos mudar estes valores, incluindo o algoritmo e assim teremos valores diferentes. No caso, mudando de HS256 para HS384, temos a seguinte Hash e Header.

```
// eyJhbGciOiJIUzIwMjNCIsInR5cCI6IkpXVCJ9
{
    "alg": "HS384",
    "typ": "JWT"
}
```

O mais comum e recomendado até o momento da escrita deste artigo é o HS256, ele balanceia performance e segurança. Quanto mais alta a encriptação, mais processamento ela requer.

O segundo item, em rosa é o Payload ou carga, que são informações que podemos incluir no Token com algumas ressalvas.

```
//eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpvag4gRG9lIiwiYWRtaW4iOnRydWUsImhlhdCI6MTUxNjIzOTAyM
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true,
  "iat": 1516239022
}
```

Assim como nos Headers o Payload também varia de acordo com a quantidade de informações que colocamos nele. Ao incluirmos a informação “premium” temos um outro valor sendo gerado.

```
//eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpvag4gRG9lIiwiYWRtaW4iOnRydWUsInByZW1pdW0iOnRydWUsI
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true,
  "premium": true,
  "iat": 1516239022
}
```

Por fim temos a parte em azul, que representa a assinatura do Token, que só existe no lado do servidor.

Em resumo o que temos é uma encriptação SHA256 de três itens convertidos para Base64, um código simples assim:

```
HMACSHA256(  
    base64UrlEncode(header) + "." +
```

```
base64UrlEncode(payload),  
    'MINHA CHAVE SECRETA'  
)
```

Então quer dizer que, além de salvar o Token eu ainda posso visualizar ele? Sim, tem sites como o jwt.io que te permite visualizar tudo que um Token contém.

Indo além disso, o JWT.io (Você pode fazer isto manualmente também) te permite alterar o código de um Token, adicionando informações extras.

Isto significa que se eu pegar o Token abaixo, gerado para mim que diz que eu só tenho o perfil “student”:

**eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9 . ey
JuYW1lIjoiQW5kcs0pIEJhbHRpZXJpIiwicm9sZ
XMiOlsic3R1ZGVudCJdLCJpYXQiOjE1MTYyMzkw
MjJ9 . YAIevyGRu0yjei0gxVr6K9IZZ-
HeRBxRe4scPqQa-EY**

```
{  
    "name": "André Baltieri",  
    "roles": ["student"],  
    "iat": 1516239022  
}
```

E adicionar o perfil “admin”:

```
{  
    "name": "André Baltieri",  
    "roles": ["student", "admin"],  
    "iat": 1516239022  
}
```

Agora eu tenho um novo Token, com acesso de administrador:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9 . ey
JuYW1lIjoiQW5kcsOpIEJhbHRpZXJpIiwicm9sZ
XMiOlsic3R1ZGVudCI_sImFkbWluIl0sImhlhdCI6
MTUxNjIzOTAyMn0 . KeJnStjrmRGc9ZKlaWiI8TY
1nhciPD6eyFTvPT0QKng

Entendendo os Payloads e assinatura

Muita calma nessa hora! É possível sim mudar um Token, mas para isto é necessário uma chave privada, que somente o servidor deve conter.

Se o Token for gerado com qualquer outra chave, diferente da qual foi gerada, o mesmo é invalidado. Isto é o que torna os Tokens seguros em relações as mudanças.

Então não importa se o Token foi alterado no cliente ou mesmo no meio do caminho, se ele não for re-gerado com a chave privada (Que só o servidor deve conhecer) ele será inválido.

MAN IN THE MIDDLE - Existe um ataque comum chamado “Homem no meio” que basicamente intercepta a comunicação entre o cliente e o servidor e rouba informações ou modifica elas.

Não armazene valores sensíveis no Payload

Outro ponto importantíssimo é sobre o uso do Payload. Embora você possa adicionar qualquer informação que desejar nele, não é recomendado trafegar informações sensíveis ali.

Cartão de crédito, telefone, endereço ou qualquer informação que comprometa os dados do seu usuário, devem ser mantidos apenas no servidor.

Em suma, tendo o E-mail ou ID do usuário no Payload já basta. Com estas informações você pode consultar o que quiser sobre ele.

Tempo de vida do JWT

Uma recomendação e padrão dos JWTs é conter no Payload a informação “iat” que significa “Issued At” ou “Gerado em”, que nada mais é do que um timestamp da data/hora que o Token foi gerado.

Desta forma, podemos criar uma validação para o Token, dizendo que o mesmo só pode existir por X tempo. Assim, se um Token for roubado, ele só vai ser útil durante X dias ou horas, passado isto ele se torna inválido.

```
{  
  "sub": "1234567890",  
  "name": "John Doe",  
  "iat": 1516239022  
}
```

Dependendo do sistema este tempo de expiração pode variar, mas em geral, ele não deve ser tão curto a ponto de incomodar o usuário, que precisará se autenticar todo, nem tão longo que alguém possa roubar e usar por meses.

Refresh Token

Caso opte pelo uso de um tempo reduzido no tempo de vida dos Tokens, uma ótima alternativa é o uso dos Refresh Tokens.

Sempre que gerar um Token para o seu usuário, gere uma chave aleatória junto, encriptada e dado o Token anterior e mais esta nova chave, um novo Token pode ser gerado.

No caso, é interessante gerar este novo Token em um intervalo menor que a expiração do Token principal, desta forma, o usuário sempre mantém a sessão ativa.

Em adicional, você pode optar por não gerar um novo Token caso o anterior já tenha expirado, isto varia bastante incluindo a base em relação tempo de vida do Token.

Em resumo, supondo que meu Token expira em 2 horas, como eu sei o quanto ele dura e quando foi emitido, posso me prontificar e já gerar um novo Token (Refresh) desde que a chave do Refresh Token senha sido enviada junto.

Caso meu Token já tenha expirado, mas há menos de 2 horas, podemos manter o processo acima e também gerar o Token (Refresh).

Caso meu Token tenha expirado há mais de 2 horas, aí não tem jeito, o usuário precisa se autenticar novamente.

Gerando Tokens com ASP.NET

Mas chega de filosofia! Vamos colocar a mão na massa e gerar Tokens JWT com ASP.NET, o que é uma tarefa bem simples.

IMPORTANTE: Não vamos cobrir a parte de Entity Framework e acesso a dados aqui, isto está bem coberto nos cursos, vamos focar em gerar Tokens e autenticação/autorização no ASP.NET.

O ASP.NET tem um Middleware que cuida da parte de autenticação e que já tem suporte a JWT, então podemos usufruir deste item “in-the-box” que vai facilitar nossa vida.

Criando a API

Nosso primeiro passo é bem simples e pode ser feito com qualquer template do .NET que suporte ASP.NET, isto é, ASP.NET MVC, ASP.NET Razor Pages, ASP.NET Minimal APIs.

