UNIVERSIDADE
FEDERAL DO CEARÁ
CAMPUS DE CRATEÚS

Nota de Aula

Segurança da API com Json Web Token

MÓDULO I: FUNDAMENTOS DE PROGRAMAÇÃO PARA A CAMADA BACK-END

Prof. Dr. Bruno Honorato

Este nota de aula visa suportar a vídeo aula do curso em que discorro sobre como garantir a segurança dos endpoints de um web service implementado com ASP.NET em C#. As seções que se seguem abordam os conceitos apresentados na respectiva vídeo aula. A seguir, você irá estudar sobre: diferenças entre uma API Stateless e Stateful; divergência de autenticação em web services stateless e stateful; conceito de *Json Web Token* (JWT); geração de tokens JWT com o .NET.

1. API Stateless vs Stateful

Considere a diferença entre os princípios *stateless* e *stateful* como uma distinção inestimável no desenvolvimento de APIs e os serviços que usam esse tipo de aplicação. Nesta seção, abordo, de forma breve, o que esses termos realmente significam.

1.1 Stateful

Quando se fala sobre desenvolvimento de software, um “estado” (do inglês *state*) é simplesmente a condição ou qualidade de uma entidade em um instante no tempo. Gerenciar o estado de entidades em uma aplicação é registrar a valoração que compõe o estado no tempo e controlar a saída (retorno de um *endpoint*, por exemplo) de acordo com as entradas e estado registrados ao longo do ciclo de vida da API.

Eis um exemplo: considere o formato de codificação binária, cuja sintaxe se desdobra apenas em 1s e 0s. O que isso representa funcionalmente é “ativado” ou “desativado”. Agora, considere uma situação teórica em que você recebe um pedaço de papel com estas instruções simples - "se o número for 0, diga não, se 1, diga sim" - e você foi colocado em uma sala com um display binário que mudou entre o número 0 e 1 a cada cinco segundos. Este é um sistema com estado. Sua resposta dependerá inteiramente do relógio indicar “0” ou “1” , i. e., você deve responder conforme o estado da máquina de display binário.

Com isso em mente, considere um *web service* com monitoração de estado, i. e, um *web service stateful*. Digamos que você faça login por meio de uma requisição e, ao fazer isso, passe sua senha e nome de usuário. Se o *web server* armazenar esses dados junto a camada de *back-end* e usá-los para identificá-lo como um cliente constantemente conectado, o serviço pode ser considerado como *stateful*.

Conforme você usa o *web service*, tudo o que você faz está associado ao estado armazenado, que no caso, seria as suas configurações de login do lado *back-end*. Por exemplo: quando você solicita um resumo da conta, o *web service* poderia solicitar duas informações de entrada (e restringir o processamento ao recebimento desta entrada):

Quem está fazendo este pedido?
Conforme o ID armazenado para quem está fazendo essa solicitação, como deve ser renderizada a página no front-end?

Em um *web service stateful* (ou com monitoração de estado), a resposta processada a partir de uma simples requisição GET depende inteiramente do estado registrado pelo servidor. Sem o conhecimento desse estado, uma requisição pode não ser retornada corretamente.

1.2 Stateless

Stateless é o oposto de *stateful*, em que qualquer resposta dada do servidor é independente de qualquer tipo de estado.

Vamos voltar para o exemplo da sala com o display binário. Você recebe o mesmo relógio, só que desta vez o papel simplesmente tem um nome - “Parker” - e as instruções são para responder “Parker” quando alguém disser a senha “Peter”. Você fica sentado olhando o relógio mudar lentamente, e cada vez que alguém diz a senha, você diz o nome “Parker”.

Isso é ausência de estado - não há necessidade nem mesmo de fazer referência ao relógio ou observar o valor binário apontado no display, porque as informações são armazenadas localmente de forma que as solicitações sejam autocontidas - depende apenas dos dados que você possui. Sua resposta é independente do momento temporal ou de “0s” ou “1s”, e cada solicitação é independente.

Agora considere a rede social *Twitter*. Se você tem o Twitter instalado no seu *smartphone*, saiba que este aplicativo está constantemente utilizando um *web service stateless*. Quando você acessa o aplicativo e solicita uma lista de mensagens diretas recentes, na prática você está fazendo uma solicitação junto a API RESTful do Twitter e emitindo a seguinte requisição:

```
GET https://api.twitter.com/1.1/direct_messages.json?since_id=240136858829479935&cou
nt=1
```

A resposta que você obterá é totalmente independente de qualquer armazenamento de estado do servidor e tudo é armazenado no lado do cliente na forma de um cache.

