Servidor de autenticação e autorização desacoplado

Hellen Desagiacomo, Leandro H. Campos,

Orientador: Walison B. Alves

[1]Curso de Sistemas de Informação – Centro Universitário UNIFAFIBE Bebedouro –SP – Brazil

bigsistemashellen@gmail.com, leandro_hcampos007@hotmail.com,
{sisunifafibe,sistemas}@unifafibe.com.br

Resumo. *Este artigo visa demonstrar a criação de um servidor de autorização e autenticação desacoplado implementando o framework OAuth 2.0 e todos seus fluxos de autorização. Ele também visa integrar com provedores de identidade externos, como Google e Microsoft, permitindo que usuários façam login com essas contas. Por último, é demonstrado o funcionamento de cada fluxo de concessão de autorização através de aplicações de teste.*

1. Introdução

Este projeto visa centralizar o processo de autorização e autenticação de usuários, para diversas aplicações de diferentes plataformas. Ao centralizar esse processo evita-se que o usuário final tenha que fornecer suas credenciais diretamente a aplicações de terceiros, um processo repetitivo que possui o grande risco de permitir acesso irrestrito a terceiros à conta em questão. Caso o usuário queira revogar acesso de alguma aplicação ele teria que trocar sua senha, o que resultaria na revogação de quaisquer outras aplicações onde ele estivesse autenticado.

Uma solução com menor impacto do que alterar credenciais e que seja mais segura é necessária, e essa segurança e praticidade pode ser atingida através da centralização do processo de autenticação e autorização. Através da centralização pode se obter mais controle, definir privilégios de acesso a outras aplicações, autorizar e revogar, encerrar sessões e outras medidas de

segurança que minimizariam o impacto de uma possível aplicação maliciosa. Eliminando o envio direto de credenciais do usuário e somando isso com o escopo de acesso uma suposta aplicação maliciosa teria seu estrago limitado.

O *Identity Server* 4 é uma biblioteca que implementa os protocolos *Open Id Connect* e *OAuth* 2.0, permitindo a criação de um servidor que centralize autenticação e autorização. Com essa ferramenta em mãos será possível criar um servidor customizado, seguro e seguindo os padrões utilizados por grandes empresas como *Google* e *PayPal*. A partir desta biblioteca, é possível criar um sistema que gerencie os usuários, clientes e recursos de forma que seja facilmente configurável e agilize o processo de integração com novos aplicativos.

2. Autenticação

De acordo com os editores do dicionário Michaellis (2017), autenticação tem os seguintes significados:

- Ato ou efeito de autenticar, de tornar autêntico.
- Selo, chancela ou abonação dados por notório ou tabelião, pelos quais se reconhece um documento como verdadeiro.
- Atestado de veracidade de algo.

O ato de autenticar, portanto, tem como intenção validar a veracidade de algo, no caso a identidade do usuário. De acordo com Erlich e Zviran(2015, p. 4248), os métodos de autenticação podem ser divididos nas seguintes categorias, de acordo com o fator validado:

- Autenticação baseada em conhecimento: valida-se o conhecimento do usuário através de perguntas, senhas dentre outras informações secretas.
- Autenticação baseada em posse: neste caso é necessário que o usuário porte algum objeto físico como um cartão, *token* ou dispositivo similar.
- Autenticação baseada em biometria: já neste caso a identidade do usuário será confirmada através de suas características físicas, como por exemplo um *scan* de retina ou impressão digital.

Na internet, a autenticação baseada em conhecimento é a mais utilizada, e é representada pelo clássico nome de usuário e senha. Mas desafiar o usuário uma única vez, exigindo somente suas credencias, não é o suficiente para garantir um grau aceitável de confiabilidade em todos os casos. Por isso utilizam-se métodos de autenticação mais forte, desafiando múltiplas vezes o mesmo usuário para tornar mais confiável o processo de autenticação (MIZRAH, 2003).

3. Autorização

Possui as seguintes definições, de acordo com os Editores do Dicionário Michaellis (2017):

- Ato ou efeito de autorizar; consentimento, permissão.
- Ordem ou determinação pela qual se autoriza ou se concede poder ou licença.
- Permissão oficial para que um indivíduo execute uma ação ou pratique determinado ato legal.
- Poder conquistado com essa permissão.
- Documento legal que registra por escrito essa permissão.

Tem papel fundamental após a autenticação do usuário, é nessa etapa que verificamos se o usuário tem permissão ou não para acessar o recurso requisitado.

4. Análise de protocolos e ferramentas disponíveis

4.1. OAuth 2.0

O *framework* de autorização *OAuth* 2 permite que uma aplicação terceira obtenha acesso limitado a um serviço HTTP utilizando a conta de um usuário. Nele a autenticação é delegada para o servidor que hospeda a conta do usuário, e autorizando a aplicação terceira a acessar seus recursos (HARDT,2012).

Na especificação do protocolo de autenticação *OAuth2*, elaborada por Hardt(2012), são definidos diversos papeis:

- ***Resource Owner*** (dono dos recursos): é geralmente o usuário final, que é capaz de dar autorização a determinadas operações da aplicação terceira através de um agente de usuário, como um navegador;
- ***Resource Server*** (servidor de recursos): Onde estão armazenados os recursos do usuário a serem consumidos pelo cliente, através do uso do*token* de acesso;
- ***Authorization Server*** (servidor de autorização): É o servidor que gera os *tokens* com base nas credenciais e escopo estipulados pelo *resource owner* (usuário final);
- ***Client*** (aplicação cliente): é aplicação que vai consumir a API do *resource server* (servidor de recursos) em nome de algum usuário.