Para este exemplo em específico, escolhi as Minimal APIs por serem mais simples e assim podermos focar no que realmente interessa.

É importante notar que não cobriremos a parte de acesso a dados aqui, as informações serão mockadas a fim de simplificar o exemplo.

Acesso a dados é um dos pontos mais importantes de uma aplicação e devemos tratá-los com responsabilidade. Caso queira se aprofundar no assunto, recomendo seguir a trilha abaixo:

Fundamentos do SQL Server

Aprenda ou consolide todos os conhecimentos necessários para trabalhar com bancos de dados relacionais tendo o SQL Server como referência.

 <https://balta.io/cursos/fundamentos-sql-server>



balta.io

**Fundamentos
SQL Server**

Acesso à dados com .NET, C#, Dapper e SQL Server

Aprenda a se conectar no SQL Server via .NET utilizando C# e Dapper, realizar operações CRUD e consolidar seus conhecimentos sobre os fundamentos.

 <https://balta.io/cursos/acesso-dados-csharp-net-dapper-sql-server>



balta.io

**Acesso à Dados
C#, .NET, Dapper e
SQL Server**

Fundamentos do Entity Framework

Neste curso vamos aprender os fundamentos do Entity Framework Core colocando a mão na massa e passando por diversos cenários com um dos mais populares ORMs do mercado

🔗 <https://balta.io/cursos/fundamentos-entity-framework>



Fundamentos do Entity Framework

Aplicações Mult-Tenant com Entity Framework Core

Precisa suportar múltiplos clientes de forma saudável e escalável? Então confira alguns prós e contras de diferentes modelos e abordagens usando Mult Tenant com Entity Framework Core

🔗 <https://balta.io/cursos/aplicacoes-mult-tenant-entity-framework-core>



.NET

Mult Tenant com Entity Framework Core

Voltando a questão da criação da API, vamos utilizar o comando abaixo para criar o projeto, navegar para pasta do mesmo e instalar o único pacote que precisamos para trabalhar com JWT no ASP.NET o `Microsoft.AspNetCore.Authentication.JwtBearer`.

```
dotnet new web -o JwtAspNetBlazor
cd JwtAspNetBlazor
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
```

Chave Privada

Lembra da explicação anterior, onde comentamos que apenas o servidor poderia descriptografar o Token devido uma chave privada que é utilizada tanto na geração quanto leitura do Token?

Chegou a hora de criar ela, e esta tarefa é relativamente simples, no caso, para fins didáticos vamos colocá-la em uma classe estática que ficará de fácil acesso para uso posteriormente.

```
// Configuration.cs

namespace JwtAspNetBlazor;

public static class Configuration
{
    public static string PrivateKey { get; set; } = "5+IV)E2gL3xCH2rNTElZ_at9(TbG1N(E=pH)29"
}
```

Caso queira aprender mais sobre classes estáticas, recomendamos nosso curso sobre Orientação a Objetos:

Fundamentos da Orientação a Objetos

Aprenda um dos paradigmas mais utilizados no mundo da programação de forma direta e objetiva, utilizando C# como linguagem.

🔗 <https://balta.io/cursos/fundamentos-orientacao-objetos>

balta.io

Fundamentos Orientação a Objetos



Em adicional, esta chave está “hard coded” ou seja, está fixa no código, o que não é uma boa prática, podemos (E devemos) fazer uso das configurações da aplicação para armazenar estas informações.

Também não devemos “comitar” este código de forma alguma, já que mais pessoas que estejam trabalhando no projeto teriam acesso a mesma chave.

Por fim, esta chave **NUNCA** deve ser a mesma de produção. Em produção a chave deve ser outra e ninguém deve ter acesso a ela.

Para saber mais sobre essa parte de configuração e ambientes, recomendamos nosso curso de Fundamentos do [ASP.NET 6](#).

Fundamentos do ASP.NET 6

Aprenda a construir APIs, implementar autenticação e autorização e desenvolvimento de serviços com o Framework mais amado pela comunidade.

🔗 <https://balta.io/cursos/fundamentos-aspnet>

balta.io

Fundamentos do **ASP.NET 6**



Para saber mais sobre a publicação e gerenciamento de configurações de diferentes ambientes como DEV, QA e PROD, recomendamos o nosso curso de Fundamentos do DevOps:

Fundamentos do Azure, Git, GitHub e DevOps

Aprenda a versionar e publicar pacotes e aplicações no Azure de forma totalmente automatizada utilizando o GitHub como plataforma.

🔗 <https://balta.io/cursos/fundamentos-azure-git-github-devops>

balta.io

Fundamentos Azure, Git, GitHub e DevOps



Apenas resumindo que o código anterior é apenas para fins didáticos, as boas práticas mostradas nos cursos acima devem ser sempre seguidas.

Criando um usuário

Movendo adiante, vamos criar um usuário, um objeto que vai conter as informações do usuário como E-mail, Id e perfis que ele possui.

Este objeto pode ser simples (Anêmico) ou complexo, novamente, para geração do Token não importa, só vamos utilizá-lo para transferência de dados.

```
// Models/User.cs

namespace JwtAspNetBlazor.Models;

public record User(int Id, string Email, string Password, string[] Roles);
```

Sinta-se à vontade para modificar este modelo como desejar, adicionando ou removendo propriedades, comportamentos e etc.

Como gerar um Token?

A real tarefa começa aqui, onde temos que gerar um Token e como vimos antes o mesmo será segmentado em três partes:

- Cabeçalho
- Payload
- Assinatura

A boa notícia é que tanto o cabeçalho quanto a assinatura podem ser gerados “automagicamente” pelo pacote que adicionamos anteriormente, sobrando apenas o PayLoad para nós.

O processo na verdade é bem simples, e todo realizado por uma classe chamada

`JwtSecurityTokenHandler`.

Esta classe possui os métodos `CreateToken` que literalmente cria um Token e `WriteToken` que encripta e gera a string que precisamos.

```
// Cria uma instância do JwtSecurityTokenHandler
var handler = new JwtSecurityTokenHandler();
```

```
...
// Gera um Token
var token = handler.CreateToken(tokenDescriptor);
// Gera uma string do Token
var strToken = handler.WriteToken(token);
```

No meio disso tudo, precisamos apenas gerar o PayLoad e informar a Chave Privada que ele irá utilizar para assinar o Token.

Assinando o Token

Nosso próximo passo é assinar este Token utilizando a Chave Privada que definimos anteriormente e que só o servidor terá.

Sem esta chave, como vimos anteriormente, é impossível alterar o conteúdo do Token, tornando-o inválido.

Este processo é relativamente simples e pode ser executado pelo `SigningCredentials`, onde precisamos informar uma chave simétrica (`SymmetricSecurityKey`).