2. Autenticação Stateless vs Stateful

A autenticação é um processo existente em quase todas as aplicações para identificar o cliente, seja um usuário ou outra aplicação. Em um *web service stateful*, a sessão de login do cliente é criada no lado do *back-end* e o ID de referência da sessão correspondente é enviada ao respectivo cliente do *web service*. Cada vez que o cliente faz uma solicitação ao servidor, o *web service* localiza a sessão junto a porção de memória primária alocada para persistir a respectiva sessão usando o ID de referência do cliente e encontra as informações de autenticação.

A seguir, são elencadas vantagens e desvantagens em se empregar o princípio *stateful* no desenvolvimento de um *web service*:

- **Vantagens**
    - Revogar a sessão a qualquer momento junto a um mecanismo **Identity Provider (IdP)** adotado no *web service*;
    - Fácil de implementar e gerenciar para cenário de servidor de uma única sessão;
    - Os dados da sessão podem ser alterados posteriormente.
- **Desvantagens**
    - Aumento da sobrecarga do servidor: conforme o número de usuários conectados aumenta, mais sessões serão criadas e mais memória primária deverá ser alocada.
    - Falha na escala: se as sessões são distribuídas em servidores diferentes, precisamos implementar um algoritmo de rastreamento para vincular uma sessão de usuário específica e o servidor de sessão específico.
    - É difícil para aplicativos de terceiros usarem suas credenciais: quando um aplicativo de terceiros permite que seus usuários façam login em seus sites, o aplicativo de terceiros não é capaz de verificar diretamente a sessão de seus usuários (eles são armazenados em seu *back-end*). A verificação deve ser redirecionada para os servidores de credenciais. Portanto, há mais trabalho entre o aplicativo de terceiros e o *back-end* da sua aplicação.

A autenticação seguindo o princípio *stateless* foi pensada para resolver as desvantagens da autenticação seguindo o princípio *stateful*. Por meio da autenticação *stateless*, um *web service* Y irá fornecer os dados da sessão do cliente para que estes fiquem persistidos do lado do cliente, i. e., na camada de *front-end* ou junto a aplicação que estará se integrando a Y. Os dados são assinados por uma chave do IdP adotado para garantir a integridade e autoridade dos dados da sessão. Como a sessão do usuário é armazenada no lado do cliente, o *web service* Y só tem a capacidade de verificar a validade desta sessão verificando se a carga útil (ou *payload*) e a assinatura correspondem ao mecanismo de teste de autenticação implementado em Y. A Figura que se segue ilustra uma resposta dada por uma requisição de autenticação stateless.

```
Payload:
{
  id: 1234,
  user: "kennethchoi",
  FirstName: "Kenneth",
  LastName: "Choi",
  Expiration: 1525132799 // 2018-04-30T23:59:59+00:00
}
Signature (a string) using a specific algorithm and the private key
to sign:
XxxxXXXxxxxXXXXXXxxxxXX
```

A seguir, são elencadas algumas vantagens e desvantagens de um serviço de autenticação orientado pelo princípio stateless:

- **Vantagens:**
    - Menor sobrecarga do servidor: o grande número de dados da sessão não é armazenado no servidor. Podemos armazenar mais propriedades do usuário nos dados da sessão do lado do cliente para reduzir o número de acessos ao banco de dados sem nos preocupar com a sobrecarga de memória no servidor.
    - Fácil de escalar: uma vez que os dados da sessão são armazenados no lado do cliente, não importa para qual servidor de *back-end* a solicitação é encaminhada, desde que todos os servidores de *back-end* compartilhem a mesma chave privada, todos os servidores têm a mesma capacidade de verificar o validade da sessão.
    - Bom para integração com aplicativos de terceiros: nos protocolos de logon único, os aplicativos de terceiros e o IdP devem ser capazes de se comunicar uns com os outros por meio de agentes de um cliente. Durante o processo de vinculação da conta entre aplicativos de terceiros e o IdP, o IdP envia uma mensagem assinada ao cliente e este redireciona essa mensagem para os aplicativos de terceiros. Usando um segredo (ou secret) compartilhado pré-configurado, os aplicativos de terceiros podem determinar se a vinculação da conta (logon único) é válida por si só.
- **Desvantagens:**
    - Não é possível revogar a sessão a qualquer momento: como a sessão do usuário é armazenada no lado do cliente, o servidor não tem direitos para excluir a sessão.
    - Relativamente complexo de implementar em um cenário de servidor de uma sessão: as vantagens da autenticação *stateless* é a escalabilidade.