No fluxo comum de autenticação entre aplicação cliente e servidor, o cliente precisa das credenciais do usuário para acessar algum recurso protegido no servidor. Para Hardt(2012), isso cria diversos problemas, entre eles:

- Exposição das credenciais para aplicações terceiras, que muitas vezes terão que salvá-las em texto legível para uso futuro.
- Servidores são obrigados a aceitar autenticação por usuário e senha, apesar dos riscos inerentes a este método.
- A aplicação terceira ganha amplo acesso a conta, sem que o usuário possa delimitar algum escopo de acesso para esta aplicação.
- O usuário dono dos recursos não pode revogar um acesso específico a sua conta, sendo forçado a trocar suas credenciais e, por consequência, revogar acesso de todas as aplicações que dependam dessas credenciais.
- O comprometimento de qualquer aplicação implica no comprometimento da credencial do usuário e quaisquer dados protegidos por ela.

Jones e Hardt(2012) explicam que é por este motivo que *bearer tokens* são utilizadas no *OAuth* 2. *Tokens*de acesso são *strings* que representam atributos de uma autorização, como usuário e senha, escopo de acesso, duração dentre outros. Geralmente, antes que uma aplicação cliente possa

acessar um recurso protegido, ela deve obter uma concessão de autorização do usuário, que em seguida é trocada por um *token* de acesso (e possivelmente um *refresh token*) que substitui o uso de credenciais a cada requisição.

Já um *token* de atualização, ou *refresh token*, substitui as credenciais na requisição de novos *tokens* de acesso quando estes se tornam inválidos ou expiram. Ele também pode ser utilizado para obter um *token* de acesso com escopo e tempo de vida diferente. *Tokens* de atualização são concedidos junto com *tokens* de acesso, a discrição do servidor de autenticação (HARDT, 2012).

O primeiro passo do *OAuth* 2 é conseguir a autorização do usuário, algo que pode ser feito utilizando 4 tipos de concessões.

4.1.1.*Authorization code grant*

No fluxo de concessão de autorização por código, também conhecido como fluxo de aplicações *web*, o usuário dono dos recursos é redirecionado da aplicação para o servidor de autorização. Este, por sua vez, checa se o usuário possui uma sessão ativa. Caso sim, o servidor de autorização irá pedir pela autorização do dono dos recursos. Se a autorização for concedida, o usuário será redirecionado de volta para a aplicação e um código de autorização será incluso como parâmetro da URL: http://www.example.com/oauth_callback?code=ABC1234

O código, por sua vez, é enviado para o servidor da aplicação cliente que o troca por um *token* de acesso (BOYD, 2012).

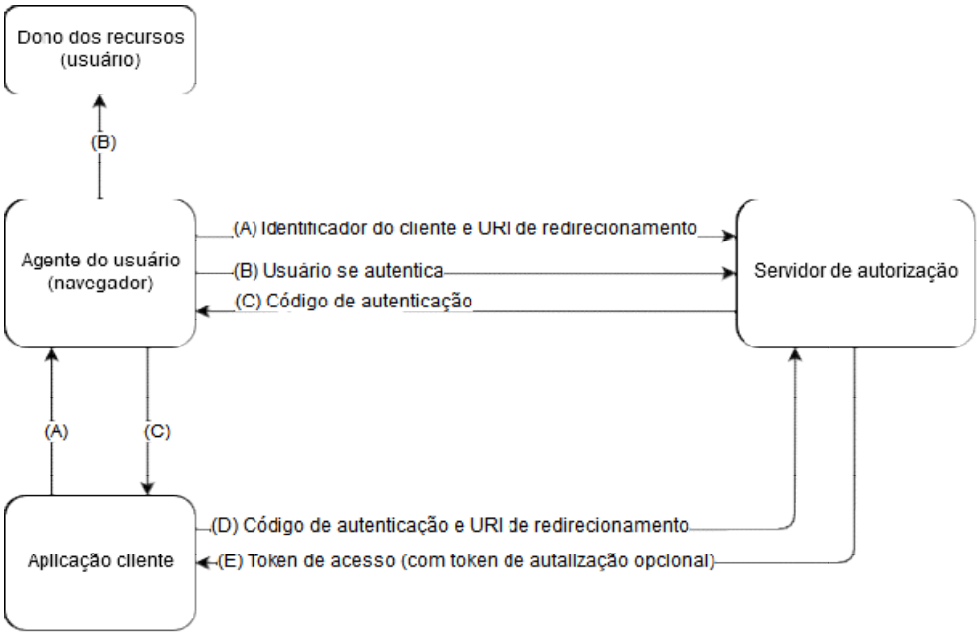


Figura 1. Fluxo da concessão por código de autorização, baseado no fluxograma da documentação de Hardt (2012).

Este fluxo é utilizado quando se precisa autenticar a aplicação cliente, além do usuário. A Figura 1 demonstra o fluxo desta concessão, e Hardt(2012) define o que acontece em cada passo:

- A: O cliente inicia o fluxo redirecionando o navegador do usuário para o servidor de autorização. O cliente inclui na requisição o seu próprio identificador, o escopo de acesso requisitado, estado local e a URI de redirecionamento para onde o usuário será redirecionado após permitir ou negar acesso.
- B: O usuário se autentica no servidor de autorização através de seu navegador e responde se permite ou nega acesso.
- C: Caso o usuário autorize, ele será redirecionado de volta a aplicação cliente usando a URI previamente enviada, mais um código de acesso e o estado local também previamente enviado.
- D: O cliente pede diretamente ao servidor de autorização um token de acesso. Ao mesmo tempo o cliente também se autentica, o cliente envia a URI de redirecionamento e o código de acesso para verificação.
- E: O servidor de autorização autentica o cliente, valida a URI enviada e o código de acesso. Se tudo estiver válido a aplicação cliente receberá um

token de acesso e, opcionalmente, um token de atualização para futuramente requisitar por novos tokens de acesso quando estes expirarem.

Por ser baseado em redirecionamentos, esse método requer um cliente capaz de interagir com o agente de usuário, muito provavelmente um navegador, que o dono do recurso está utilizando.

Para Hardt (2012), este fluxo proporciona algumas vantagens. Dentre elas está o fato de que o *token* de acesso não é exposto ao usuário e seu navegador, mas sim enviado diretamente do servidor de autorização para o servidor da aplicação cliente, o que minimiza as chances de expor o *token*de acesso a outros.