O objeto `SigningCredentials` espera dois parâmetros, uma chave simétrica e o algoritmo que usará para encriptação.

```
new SigningCredentials(CHAVE, ALGORITMO)
```

Como vimos anteriormente, o algoritmo sugerido e amplamente utilizado até a escrita deste artigo é o Sha256, e temos seu valor descrito em uma constante chamada

`SecurityAlgorithms.HmacSha256Signature`.

```
new SigningCredentials(CHAVE, SecurityAlgorithms.HmacSha256Signature)
```

Estamos quase lá, falta apenas a chave simétrica, que será gerada utilizando o `SymmetricSecurityKey`. O problema é que este objeto espera um array de bytes como entrada ao invés de uma string.

```
// Converte nossa chave (string) para um array de bytes
var key = Encoding.ASCII.GetBytes(Configuration.PrivateKey);
```

```
new SigningCredentials(  
    new SymmetricSecurityKey(key),  
    SecurityAlgorithms.HmacSha256Signature)
```

Pronto, desta forma temos nossas credenciais para gerar e ler a assinatura dos Tokens gerados. Até o momento temos o seguinte código (ainda não é o final).

```
// Cria uma instância do JwtSecurityTokenHandler  
var handler = new JwtSecurityTokenHandler();  
  
var key = Encoding.ASCII.GetBytes(Configuration.PrivateKey);  
  
var credentials = new SigningCredentials(  
    new SymmetricSecurityKey(key),  
    SecurityAlgorithms.HmacSha256Signature)  
  
...  
  
// Gera um Token  
var token = handler.CreateToken(tokenDescriptor);  
// Gera uma string do Token  
var strToken = handler.WriteToken(token);
```

Token Descriptor

Como vimos anteriormente, o `TokenHandler` tem um método chamado `CreateToken` que espera um `tokenDescriptor`, que nada mais é um objeto que reune o cabeçalho, payload e assinatura do token.

```
handler.CreateToken(tokenDescriptor);
```

Nós já temos as informações de credenciais que conta com o algoritmo então já está meio caminho andado.

Podemos então criar uma instância do `SecurityTokenDescriptor` e informar as credenciais.

```
var tokenDescriptor = new SecurityTokenDescriptor  
{  
    SigningCredentials = credentials,  
};
```

Desta forma, já temos um Token que pode ser gerado, mas calma pois ainda faltam dois itens que são muito importantes.

```
var handler = new JwtSecurityTokenHandler();

var key = Encoding.ASCII.GetBytes(Configuration.PrivateKey);

var credentials = new SigningCredentials(
    new SymmetricSecurityKey(key),
    SecurityAlgorithms.HmacSha256Signature)

var tokenDescriptor = new SecurityTokenDescriptor
{
    SigningCredentials = credentials,
};

var token = handler.CreateToken(tokenDescriptor);
var strToken = handler.WriteToken(token);
```

Expirando o Token

Lembra que falamos sobre o tempo de vida de um Token? Pois é, chegou o momento que podemos definir este tempo de expiração.

Ele é feito através da propriedade `Expires` do `SecurityTokenDescriptor`, como demonstrado abaixo.

```
var tokenDescriptor = new SecurityTokenDescriptor
{
    Expires = DateTime.UtcNow.AddHours(2),
    SigningCredentials = credentials,
};
```

É importante notar que o uso das datas no formato UTC é altamente recomendado aqui, principalmente se os seus servidores trabalham em regiões diferentes.

Para saber mais sobre `DateTime`, formatos, globalização e UTC, recomendamos nosso curso de Fundamentos do C#.

Fundamentos do C#

Aprenda ou consolide todos os conhecimentos necessários para trabalhar com C# e .NET!

 <https://balta.io/cursos/fundamentos-csharp>

 balta.io

Curso completo
**Fundamentos
do C#**

GRATUITO

Em adicional, fique à vontade para alterar o tempo de vida do Token, no exemplo anterior utilizamos duas horas, mas como discutimos previamente neste documento, fica a seu critério definir quando este Token deve expirar.

Gerando o Payload

Nosso passo final é gerar o PayLoad, a carga, o conteúdo do nosso Token e isto pode ser feito de diferentes formas.

Minha recomendação é fazer uso dos `ClaimsIdentity` do ASP.NET por conta da integração que temos com eles.

Entendendo os Claims

Deixa eu te explicar melhor, no ASP.NET temos um conceito chamado de **Claim** ou traduzindo, uma “Afirmação”.

As Claims são objetos do tipo “Chave/Valor” que definem uma afirmação, como por exemplo, dizer que o nome do usuário é “André Baltieri”.

```
new Claim("Nome", "André Baltieri");
```

Criar um `Claim` é uma tarefa relativamente fácil, porém tem uma sacada que vai te ajudar muito a capturar estas informações depois.

Embora você possa colocar qualquer nome na “Chave”, o ASP.NET tem alguns nomes que são recomendados, e que se utilizados, já fazem um vínculo automático com o `HttpContext` durante a requisição.

Isto quer dizer que se ao invés de você utilizar uma string “Nome” como chave do Claim e utilizar a constante recomendada pelo ASP.NET que é `ClaimTypes.Name`, o ASP.NET será capaz de ler automaticamente este nome de usuário durante as requisições.

```
new Claim("Nome", "André Baltieri"); △  
  
public ActionResult Get()  
{  
    var user = User.Identity.Name; // NULL ✗  
}
```

No caso acima, o ASP.NET não consegue capturar o nome do usuário pois uma chave conhecida ou padrão não foi utilizada na geração do `Claim`.

```
new Claim(ClaimTypes.Name, "André Baltieri"); ✓  
  
public ActionResult Get()  
{  
    var user = User.Identity.Name; // André Baltieri ✓  
}
```

No exemplo acima, como utilizamos o `ClaimTypes.Name` como chave, o ASP.NET consegue automaticamente reconhecer a chave e buscar o nome do usuário quando utilizamos o `User.Identity.Name`.

O mais interessante é que temos diversos `ClaimTypes.Names` que funcionam desta forma, incluindo um dos que mais utilizamos que é o `Role`.

```
new Claim(ClaimTypes.Name, "André Baltieri"); ✓  
new Claim(ClaimTypes.Role, "premium"); ✓  
  
public ActionResult Get()  
{  
    var user = User.Identity.Name; // André Baltieri ✓  
    // Como definimos o role como Premium  
    // Agora podemos checar se o usuário pertence a este role  
    if(User.IsInRole("premium"))  
    ...  
}
```

Desta forma, tudo o que precisamos fornecer para o `TokenDescriptor` é um `ClaimsIdentity`, que nada mais é do que um conjunto de `Claims`, que como vimos anteriormente é apenas uma Chave/Valor.