- Os dados da sessão não podem ser alterados até o seu tempo de expiração: suponha que queremos adicionar a propriedade “Age” aos dados da sessão acima, provavelmente podemos pedir ao cliente para atualizá-los, mas não podemos ter certeza de que o cliente os atualiza, uma vez que sua sessão anterior os dados ainda não expiraram, então o cliente ainda tem a chance de fazer solicitações com dados de sessão antigos.

3. Introdução conceitual ao Json Web Token

Normalmente, os dados de um serviço de autenticação *stateless* são retornados de forma criptografada em uma cadeia de caracteres hexadecimal seguindo algum padrão estabelecido na literatura. A esta cadeia de caracteres hexadecimal podemos chamar de “token”. Um dos padrões mais conhecidos para se orientar a fabricação de tokens de acesso junto a um IdP é o JWT.

A criação de tokens JWT contempla dois conceitos fundamentais:

- Autenticação: é o ato de identificar que o usuário é quem ele diz ser, validando suas credenciais de acesso que geralmente são um login e uma senha. Também podem ser usados outros meios para identificar um usuário de um sistema, como por exemplo usando biometria ou cartões de identificação;

- Autorização: é o ato de verificar se o usuário pode ou não ter acesso à um recurso ou executar determinada ação dentro do sistema. Nesse ponto o usuário já foi identificado (autenticado) previamente. Normalmente usamos Roles ou Policies para autorizar o acesso à determinado recurso.

Segue uma exemplificação para reforçar o entendimento sobre autenticação e autorização: quando você entra em um condomínio empresarial e lá eles pedem pra você se identificar com algum documento, seja ele RG ou CPF, fazendo isso você está autenticado, você provou para a recepção do condomínio (utilizando seu documento) que você é realmente você. Uma vez autenticado, você deve informar que local do condomínio você deseja visitar junto à recepção. Em seguida, a recepção irá contactar o local perguntado se você realmente pode entrar Se o local liberar o seu acesso, você então estará autorizado a visitá-lo. Essa simples descrição conjectura um simples **esquema de autenticação/autorização**.

Essa noção é importante também que se vai desenvolver um *web service*, pois não queremos que os endpoints do nosso *web service* fiquem disponíveis para usuário não identificados e sem credenciais válidas de acesso, i. e., não queremos que anônimos tenham acesso às funcionalidades previstas no nosso *web service*. Pelo contrário, o correto é que apenas usuários autenticados e autorizados possam ter acesso aos recursos do *web service*. É aí que entra a fabricação de tokens de acesso seguindo o padrão JWT.

O JWT é um padrão RFC 7519 utilizado para realizar autenticação e autorização entre duas partes por meio de tokens, esse token é assinado digitalmente. Basicamente esse token é formado por três partes: **Header**; **Payload** e **Signature**.

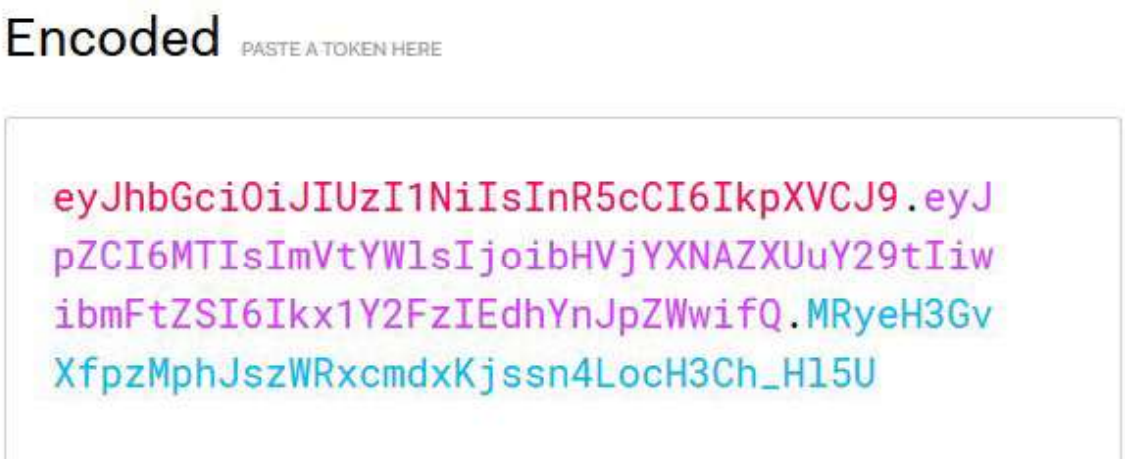
No nosso **Header**, nós temos as informações sobre a composição do token, no caso JWT (typ) e qual algoritmo de criptografía ele irá utilizar (alg).