4.1.2.*Implicit grant*

Hardt(2012) define a concessão implícita como uma concessão por código de autorização simplificada, otimizada para aplicações cliente que rodam em navegadores usando linguagens como o *JavaScript*. Nessa concessão, a aplicação cliente recebe diretamente um *token* de acesso como resultado da autorização do usuário. Por não utilizar nenhum código de autorização como intermediário, essa concessão é implícita.

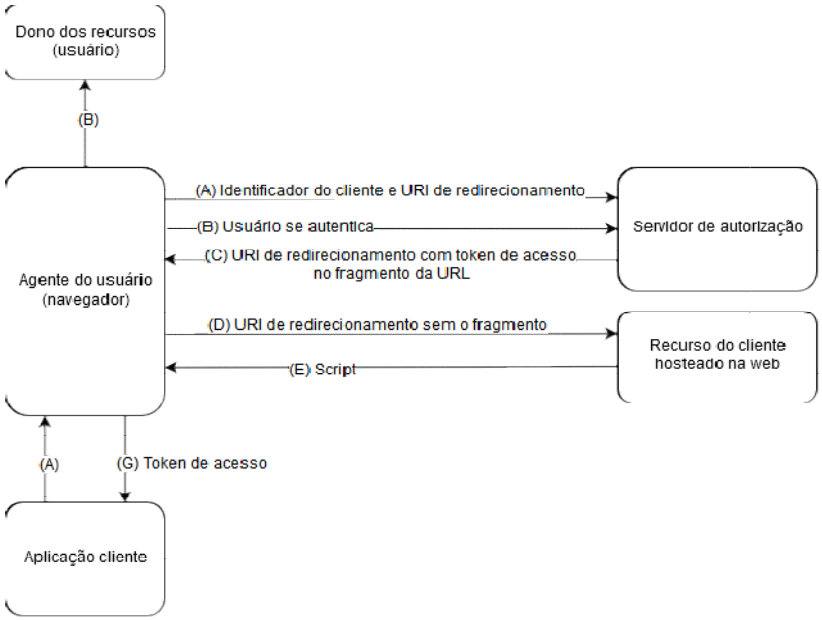


Figura 2. Fluxo da concessão implícita, baseado na documentação feita por Hardt (2012).

A seguir estão os passos da concessão, como descritos por Hardt(2012) na documentação e demonstrados na Figura 2:

- A: O cliente inicia o processo redirecionando o agente de usuário para o servidor de autenticação. Ele inclui seu identificador de cliente, escopo de acesso requisitado, estado local e a URI de redirecionamento para qual o navegador será redirecionado após o usuário permitir ou negar acesso.
- B: O usuário é autenticado no servidor de autorização através de seu navegador e permite ou nega acesso ao recurso protegido requisitado pelo cliente.
- C: Caso o usuário permita acesso, seu navegador é redirecionado para o cliente usando a URI previamente enviada junto com seu *token* de acesso no fragmento identificador.
- D: O navegador é redirecionado para a página cliente sem o token no fragmento da URI, que só é armazenado localmente.
- E: Após o get, a página cliente é carregada junto com seu script. Esse por sua vez consegue acessar a URI de redirecionamento completa (incluindo o token no fragmento).
- F: O script é executado e extrai o token e quaisquer outros parâmetros na URI de redirecionamento utilizada.
- G: O navegador entrega o token para o cliente.

A concessão implícita deve ser realizada quando se deseja um acesso temporário a algum recurso, nesta concessão *refresh tokens* não são concedidas para minimizar os riscos devido a exposição do *token* ao usuário, sua aplicação agente(o navegador) e até terceiras (BOYD, 2012).

4.1.3.*Resource owner password credentials*

As credenciais do dono dos recursos (usuário e senha) podem ser utilizadas para se obter um *token* de acesso e *refresh*, de preferência somente quando outros métodos mais seguros não estiverem disponíveis. Essa concessão só deve ser utilizada quando se confia muito na aplicação cliente a

ser autorizada, como aplicações *first-party*. Por utilizar as credenciais somente para fazer requisições pelos *tokens*, as credenciais em si não precisam ser salvas (HARDT, 2012).

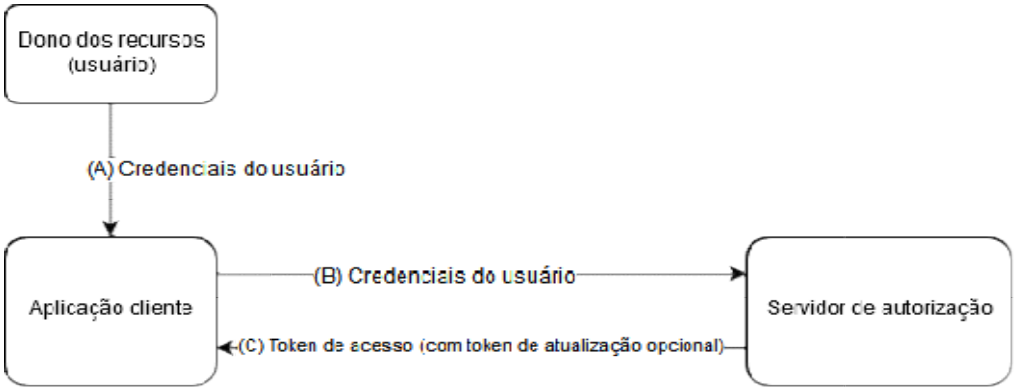


Figura 3. Fluxo da concessão por credenciais do dono dos recursos, baseado na documentação feita por Hardt (2012).

A seguir está a descrição dos passos dessa concessão, cujo fluxo foi exemplificado na Figura 3, de acordo com a documentação de Hardt (2012):

- A: O dono do recurso fornece seu usuário e senha a aplicação cliente.
- B: O cliente pede por um token de acesso usando as credenciais fornecidas. Ele também se autentica no servidor de autorização.
- C: O servidor de autorização autentica o cliente, valida as credenciais e, se tudo estiver correto, é retornado um token de acesso e opcionalmente um token de atualização.