Gerando um `ClaimsIdentity`

Para facilitar, vamos criar um método que recebe um `User` e gera os **Claims** baseados em seu nome e perfis.

```
private static ClaimsIdentity GenerateClaims(User user)
{
    var ci = new ClaimsIdentity();
    ci.AddClaim(new Claim(ClaimTypes.Name, user.Email));
    foreach (var role in user.Roles)
        ci.AddClaim(new Claim(ClaimTypes.Role, role));

    return ci;
}
```

No caso, estamos utilizando o E-mail como **Name**, mas sinta-se livre para utilizar qual propriedade desejar.

Em adicional, como temos vários **Roles** no `User`, precisamos iterar entre eles e popular o `ClaimsIdentity` com seus valores.

É importante notar que não há problema em ter múltiplas chaves do tipo `ClaimTypes.Role`, diferente do Name, Id ou Email.

Token Service

Para finalizar com chave de ouro, vamos colocar este código em um serviço no qual podemos instanciar e fazer uma chamada para gerar Tokens sempre que necessário.

No caso, recomendo criar esta classe na seguinte estrutura: `Services/TokenService.cs`

```
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Text;
using JwtAspNetBlazor.Models;
using Microsoft.IdentityModel.Tokens;

namespace JwtAspNetBlazor.Services;

public class TokenService
{
    public string Generate(User user)
    {
        var handler = new JwtSecurityTokenHandler();
        var key = Encoding.ASCII.GetBytes(Configuration.PrivateKey);
        var credentials = new SigningCredentials(
            new SymmetricSecurityKey(key),
```

```

        SecurityAlgorithms.HmacSha256Signature);

var tokenDescriptor = new SecurityTokenDescriptor
{
    Subject = GenerateClaims(user),
    Expires = DateTime.UtcNow.AddHours(2),
    SigningCredentials = credentials,
};

var token = handler.CreateToken(tokenDescriptor);
return handler.WriteToken(token);
}

private static ClaimsIdentity GenerateClaims(User user)
{
    var ci = new ClaimsIdentity();
    ci.AddClaim(new Claim(ClaimTypes.Name, user.Email));
    foreach (var role in user.Roles)
        ci.AddClaim(new Claim(ClaimTypes.Role, role));

    return ci;
}
}

```

Pronto, agora temos um serviço que gera Tokens baseado em um objeto `User` através do método `Create` e retorna o mesmo como uma String, pronto para enviar para tela.

Registrando o Token Service

Uma boa prática é registrar o serviço no **Service Locator** e assim poder fazer uso da injeção de dependência em qualquer lugar da nossa API.

Este é um processo relativamente simples e podemos utilizar o modelo Transiente já que não precisamos de um tempo de vida longo para nosso objeto.

```

// Program.cs
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddTransient<TokenService>();

```

Caso queira se aprofundar mais na parte de Injeção de Dependências, recomendamos nosso curso Dominando Injeção de Dependência.

Dominando Injeção de Dependência

Aprenda tudo sobre injeção de dependência, inversão de controle e DIP unindo teoria e prática.



<https://balta.io/cursos/dominando-injecao-de-dependencia>

Testando a API

Para evitar frustrações futuras, nada melhor que testar a API e ver se o Token está sendo gerado corretamente.

Nosso método vai ser extremamente simples, vamos receber um User do corpo da requisição e uma instância do `TokenService` que o **Service Locator** do `ASP.NET` se encarregará de encontrar.

Com ambos em mãos, podemos simplesmente retornar o resultado do `TokenService.Create` já que ele retorna uma string.

```
app.MapPost("/login", (User user, TokenService tokenService)
    => tokenService.Generate(user));
```

Lembrando que este é apenas um exemplo, não estamos tratando erros, nem normalizando retorno das APIs ou algo do tipo, é importante sempre seguir os padrões e recomendações para criação de APIs como mostramos no curso Fundamentos do `ASP.NET`.

O resultado final da nossa API até o momento é este código abaixo. Até que ficou enxuto e organizado.

```
using JwtAspNetBlazor.Models;
using JwtAspNetBlazor.Services;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddTransient<TokenService>();

var app = builder.Build();

app.MapPost("/login", (User user, TokenService tokenService)
    => tokenService.Generate(user));

app.Run();
```

Testando via Postman

Você pode testar a API chamando a URL <http://localhost:XXXX/login> via CURL, mas recomendo fazer o uso do Postman por ser mais completo.

Caso ainda não tenha o Postman instalado, você pode fazer o Download e instalação dele diretamente do seu site oficial: <https://www.postman.com>

Então vamos lá, execute sua aplicação com o comando `dotnet run` e abra o Postman para fazermos uma requisição do tipo POST.

The screenshot shows the Postman interface with a POST request to `http://localhost:5280/login`. The 'Body' tab is selected, showing the following JSON content:

```
1 {  
2   "email": "teste@balta.io",  
3   "roles": ["admin", "premium"]  
4 }
```

Note que a porta da URL será diferente no seu caso, e atente-se ao tipo da requisição que no caso é **POST**.

Para o corpo da requisição (**BODY**) selecionei **RAW** e nas opções selecionei **JSON**, adicionando o seguinte conteúdo.

```
{  
  "email": "teste@balta.io",  
  "roles": ["admin", "premium"]  
}
```

Como resultado, tive o seguinte Token como retorno.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1bmJxdWVfbmFtZSI6InRlc3RlQGJhbHRhLmlvIiwicm9sZSI6WyJ
```

Inspecionando o Token

Com o Token em mãos, nosso último passo nos testes é inspecionar o Token no <https://jwt.io>, simplesmente colando o valor que recebemos anteriormente no site.

Encoded PASTE A TOKEN HERE

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1bmldWVfbmFtZSI6InRlc3RlQGJhbHRhLmlvIiwicm9sZSI6WyJhZG1pbisInByZW1pdW0iXSwibmJmIjoxNjg0NTAzOTQ3LCJleHAiOjE2ODQ1MTExNDcsImlhCI6MTY4NDUwMzk0N30.qaJDssU17PuPWvfwj6NpbUiGuf30-JEfa41rhC61jAg

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE	
{ "alg": "HS256", "typ": "JWT" }	
PAYLOAD: DATA	
{ "unique_name": "teste@balta.io", "role": ["admin", "premium"], "nbf": 1684503947, "exp": 1684511147, "iat": 1684503947 }	
VERIFY SIGNATURE	
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), your-256-bit-secret) <input type="checkbox"/> secret base64 encoded	

SHARE JWT

✖ Invalid Signature

Se tudo deu certo, você verá as seguintes informações, contendo o “unique_name”, “roles” e “iat” conforme comentamos no começo deste documento.

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}  
  
{  
  "unique_name": "teste@balta.io",  
  "role": [  
    "admin",  
    "premium"  
  ],  
  "nbf": 1684503947,  
  "exp": 1684511147,  
  "iat": 1684503947  
}
```

Implementando autenticação na API

Com a API iniciada e o Token sendo gerado, chegou a hora de restringir o acesso a determinadas rotas da API utilizando autenticação.