O **Payload** é um objeto que contém as informações do usuário autenticado.

E por fim temos a **Signature**, que é a junção do hash nosso **Header** e nosso **Payload**, ambos em base64 junto com uma chave secreta definida pelo usuário. A chave secreta, ou *secret*, irá funcionar como uma semente para o algoritmo de criptografia gerar a sequência combinatória de caracteres que dará forma ao token.

Essa assinatura serve para garantirmos a integridade e veracidade do nosso token e para assegurar que realmente foi gerado pela sua aplicação e não é forjado.

Por fim temos um token contendo as 3 partes (**Header**, **Payload** e **Signature**) supracitadas e separadas por um caractere coringa (o ponto final):



Você também pode testar geração de tokens visitando https://jwt.io/ e configurando os seus próprios tokens.

4. Geração de tokens JWT em .NET

Em .NET, pode-se recorrer a classe *JwtService* de modo a simplificar a criação de um token JWT. Ela faz uso da classe *JwtSecurityTokenHandler* (pacote nuget System.IdentityModel.Tokens.Jwt) que é quem realmente irá gerar um JWT devidamente assinado e válido.

As informações contidas no token JWT são armazenadas em formato de *claims*, sendo assim, podemos usar o método *GetClaimsIdentity* passando uma instância de um usuário previamente autenticado através de suas credenciais de acesso para obter a identificação do usuário. Esse método irá ler as propriedades do usuário e criar um objeto *ClaimsIdentity*, que é armazenado na *claim Subject* representando a identidade do usuário dentro do token.

Basicamente as *claims* se desdobram em:

- *Issuer* (iss): quem emite o token JWT;
- *Audience* (aud): aplicações que podem usar o token JWT, normalmente temos apenas um valor, mas você pode informar mais de uma;
- *Expires* (exp): data e hora em que o token irá expirar;
- *IssuedAt* (iat): data e hora em que o token foi emitido.

Durante a criação do token informamos mais alguns dados, chamados de *Reserved Claims* segundo a especificação do JWT, que são atributos não obrigatórios (mas recomendados) usados na validação do token pelos protocolos de segurança das APIs.

Todo este arranjo de dados é encapsulado em uma classe chamada *JwtSettings*, onde uma instância dessa classe é criada no momento em que a aplicação se inicia, sendo ainda esta instância registrada no container de injeção de dependências.

De modo a garantir a segurança o token deve ser assinado digitalmente, sendo que para isso normalmente usamos algoritmos como HMAC ou RSA. Para assinar o token via RSA você precisa ter um certificado digital válido, já para a assinatura HMAC é necessário apenas uma chave privada.

Se um cliente de um *web service resteless* fornecer credenciais válidas junto a um endpoint de autenticação/autorização, a resposta deste endpoint deve ser um JWT. Com um token JWT devidamente criado, o cliente poderá ter acesso aos endpoints das classes Controllers que estejam protegidos conforme a semântica da anotação **[Authorize()]**. Para tal, o usuário deverá incluir no cabeçalho (ou header) de suas requisições, uma diretiva **Authorization** junto do token de acesso no formato JWT obtido precedido pela diretiva **Bearer**.

O .NET dispõe de um middleware reutilizável para validação de tokens JWT e configurável no método *ConfigureServices* da classe contida em *Startup*.cs. Eis um exemplo de definição e configuração do teste de validação de tokens JWT reutilizando o middleware previsto em *IServiceCollection*:

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme).AddJwtBearer(
    opt =>
    {
        var s = Encoding.UTF8.GetBytes(Configuration["SecurityKey"]);
        opt.TokenValidationParameters = new Microsoft.IdentityModel.Tokens.TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = Configuration["Issuer"],
            ValidAudience = Configuration["Audience"],
            IssuerSigningKey = new SymmetricSecurityKey(s)
        };

        opt.Events = new JwtBearerEvents
        {
            OnAuthenticationFailed = context =>
            {
                return Task.CompletedTask;
            },
            OnTokenValidated = context =>
            {
                return Task.CompletedTask;
            }
        };
    });
```

No exemplo acima, eu configurei o teste para que este validasse os tokens JWT com base em *claims* e na chave da assinatura definida em um arquivo de configurações *appsettings*. Para ver o exemplo na prática, basta consultar o video disposto no link que se segue:

https://www.youtube.com/watch?v=Gb81jHCkSes