4.1.4.*Client credentials*

O quarto e último fluxo utiliza as credenciais do próprio cliente, quando este está agindo por conta própria. Isso acontece quando a aplicação cliente tenta acessar um recurso do qual ela mesma seja dona, o que implica que o cliente neste caso também é dono do recurso, ou então quando a aplicação cliente tenta acessar um conteúdo previamente autorizado por algum usuário(HARDT, 2012).

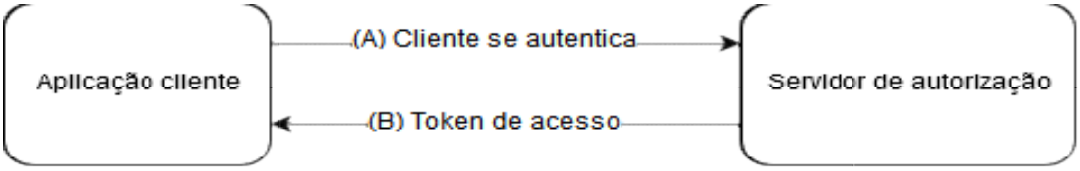


Figura 4. Fluxo da concessão por credenciais do dono dos recursos, baseado na documentação feita por Hardt (2012).

Logo em seguida está a descrição dos passos desta concessão, conforme o trabalho de Hardt(2012) e o que foi exemplificado na Figura 4:

- A: O cliente envia suas credencias ao servidor de autorização.
- B: O servidor valida as credenciais e, se tudo estiver certo, envia um token de acesso ao cliente.

4.2. OpenID Connect

A Fundação *OpenID*(2014) define o *OpenID Connect* como um protocolo de autenticação, implementado por cima das especificações do *framework OAuth* 2.0. Ele permite, através de *APIs REST/JSON*, autenticar usuários através de vários sites sem ter o trabalho de guardar senhas.

Mainka, Mladenov e Schwenk(2015) afirmam que, pelo fato do *framework OAuth*2.0 lidar somente com as autorizações,o *OpenIDConnect* foi criado para preencher esta falta de autenticação. Eles adicionam que o *OpenID Connect* já está sendo utilizado por empresas como *Google, Microsoft e Paypal*.

Em sua descrição do protocolo e do recurso de *Single Sign-On*, Mainka et al. (2015) afirmam que protocolos como o *OpenID Connect* substituem múltiplas autenticações em diversos provedores de serviço por uma única autenticação a um servidor chamado de *Identity Provider*, o Provedor de Identidade. Após essa autenticação, diversas chamadas a *APIs REST* são feitas, invisíveis ao usuário, onde *tokens* de acesso são entregues aos múltiplos provedores de serviço.

4.3. Identity Server 4

De acordo com Allen e Dominick(2016), o *Identity Server* é um *framework* que implementa *OAuth* 2.0 e *OpenIDConnect*, utilizando *ASP.NET Core*. Dentre suas características estão:

•Autorização como serviço para diversas aplicações, sejam elas *desktop*, *web*, *mobile* ou serviços.

•*Single sign-on* e*sign out*emmúltiplasaplicações.

•Controle de acesso para *APIs*, através da emissão de *tokens* de acesso.

•*Federation Gateway*, que irá intermediar o acesso à provedores externos, como *Azure Active Directory*, *Google* ou*Microsoft*.

•Foco na customização

5. Artigo

5.1. Ferramentas utilizadas

As ferramentas utilizadas para o desenvolvimento serão o *Visual Studio*, para desenvolver o servidor, os clientes teste e as integrações, e o *Sql Server* para a base de dados que irá armazenar, as aplicações, permissões e usuários.

5.2. Configurações e autenticação com *ASP.NET Identity*

O servidor desenvolvido funciona da seguinte forma, a aplicação cliente envia uma requisição e o usuário é direcionado para o servidor, que por sua vez, vai disponibilizar uma interface para *login*, e registro do usuário. A figura 1, abaixo, exibe as interfaces de log in e registro, destinadas ao usuário final.

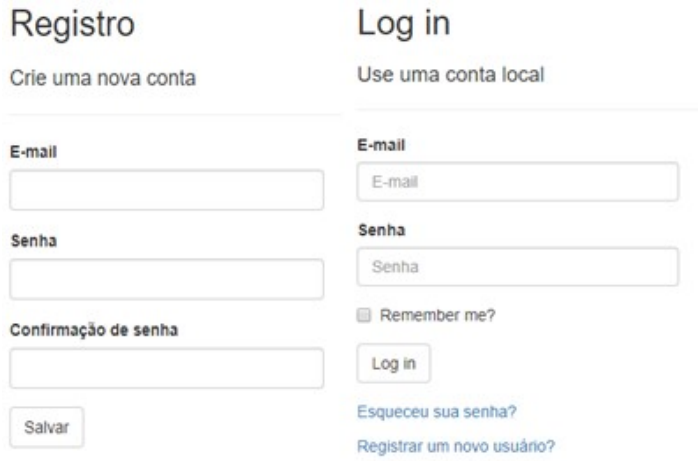


Figura 5. Interfaces para registro e *login* dos usuários

Durante o desenvolvimento utilizamos 4 *frameworks* fundamentais, são eles, o *Identity Server* 4, que é usado para a validação do acesso, autenticação e autorização, o *Entity Framework*, que é o responsável pela persistência e manutenção da base de dados, o *ASP.NET Identity* que é um modelo de *login*padrão da *Microsoft* e o *ASP.NET Core*2.0 que é uma versão *Open Source* do *ASP.NET* desacoplada do *.NET Framework*.