Isto pode ser feito de forma simples e fácil no ASP.NET, através de Middlewares, no caso o `AddAuthentication` e `AddAuthorization`.

Estes dois Middlewares permitem dizer quem um usuário é e o que ele pode fazer na API. Neste ponto, podemos notar que gerar Tokens e proteger APIs são coisas que embora estejam ligadas, são distintas.

Poderíamos simplesmente parar por aqui e ter uma API (Microsserviço) apenas para gerar tokens. Até que não seria uma má ideia.

De qualquer forma, vamos começar pela autenticação, pois sem saber quem é o usuário, não podemos dizer o que ele pode fazer.

Adicionando autenticação

O processo de adicionar um Middleware ao pipeline de execução do [ASP.NET](#) normalmente se divide em duas partes, uma adicionando o Middleware (Add) e outra utilizando ele (Use).

```
// Adiciona o Middleware de autenticação  
builder.Services.AddAuthentication();  
  
// Usa o Middleware de autenticação  
app.UseAuthentication();
```

Outro ponto importante a se notar é “quando” devemos chamar este Middleware. O Middleware é um “passo” a mais na execução de uma requisição.

Basicamente estamos falando para o [ASP.NET](#) que agora também queremos verificar o cabeçalho da requisição para buscar um Token ou algo do tipo.

Então esta ordem importa! Supondo que você chame o Middleware de autenticação depois de ter registrado as rotas. Neste caso, seria inefetivo, pois as rotas já foram registradas quando não havia nenhum Middleware de autenticação.

Normalmente utilizo as chamadas de autenticação e autorização o mais breve possível, ou seja, o mais próximo possível da instância do `builder` e do `app`.

```
...
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddAuthentication();
...

var app = builder.Build();
app.UseAuthentication();
...

app.MapGet("/login", () => "Hello World");
...
```

Adicionando autorização

Com o Middleware de autenticação adicionado, vamos para a autorização, que vai seguir o mesmo exemplo.

Novamente é importante salientar a ordem dos Middlewares, lembre-se sempre que primeiro autenticamos e depois autorizamos, ou seja, precisamos saber quem é para depois dizer o que pode fazer.

Desta forma, o Middleware de autorização deve vir sempre após o Middleware de autenticação, seguindo conforme descrito acima.

```
...
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddAuthentication();
builder.Services.AddAuthorization(); ⚡
...

var app = builder.Build();
app.UseAuthentication();
app.UseAuthorization(); ⚡
...

app.MapGet("/login", () => "Hello World");
...
```

Seguindo o mesmo exemplo temos o `AddAuthorization` e `UseAuthorization`, que respectivamente fazem a adição e uso do Middleware de autorização.

Caso não haja necessidade, você pode dispensar o Middleware de autorização, por exemplo, se não for trabalhar com perfis, claims, policies ou qualquer outro nível de acesso, você precisa apenas da autenticação.

É importante lembrar também que os Middlewares são passos a mais na requisição, ou seja, quanto mais passos, mais demorada ela se torna, mas neste caso não precisa se preocupar tanto pois eles são bem otimizados.

Adicionando suporte ao JWT

Com autenticação e autorização prontas para serem utilizadas, vamos agora implementar seu uso, afinal, até agora só dissemos que queremos ter autenticação e autorização, mas não dissemos ao [ASP.NET](#) como fazer isto.

Bearer Authentication

Nosso primeiro passo então, é dizer ao [ASP.NET](#) que a nossa autenticação será feita utilizando um Token no formato JWT que vem no cabeçalho da requisição.

Este tipo de autenticação, onde informamos um Token é chamado de Bearer Authentication, por este motivo ouvimos muito falar em “Bearer JWT” ou “JWT Bearer”, pois enquanto Bearer define como o Token é trafegado, JWT define o tipo do formato do Token.

Apenas para intuito de conhecimento, o Bearer é um padrão amplamente utilizado, tanto no Backend quanto no Frontend, e seu principal destaque é a facilidade.

Tudo o que precisamos fazer para utilizar este padrão é definir um item no cabeçalho da requisição, chamado de `Authorization` e que contenha o valor `Bearer SEU_TOKEN`.

Como temos outros tipos de autorização, como `Basic`, onde apenas encriptamos o usuário e senha em `Base64`, é necessário especificar a palavra `Bearer` antes do Token.

Não esqueça que depois da palavra `Bearer` deve haver um espaço em branco e só então seu Token, como mostrado no exemplo abaixo.

```
curl  
--location 'http://localhost:5049/hello' \  
--header 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...'
```

Definindo o esquema de autenticação

Bom, já que sabemos como é o padrão **Bearer**, ficou fácil, é só interceptar a requisição, buscar pelo **Header**, encontrar o atributo **Authorization** e recuperar o valor que venha depois do espaço em branco após a palavra **Bearer**.

Na verdade é bem mais simples do que isto, pois como este é um padrão comum, o ASP.NET já consegue fazer tudo isto “automagicamente” para nós.

Tudo o que precisamos fazer é informar a ele qual esquema de autenticação estamos utilizando (Lembre-se que existem vários esquemas, então é necessário especificar o Bearer/JWT).

O método `AddAuthentication` pode receber algumas opções, dentre elas o `DefaultAuthenticateScheme` que queremos definir.

```
builder.Services
    .AddAuthentication(x =>
{
    x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
})
```

Neste caso, utilizamos a constante `JwtBearerDefaults.AuthenticationScheme` para definir que o esquema de autenticação que queremos utilizar é o **Bearer** e o Token virá no formato JWT.

Challenge

Como vimos anteriormente, é possível que nosso Token seja gerado de várias formas inclusive em sites externos, como Facebook, Twitter, Google e Microsoft.

Este processo é chamado de **Challenge** ou desafiar, onde confrontamos um servidor para que ele faça uma autenticação e nos retorne o Token.

Inclusive podemos ser este servidor e definir que após validar um Token redirecionamos o usuário para outro endereço.

Desta forma, nosso desafio aqui é dizer para o ASP.NET que estamos utilizando uma autenticação interna, ou seja, ela é gerada neste servidor e vale para este servidor apenas.

Pode este motivo, precisamos de uma configuração adicional, chamada `DefaultChallengeScheme` que definirá este processo.

```
builder.Services
    .AddAuthentication(x =>
{
    x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
```

```
        x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
    })
}
```

Desta forma temos todas as definições que precisamos, dizendo ao [ASP.NET](#) que nosso modelo de autenticação é o Bearer/JWT e que nossa autenticação é interna.

Lendo o Token

Para finalizar, as informações que precisamos, como nome do usuário e perfis que ele possui, estão dentro do Token.

Porém, o Token está encriptado e precisamos fazer o processo reverso feito no TokenService, para extrair as informações do Token e popular o `ClaimsPrincipal` (Usuário logado).

Já sabe né? Novamente o [ASP.NET](#) traz tudo pronto para nós, basta definir algumas configurações e dizer qual chave utilizamos para encriptar o Token.

```
.AddJwtBearer(x =>
{
    // Obriga uso do HTTPS
    x.RequireHttpsMetadata = false;

    // Salva os dados de login no AuthenticationProperties
    x.SaveToken = true;

    // Configurações para leitura do Token
    x.TokenValidationParameters = new TokenValidationParameters
    {
        // Chave que usamos para gerar o Token
        IssuerSigningKey = new SymmetricSecurityKey(Encoding.ASCII.GetBytes(Configuration.Pr
        // Validações externas
        ValidateIssuer = false,
        ValidateAudience = false
    };
});
```

Como resultado final, temos o seguinte código:

```
builder.Services
    .AddAuthentication(x =>
{
    x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
})
    .AddJwtBearer(x =>
```

```

{
    x.RequireHttpsMetadata = false;
    x.SaveToken = true;
    x.TokenValidationParameters = new TokenValidationParameters
    {
        IssuerSigningKey = new SymmetricSecurityKey(Encoding.ASCII.GetBytes(Configuration
            ValidateIssuer = false,
            ValidateAudience = false
        );
    });
    builder.Services.AddAuthorization();

    // ...

    app.UseAuthentication();
    app.UseAuthorization();
}

```

Restringindo rotas

Com as configurações devidamente feitas, vamos então restringir nossas rotas para acessos não autenticados.

Para isto, vamos utilizar o `RequireAuthorization`, ou `AuthorizeAttribute`, dependendo do tipo de projeto que esteja trabalhando.

```

// Exemplo em Minimal APIs
app.MapGet("/hello", () => "Hello World")
    .RequireAuthorization(); ⏪

```

Como podemos ver acima, no cenário em que estamos trabalhando com Minimal APIs, basta adicionar o `RequireAuthorization` no fim da chamada.

```

// Exemplo com MVC
[Authorize]
public Task<IActionResult> Get()
{
    ...
}

```

Já no MVC ou Razor Pages, basta adicionar o atributo `Authorize` no método e pronto, temos uma restrição na rota.

É importante notar que o embora o atributo faça referência a **autorização**, caso não seja especificado nenhum perfil, afirmação ou política, o mesmo serve apenas como **autenticação**.

Na prática, isto significa que receberemos o erro **401 - Unauthorized** quando tentarmos acessar algum endpoint com esta restrição.

Caso algum perfil, afirmação ou política seja explicitado e o Token não atenda o requisito, o erro recebido será o **403 - Forbidden**.

Obtendo o usuário logado

Bom, as rotas estão seguras, e vamos supor que recebemos uma requisição válida, com um Token válido, como descobrimos as informações de um usuário? Como obtemos valores do PayLoad do Token?

Este cenário é comum em diversos sistemas, imagina que estamos alterando informações de um usuário, onde ele só pode ver e alterar suas próprias informações.

Sabemos que ao seguir as recomendações para padronização da nossa API, recebemos o ID do usuário na rota, e então fazemos a alteração.

Mas e se o usuário informar o ID de outro usuário? E se por algum motivo a URL que ele recebeu tem um valor diferente do seu ID?

A maneira mais segura de fazer qualquer alteração que deve afetar apenas o próprio usuário, é utilizar as informações do Token.

Desta forma, nosso método que altera informações de um usuário sempre irá alterar o próprio usuário logado, não precisa (Nem adianta) passar o ID.

Para este exemplo, vamos tomar como base o seguinte Claim, atribuído ao `ClaimTypes.Name`, que facilitará nosso código.

```
new Claim(ClaimTypes.Name, "teste@balta.io");
```

Lembra que comentamos anteriormente que ao usar os `ClaimTypes` temos algumas vantagens especiais? Pois é, uma delas é que o `HttpContext`, um objeto especial no `ASP.NET` que contém dados da requisição é populado automaticamente.

Isto significa que algumas informações do PayLoad serão automaticamente carregadas para um objeto chamado `User`, dentro do `HttpContext`.

Como exemplo, temos o `User.Identity.Name` e `User.IsInRole`, dois métodos muito comuns e presentes no objeto `User` do `HttpContext`.

```
User.Identity.Name; // Nome do usuário (Que pode ser o ID ou E-mail)  
User.IsInRole("ROLE"); // Verifica se o usuário pertence a um role
```

Acessando o HttpContext

No caso dos projetos MVC e Razor Pages, o `HttpContext` é injetado automaticamente nos Controllers e Pages, tornando seu acesso bem simples.

```
[Authorize]  
public ActionResult Get()  
{  
    var userName = User.Identity.Name ?? string.Empty;  
}
```

A única coisa que precisamos nos atentar é que o usuário pode ser nulo, bem como o `Identity.Name`, então precisamos da coalescência nula para compensar este fator.

Em adicional, é bom utilizar este recurso apenas em métodos cujo atributo `Authorize` esteja presente, já que não faz sentido o método ser chamado se o usuário não for autenticado.

No caso das Minimal APIs, podemos injetar o `HttpContext` nos endpoints e a partir dele fazer a chamada para o objeto `User` que contém as informações que precisamos.

```
app.MapGet("/hello", (HttpContext context)  
    => Results.Ok(context.User.Identity?.Name ?? string.Empty))  
    .RequireAuthorization();
```

A mesma recomendação de chamada anterior se repete aqui, notando que é importante o uso do `RequireAuthorization` em métodos que farão uso do `User.Identity.Name`.

Autorizando as rotas da API

Até o momento temos uma API que consegue gerar Tokens e verificar se um Token válido foi recebido no cabeçalho, bem como ler informações do mesmo, como nome do usuário.

Embora este seja um importante passo para a API, ele não reflete as necessidades como um todo, pois muitas vezes temos que classificar os usuários.

Este ato de classificar é meramente rotular um usuário, dizer que de alguma forma, ele tem um determinado “poder” dentro da API.

Por exemplo, podemos dizer que o usuário batman@balta.io é o administrador e pode criar novos usuários, enquanto o usuário robin@balta.io pode apenas visualizar informações de um usuário.

Ainda podemos ter outros usuários como alfred@balta.io que além dos poderes do robin@balta.io ainda tem pode acessar a parte financeira da empresa.

Entendendo Roles, Claims e Policies

Isto nos trás para o conceito de Roles, Claims e Policies, onde Role é um Perfil, uma simples String que podemos usar para qualificar um usuário, um Claim é uma Afirmação, onde dizemos por exemplo que o usuário X possui um perfil Y.