A base do projeto é a documentação oficial do *Identity Server* 4, realizamos diversas evoluções com customizações próprias de desenvolvimento.Um detalhe que notamos é que a documentação foi feita para o *ASP.NET Core 1.0*, o que limita a instalação de praticamente todos os pacotes relacionados ao *Identity Server* que utilizamos no desenvolvimento para a versão 1.1.2. Para utilizar a última versão disponível dos pacotes, é necessário iniciar um projeto novo, através do comando "*dotnet new mvc –auth Individual*", executado direto no *PowerShell*. Com o projeto criado desta maneira, todos os pacotes necessários como o *Microsoft.AspNetCore*, e todos os pacotes pertencentes a ele como por exemplo o, *Microsoft.AspNetCore.Authentication.OpenIdConnect* e o *Microsoft.AspNetCore.Authentication.Google* serão compatíveis com o servidor em sua última versão release 2.0.0.

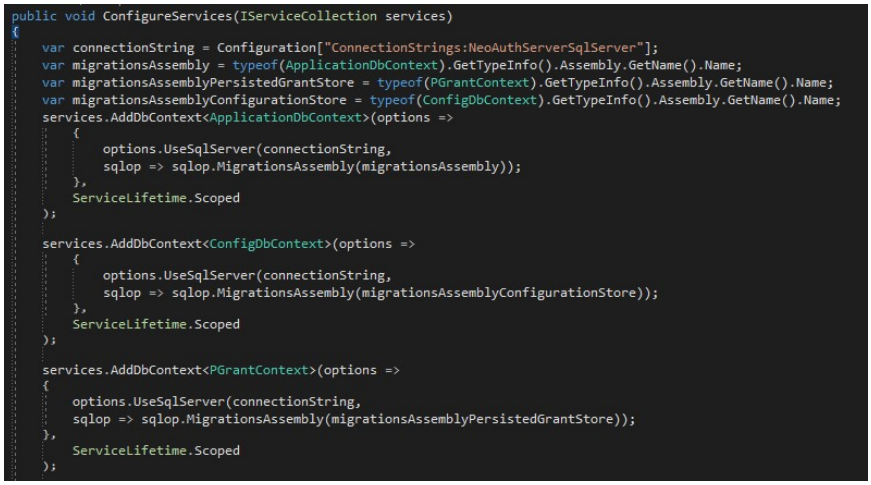
Ao desenvolver o projeto na versão citada acima, alguns métodos utilizados anteriormente se tornam obsoletos e outros métodos ainda não utilizados se tornam disponíveis para utilização, um exemplo claro disso é o construtor *IApplicationBuilderapp*. Segue abaixo um exemplo de sua utilização antes e depois:

- Versão 1.1.2: *app.UseIdentity*();
- Versão 2.0.0: *app.UseAuthentication*();

Antes as configurações eram feitas direto no pipeline, no método *Configure()*da classe *Startup*. Após a refatoração do ASP.NET Core e Identity Server 4, todas as configurações são feitas no método*ConfigureServices()*.

Para o funcionamento do projeto proposto, temos ainda serviços fundamentais, que podemos destacar, todos no método *ConfigureServices*. São eles:

- *services.AddDbContext* (adiciona e configura os serviços para a conexão com o banco de dados)
- *services.AddIdentity* (adiciona e configura os serviços do *ASPNET Identity*)
- *services.AddIdentityServer* (adiciona e configura os serviços do *Identity Server*)
- *services.AddAuthentication* (adiciona e configura as formas de autenticação do projeto)



```
public void ConfigureServices(IServiceCollection services)
{
    var connectionString = Configuration["ConnectionStrings:NeoAuthServerSqlServer"];
    var migrationsAssembly = typeof(ApplicationDbContext).GetTypeInfo().Assembly.GetName().Name;
    var migrationsAssemblyPersistedGrantStore = typeof(PGrantContext).GetTypeInfo().Assembly.GetName().Name;
    var migrationsAssemblyConfigurationStore = typeof(ConfigDbContext).GetTypeInfo().Assembly.GetName().Name;
    services.AddDbContext<ApplicationDbContext>(options =>
        {
            options.UseSqlServer(connectionString,
            sqlop => sqlop.MigrationsAssembly(migrationsAssembly));
        },
        ServiceLifetime.Scoped
    );

    services.AddDbContext<ConfigDbContext>(options =>
        {
            options.UseSqlServer(connectionString,
            sqlop => sqlop.MigrationsAssembly(migrationsAssemblyConfigurationStore));
        },
        ServiceLifetime.Scoped
    );

    services.AddDbContext<PGrantContext>(options =>
    {
        options.UseSqlServer(connectionString,
        sqlop => sqlop.MigrationsAssembly(migrationsAssemblyPersistedGrantStore));
    },
        ServiceLifetime.Scoped
    );
```

Figura 6. Configurações do banco de dados.

Na Figura 6 consta a configuração dos *DbContexts*, que servem para acessar a base de dados utilizando o *Entity Framework*. O *Identity Server 4* disponibiliza *DbContexts* já configurados, que podem ser herdados e extendidos conforme o desejado. O *ApplicationDbContext*é o contexto onde todas as informações do usuário são salvas. Já o *ConfigDbCotnext* é utilizado para salvar dados de configuração, como credenciais de *Clients*, *Resources* a serem protegidos

e informações sobre a identidade dos mesmos. Por último, o *PGrantContext* armazena informações de concessões de autorização. São utilizados três

DbContexts pois cada um pode estar em sua própria base, mas neste caso persistimos todos no mesmo banco.

```
services.AddAuthentication()
.AddGoogle(options =>
{
    options.SignInScheme = "Identity.External";
    options.AccessType = "offline";
    options.ClientId = Configuration["Autenticacao:Google:ClientI
    options.ClientSecret = Configuration["Autenticacao:Google:Cli
})
.AddMicrosoftAccount(options =>
{
    options.SignInScheme = "Identity.External";
```

Figura 7. Código para integração com provedores de identidade externos.

Na Figura 7, é possível observar os métodos chamados após *services.AddAuthentication()*, onde são configurados as formas de autenticação. Os parâmetros *ClientId* e *ClientSecret* são disponibilizados pelo provedor de identidade desejado para os desenvolvedores. Neste caso, escolhemos o *Google* e a *Microsoft.* Ao adicionar esse trecho de código as opções de integração, no caso exemplificado *Google*e *Microsoft*, aparecerão como botões na página de *login*, tal como na Figura 8.

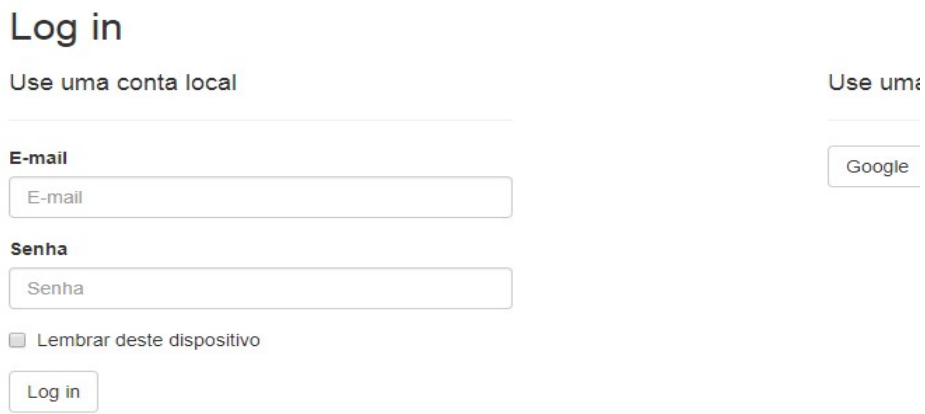


Figura 8. *Login* com conta local do *ASP.NET Identity* a esquerda, login com conta externa a direita.

Após o registro do usuário, ou caso o usuário já tenha se cadastrado, é efetuado o *login*. O usuário deve utilizar o *e-mail* e senha cadastrado ou clicar

no botão da integração escolhida para *login*, no fim do processo, o servidor enviará o retorno para a aplicação cliente e o usuário será redirecionado de volta para a página onde agora estará logado e com as permissões definidas para utilização.

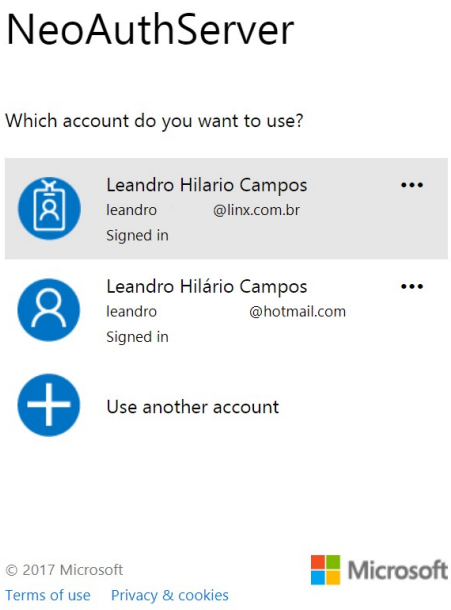
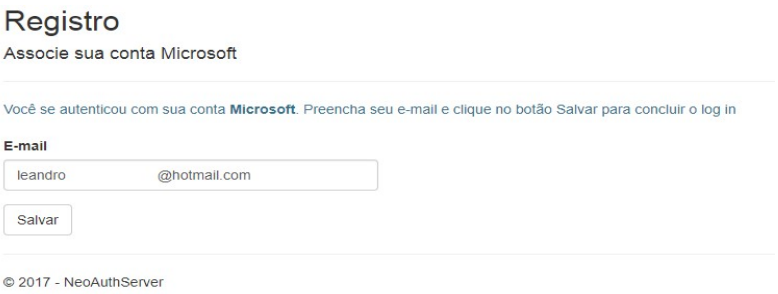


Figura 9. Tela exibida após selecionar o *login* usando uma conta *Microsoft.*

Após o usuário clicar no botão referente a conta que deseja utilizar, ele é redirecionado para o site do provedor externo. De forma invisível ao usuário, o servidor irá se autenticar utilizando o *ClientId* e *ClientSecret* fornecidos pelo provedor de identidade, conforme a configuração exibida na Figura 7 e o especificado na concessão de autorização por credenciais de cliente, um dos fluxos do *OAuth 2.0*.

O usuário poderá escolher a conta a ser utilizada neste processo, de



forma semelhante à Figura 10.

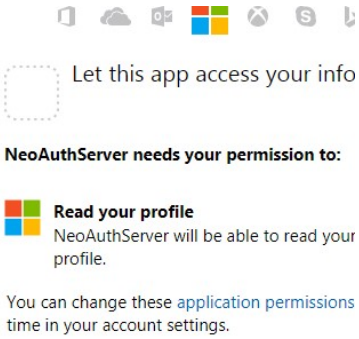


Figura 10. Tela de concessão de permissões

Caso seja a primeira vez acessando tal aplicação cliente (*NeoAuthServer* neste caso), será exibida uma lista de permissões necessárias semelhante a exibida na Figura 10. Por padrão, o *NeoAuthServer* precisa ler informações do perfil do usuário tais como e-mail, nome e afins, por isso durante o cadastro de nosso servidor junto a Microsoft requisitamos o acesso de leitura ao perfil do usuário.

Após escolher uma das opções, serão feitas chamadas entre os servidores (*NeoAuthServer*, o servidor deste artigo, e o servidor do *Outlook* neste caso). Essas chamadas seguirão o protocolo *OAuth 2.0*, mais especificamente a concessão por código de autorização, ou *authorization code grant,* conforme o descritona documentação da especificação *OAuth 2.0.* Isso quer dizer que será delegado um *token* de acesso diretamente ao nosso servidor, sem nenhuma exposição ao navegador utilizado. Esse *token* contém o escopo de acesso “leitura de perfil”, que nos permite trazer informações da conta Microsoft para criarmos nossa conta local.

Após a confirmação de e-mail do usuário, uma nova conta local será cadastrada a partir dos dados do provedor externo. Entretanto esta conta não possuirá uma senha, algo que deve ser definido depois caso ele queira efetuar o *login* sem usar sua conta do provedor externo.