O que realmente nos chama a atenção no ASP.NET são as Policies, ou políticas, que são literalmente trechos de código, que lidam com Roles e até acesso à dados.

Podemos criar uma política que exige que um usuário contenha um ou mais roles ou até mesmo que o usuário possua uma determinada informação como a data de nascimento preenchida.

É possível também acessar o banco de dados ou verificar um serviço externo durante a execução de uma política, mas muito cuidado com isto, pois as políticas são executadas sempre que uma rota com `RequireAuthorization` ou `AuthorizeAttribute` são chamadas.

No mais, opte sempre pelo simples e básico no começo, escale apenas se precisar, assim garantimos que a API roda o mais performático possível.

Criando Políticas

Criar uma Policy é um trabalho relativamente simples, bastando chamar o método `AddPolicy`, informando o nome e validações que esta política aplica.

```
AddPolicy("NOME_DA_POLICY", x => x.RequireRole("ROLE_OBRIGATORIO"));
```

Além disso, podemos criar várias políticas, para diferentes cenários e posteriormente identificar quais queremos utilizar.

```
AddPolicy("Administrador", x => x.RequireRole("admin"));
AddPolicy("Funcionario", x => x.RequireRole("employee"));
```

Registrando as políticas

Assim como fizemos uso do `AddAuthentication` para registrar o tipo de autenticação como JWT/Bearer e informar outras opções de autenticação, faremos uso do `AddAuthorization` novamente, informando as políticas como opção.

```
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("Admin", policy => policy.RequireRole("manager"));
    options.AddPolicy("Employee", policy => policy.RequireRole("employee"));
});
```

Neste caso, criamos duas políticas, uma chamada **Admin** e outra chamada **Employee**, para usuários administradores e funcionários, respectivamente.

Da mesma forma que a política **Admin** requer o perfil **manager**, a política **Employee** requer o perfil **employee**.

Utilizando as políticas

Para fechar a criação e uso de políticas, vamos aplicar suas restrições nas rotas da API, ainda utilizando o `RequireAuthorization` como fizemos anteriormente, porém, desta vez informando o nome da política.

```
app.MapGet("/employee", (ClaimsPrincipal user) =>
{
    return Results.Ok(new { message = $"Authenticated as {user.Identity?.Name}" });
}).RequireAuthorization("Employee");

app.MapGet("/manager", (ClaimsPrincipal user) =>
{
    return Results.Ok(new { message = $"Authenticated as {user.Identity?.Name}" });
}).RequireAuthorization("Admin");
```

Note que nos Minimal APIs podemos encurtar o caminho até o usuário logado, obtendo o `ClaimsPrincipal` ao invés do `HttpContext`.

Testando a API

Com tudo pronto, vamos testar a API, e para facilitar, deixei uma coleção com os endpoints já configurados no GitHub deste projeto.

segurança-em-apis-aspnet-com-jwt-e-bearer-authentication/endpoints.postman_collection at main · balta-io/segurança-em-apis-aspnet-com-jwt-e-bearer-authentication

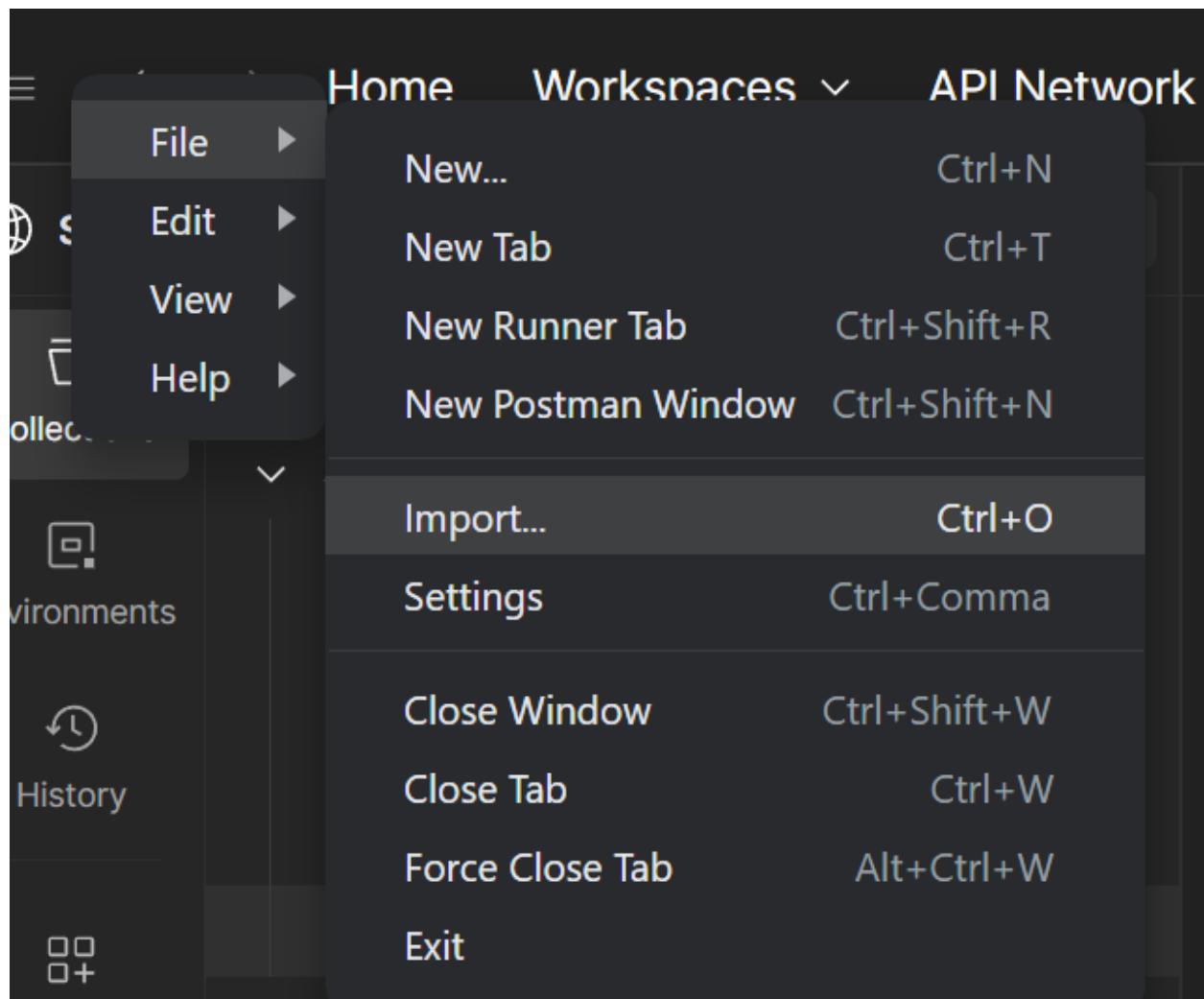
Repositório do eBook Segurança em APIs ASP.NET com JWT e Bearer Authentication - segurança-em-apis-aspnet-com-jwt-e-bearer-authentication · GitHub

 https://github.com/balta-io/segurança-em-apis-aspnet-com-jwt-e-bearer-authentication/blob/main/endpoints.postman_collection