5.3. Autorização com *Identity Server* 4

A estrutura do cliente utilizado neste teste está representada na Figura 11, onde é apresentado um objeto do tipo *Client*, salvo na base de dados através do *ConfigDbContext*.

```
new Client
{
    ClientId = "mvc.Hibrido",
    ClientName = "MVC HybridClient",
    AllowedGrantTypes = GrantTypes.HybridAndClientCredentials,

    RequireConsent = false,

    ClientSecrets =
    {
        new Secret("secret".Sha256())
    },

    RedirectUris           = { "http://localhost:5002/signin-oidc" },
    PostLogoutRedirectUris = { "http://localhost:5002/signout-callback-oidc" },

    AllowedScopes =
    {
        IdentityServerConstants.StandardScopes.OpenId,
        IdentityServerConstants.StandardScopes.Profile,
        "api1",
        "offline_access"
    },
    AllowOfflineAccess = true
},
```

Figura 11. *Objeto Client*.

As propriedades do objeto de classe *Client* exibidas na Figura 12 possuem as seguintes definições:

- *ClientId*: Identificador único da aplicação, preferencialmente um *guid.*
- *ClientName*: Nome de exibição da aplicação.
- *AllowedGrantTypes*: Tipos de concessões utilizadas. No exemplo da Figura 13 são informados dois tipos de concessões através da lista genérica *HybridAndClientCredentials*, eles são "*hybrid*" e "*client_credentials*". Isso significa que serão utilizadas as credenciais da aplicação cliente e as credenciais do usuário. A concessão híbrida é similar a autorização por código de autorização, em ambas o *token* de acesso é transmitido diretamente do servidor de autorização para o servidor da aplicação cliente. A concessão híbrida adiciona um *token* de identidade, que é transmitido para o navegador, ele contém informações da identidade do usuário.

- *RequireConsent*: Especifica se deverá ser exibida uma tela de permissões após o *login,* onde o usuário pode aceitá-las ou rejeitá-las.
- *ClientSecrets*: *Hashes* das senhas salvas, e suas datas de validade. Não é obrigatório.
- *RedirectUris*: São as *urls* válidas para onde os *tokens* podem ser enviados, a fim de evitar um ataque de redirecionamento.
- *PostLogoutRedirectUris*: Similar a *RedirectUris*, mas referente ao término da sessão.
- *AllowedScopes*: Lista de escopos que essa aplicação irá requisitar. São respectivamente o código do usuário, perfil, acesso a *API* de teste e acesso *offline*, em outras palavras, permissão para pedir *refresh tokens.*
- *AllowOfflineAccess*: Especifica se a aplicação pode requisitar *refresh tokens,* é necessário possuir acesso ao escopo "*offline_access*".

Para testar o presente trabalho e demonstrar a rotina de autorização, foi disponibilizado um cliente de teste. Nele foram feitas as configurações na classe *Startup,* exibidas na Figura 12, para que o cliente se autentique e autorize perante o servidor de autorização. Adicionalmente, deve-se incluir a linha *app.AddAuthentication();* antes da linha*app.UseMvc();* no método *Configure()* para que essas configurações sejam carregadas e utilizadas.

Figura 12. Configuração da autenticação do cliente no servidor de autorização.

Na página inicial da aplicação de teste é possível visualizar um *link* chamado “Recurso Protegido”, conforme pode-se observar na Figura 13.

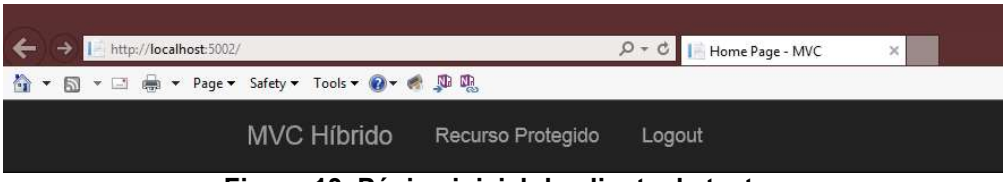


Figura 13. Página inicial do cliente de teste.

Ao clicar em “Recurso Protegido”, o cliente direciona o usuário ao servidor através de uma *url* similar a seguinte:

http://localhost:5000/account/login?returnUrl={parâmetros do cliente}

Nessa *URL* de retorno estão contidas as informações da aplicação cliente, a *URL* de redirecionamento, escopos necessários, fluxo de autorização utilizado dentre outras informações. Mas para que isso aconteça, deve ser adicionado a anotação *[Authorize]* acima do método da *Controller* respectiva, identificando que somente usuários autorizados terão acesso.

Após o processo de redirecionamento, o usuário efetuará o *login*, e o servidor enviará a resposta e direcionará o usuário de volta ao cliente. O usuário receberá, em seu navegador, o *token* de identidade do *OpenIdConnect,* enquanto o servidor da aplicação cliente receberá o *token* de acesso emitido pelo servidor de autorização.

Figura 14. Acesso ao recurso protegido que exibe os dados de acesso do usuário.

Na Figura 14, é possível observar que o usuário foi redirecionado para a aplicação cliente e a aplicação por sua vez, exibe ao usuário os seus dados de acesso. O primeiro deles é o *token* de identidade, para fins de autenticação e validações referentes a dados cadastrais, o *token* de acesso (trazido do servidor da aplicação) que é utilizado para acessar recursos protegidos, o *refresh token*, para ser possível emitir novos *tokens* de acesso, o *sid* que representa a sessão atual, *sub* que armazena o código único do usuário, o *idp* ou provedor de identidade que emitiu esses tokens, e por último o *name*ou nome do usuário.

A Figura 14, também exibe o botão “Chamar *API*”, que ao ser clicado faz uma requisição em uma *API* de recurso sendo protegida pelo servidor de autorização. A API de teste possui as seguintes configurações, conforme a Figura 15. Na configuração foi definido o nome da *API,* nome que deve constar na lista de escopos da aplicação cliente que tenta acessar este recurso, a autoridade (propriedade *Authority*) que aponta para o servidor de autorização, e que metadados HTTPS não são necessários. A política de autorização é definida no método *AddPolicy(),* de acordo com a Figura 15 onde é configurado uma política padrão onde usuários meramente autenticados podem acessar o recurso desejado. Essa política foi passada como parâmetro no método *AddCors(),* onde as configurações de requisições *AJAX* do tipo *CORS (Cross-Origin Resource Sharing)*são feitas.

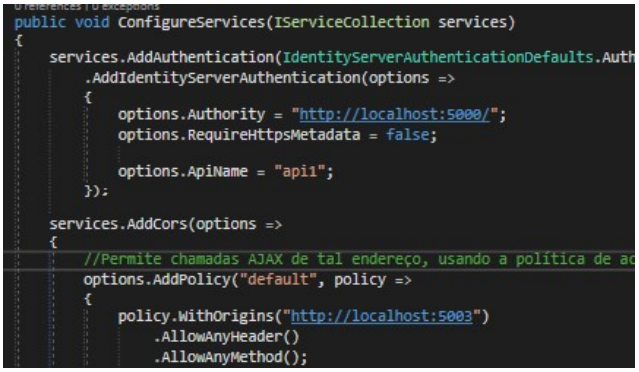


Figura 15. Configuração dos serviços de autenticação e autorização.

O método chamado retorna um *JSON*contendo decriptados dados do *token* de acesso. Essa*Controller* também possui a anotação *[Authorize],* indicando todos seus métodos só são acessíveis a quem satisfaça a política de acesso definida, como visto na Figura 16.

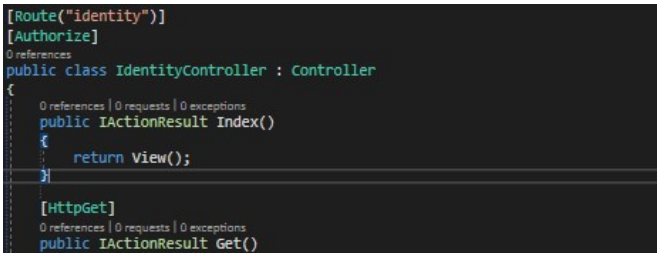


Figura 16. *Controller* protegida, com método que inspeciona dados do usuário da sessão.

O retorno deste método é exibido na página, conforme a Figura 17.

Figura 17. *JSON* de resposta da *API* protegida.

5.4. Resultados e conclusão

Através desta implementação do *Identity Server* 4, foi possível criar um servidor de autenticação e autorização utilizando os protocolos *OpenIdConnect* e *OAuth* 2.0 de forma complementar. Um usuário é capaz de, através deste servidor, aliviar a necessidade de lembrar-se de múltiplas senhas devido ao recurso de *single sign-on*, centralizando o acesso de múltiplos aplicativos à uma única credencial.

Em conjunto com a implementação do servidor, foi disponibilizado um *backoffice* com intuito de facilitar o gerenciar a base de dados, permitindo o cadastro de usuários através da biblioteca *ASP.NET Identity*, cadastros locais que podem ser feitos também a partir de uma conta externa já existente. Neste *backoffice* também está disponível um cadastro para aplicações cliente e recursos, para que seja possível delegar o acesso de recursos de usuários e controlar o escopo de acesso das aplicações clientes respectivamente.

Naturalmente as funcionalidades foram testadas através de aplicações clientes e uma *API* protegida. Cada aplicação cliente tem o intuito de testar um fluxo ou conjunto de fluxos específicos, utilizando tanto aplicações *ASP.NET Core MVC* quanto *Console Applications*. Por meio dos resultados obtidos nos cenários de teste, conclui-se que os objetivos foram atingidos, e a implementação possibilita que somente usuários autenticados e clientes cadastrados, possuindo os escopos corretos, consigam acessar páginas e recursos de acesso restrito. Portanto a aplicação do projeto é válida e útil em ambientes reais, onde pode ser utilizada para gerenciar credenciais e restringir acessos a recursos específicos.

Referências Bibliográficas

ALLEN, Brock; BAIER, Dominick. *Welcome to IdentityServer4 - IdentityServer4 1.0.0 documentation*. 2016. Disponível em: <http://identityserver4.readthedocs.io/en/release/index.html>. Acessadoem: 12 de mar. de 2017.

BOYD, Ryan. *Getting started with OAuth 2.0.* O'Reilly Media, Inc., 2012.

ERLICH, Zippy; ZVIRAN, Moshe. *Authentication Practices from Passwords to Biometrics.*Em: Encyclopedia of Information Science and Technology, Third Edition. IGI Global, 2015. p. 4248-4257.

HARDT, Dick. *The OAuth 2.0 authorization framework.*2012. Disponível em: <https://tools.ietf.org/html/rfc6749>. Acessadoem: 15 de mar. de 2017.

JONES, Mike; HARDT, Dick. *The oauth 2.0 authorization framework: Bearer token usage.*2012. Disponível em: <https://tools.ietf.org/html/rfc6750>. Acessado em: 11 de jun. de 2016.

MIZRAH, Len. *Strong authentication systems built on combinations of" what user knows" authentication factors*. U.S. Patent Application n. 10/431,396, 7 de maio de 2003.

MLADENOV, Vladislav; MAINKA, Christian; SCHWENK, Jörg. *On the security of modern Single Sign-On Protocols: Second-Order Vulnerabilities in OpenID Connect.*arXiv preprint arXiv:1508.04324, 2015. Disponível em: <https://arxiv.org/pdf/1508.04324.pdf>. Acessado em: 7 de jun. de 2017.

SAKIMURA, Nat; BRADLEY, John; JONES, Michael B.; MEDEIROS, Breno de; MORTIMORE, Chuck. *OpenID Connect Core 1.0 incorporating errata set 1,* 2014. Disponível em: <http://openid.net/specs/openid-connect-core-1_0.html> Acesso em: 4 de junho de 2017