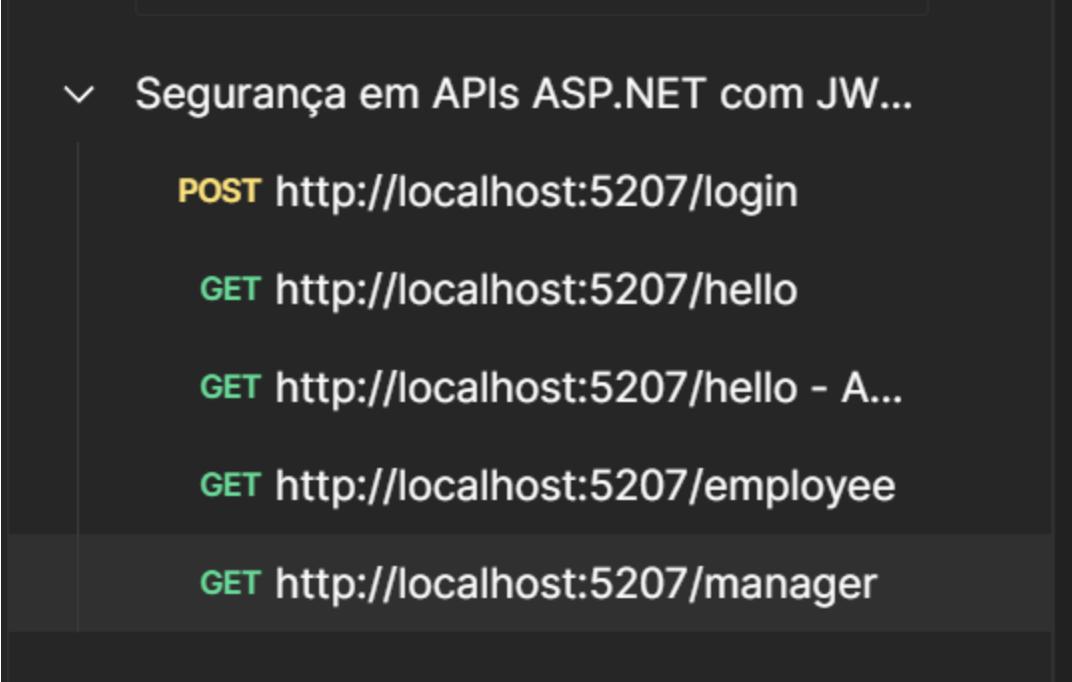
Basta rodar sua API com o comando `dotnet run` e executar os endpoints conforme mostramos abaixo.

Importando a coleção

O primeiro passo é importar a coleção no Postman, para isto vá em File > Import e selecione o arquivo `endpoints.postman_collection` baixado anteriormente.



Como resultado, você terá a coleção “[**Segurança em APIs ASP.NET com JWT e Bearer Authentication**](#)” adicionada ao menu lateral.



Segurança em APIs ASP.NET com JW...

POST http://localhost:5207/login

GET http://localhost:5207/hello

GET http://localhost:5207/hello - A...

GET http://localhost:5207/employee

GET http://localhost:5207/manager

Basta clicar em algum endpoint e fazer a requisição ou mesmo alterá-lo. Note que neste exemplo, meu App criou outra porta, a 5207, então talvez o seu esteja rodando em uma porta diferente também, basta alterar a URL.

Testando o login

Vamos começar pelo principal, e como não temos um sistema de senhas aqui, a composição dele fica como mostrado abaixo, basta clicar no endpoint e executar a requisição.

The screenshot shows a Postman interface with the following details:

- Method:** POST
- URL:** http://localhost:5207/login
- Body tab selected:** Contains a JSON object with fields: email (teste@balta.io), roles (admin, premium).
- Response tab (Pretty):** Shows a long JWT token.

Como resultado, temos um Token sendo gerado, salve este Token em algum lugar pois ele será necessário nas próximas requisições.

401 - Unauthorized

Ao tentar fazer uma requisição ao endpoint de funcionários, que está protegido, nos deparamos primeiramente com um erro 401 - Unauthorized.

Este erro indica que não estamos **AUTENTICADOS**, embora a mensagem diga “Unauthorized”.

The screenshot shows the Postman interface with a GET request to `http://localhost:5207/employee`. The **Authorization** tab is selected, showing "No Auth". A note below states: "This request does not use any authorization. Learn more about [authorization](#)". The response status is 401 Unauthorized.

Para informar o Token e realizar a autenticação, podemos mover para a aba Authorization, selecionar o tipo Bearer Token e informar o Token na caixa como mostrado abaixo.

The screenshot shows the Postman interface with a GET request to `http://localhost:5207/employee`. The **Authorization** tab is selected, showing "Bearer Token". A warning message says: "⚠️ Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaboration, recommend using variables. Learn more about [variables](#)". The **Token** field contains a long JWT token: `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ...`.

403 - Forbidden

Ao informar o Token, notamos um retorno diferente, com a mensagem “Forbidden” que remete ao acesso negado.

Esta sim é a mensagem referente sobre a autorização, afinal, temos um Token, estamos autenticados, mas não temos o perfil Employee, necessário para acessar a rota.

Para poder acessar esta rota, precisamos de um novo Token, com o perfil “employee”, e já sabemos como obtê-lo.

```

1 {
2   ...
3   "email": "teste@balta.io",
4   ...
5   "roles": [
6     ...
7     "admin",
8     ...
9     "employee"
10    ...
11  ]
12}
  
```

Com o Token em mãos, tudo o que precisamos fazer é atualizar o cabeçalho da nossa requisição para o endpoint de Employee e pronto!

```

1 {
2   ...
3   "message": "Authenticated as teste@balta.io"
4   ...
5 }
  
```

Agora temos acesso ao conteúdo do endpoint, com retorno 200 - OK, já que nosso Token possui o perfil Employee, obrigatório segundo a política do endpoint.

Código Fonte

GitHub - balta-io/seguranca-em-apis-aspnet-com-jwt-e-bearer-authentication: Repositório do eBook Segurança

Repositório do eBook Segurança em APIs ASP.NET com JWT e Bearer Authentication - GitHub - balta-io/seguranca-em-apis-
eBook Segurança em APIs A...

 <https://github.com/balta-io/seguranca-em-apis-aspnet-com-jwt-e-bearer-authentication>