Xavier Hahn · Seguir
26 de setembro de 2017 · 7 minutos de leitura

ASP.Net Core e Angular OpenID Connect usando Keycloak

Como configurar a autenticação do usuário para uma API da Web ASP.Net Core e um front-end Angular usando o servidor Keycloak para federação de usuários e OpenID Connect.

Manto-chave

Keycloak é um provedor de identidade de código aberto de propriedade da Red Hat. Ele permite adicionar autenticação facilmente a qualquer aplicativo e oferece recursos muito interessantes, como federação de usuários, intermediação de identidade e login social. No ecossistema .Net, um de seus "concorrentes" seria o Identity Server ou OpenIddict com Asp.Net Identity .

A principal diferença entre o Keycloak e as duas outras soluções mencionadas é que é um produto autônomo e não uma biblioteca. Isso significa que ele pode ser executado "por conta própria". É bom se você tem um ecossistema de aplicativos (talvez construído com tecnologias diferentes) e não quer fazer de um aplicativo o "mestre" que faz todo o gerenciamento de usuários. Também é um pouco mais completo, com uma interface de usuário de gerenciamento completa e suporte para multilocação.

Meu objetivo neste artigo é descrever como instalar uma instância simples do Keycloak, proteger o lado da API de um aplicativo desenvolvido com ASP.Net Core WebApi e proteger um aplicativo do lado do cliente Angular que usará o WebApi como backend.

Chega de falar, vamos começar

Instalando o Keycloak

O Keycloak foi construído com Java sobre o servidor de aplicativos Wildfly. Existe um procedimento de instalação bastante longo descrito na documentação . No nosso caso, queremos ter um servidor de teste simples para poder desenvolver nosso aplicativo de exemplo e faremos isso usando o Docker.

Não vou explicar aqui como instalar o docker, é bem simples, vá para a página inicial do docker se precisar de orientação. Vou apenas assumir que você já tem o docker instalado em sua máquina e está funcionando corretamente.

Existem algumas imagens docker disponíveis no Docker Hub. Vamos obter o pacote oficial "padrão" que é basicamente uma instalação do Keycloak pronta para usar, completa com o banco de dados e tudo mais. Você pode passar algumas variáveis de ambiente para criar diretamente a conta de administrador porque nenhuma é criada por padrão.

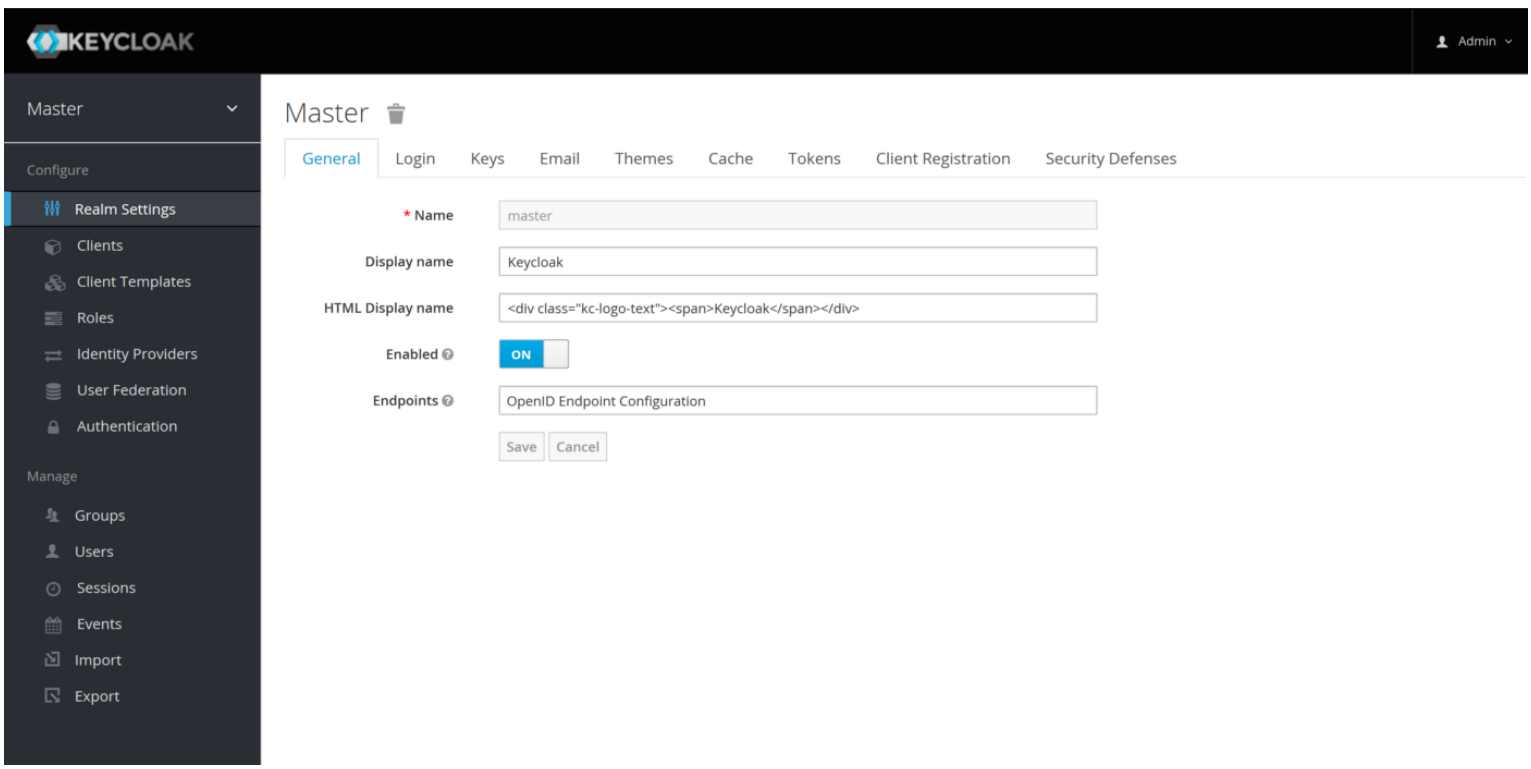
Observação: Eu não recomendaria usar essa instalação do Keycloak em produção, pois não há redundância e nem backup dos dados. Eu recomendo que você leia a configuração de alta disponibilidade do Keycloak na documentação oficial.

```
docker run --name keycloak -p 8080:8080 -e KEYCLOAK_USER=<USERNAME> -e KEYCLOAK_PASSWORD=<PASSWORD> jboss/keycloak
```

Isso anexará o Keycloak à porta 8080 da máquina host. Assim que a máquina iniciar, abra seu navegador e vá para localhost:8080. Ele deve exibir a página de destino do Keycloak.

A página de destino padrão do Keycloak

Clique no link do *Console de Administração* e conecte-se usando a combinação de nome de usuário e senha que você definiu no comando de execução do Docker. Agora você deve ver a página de administração do Keycloak.

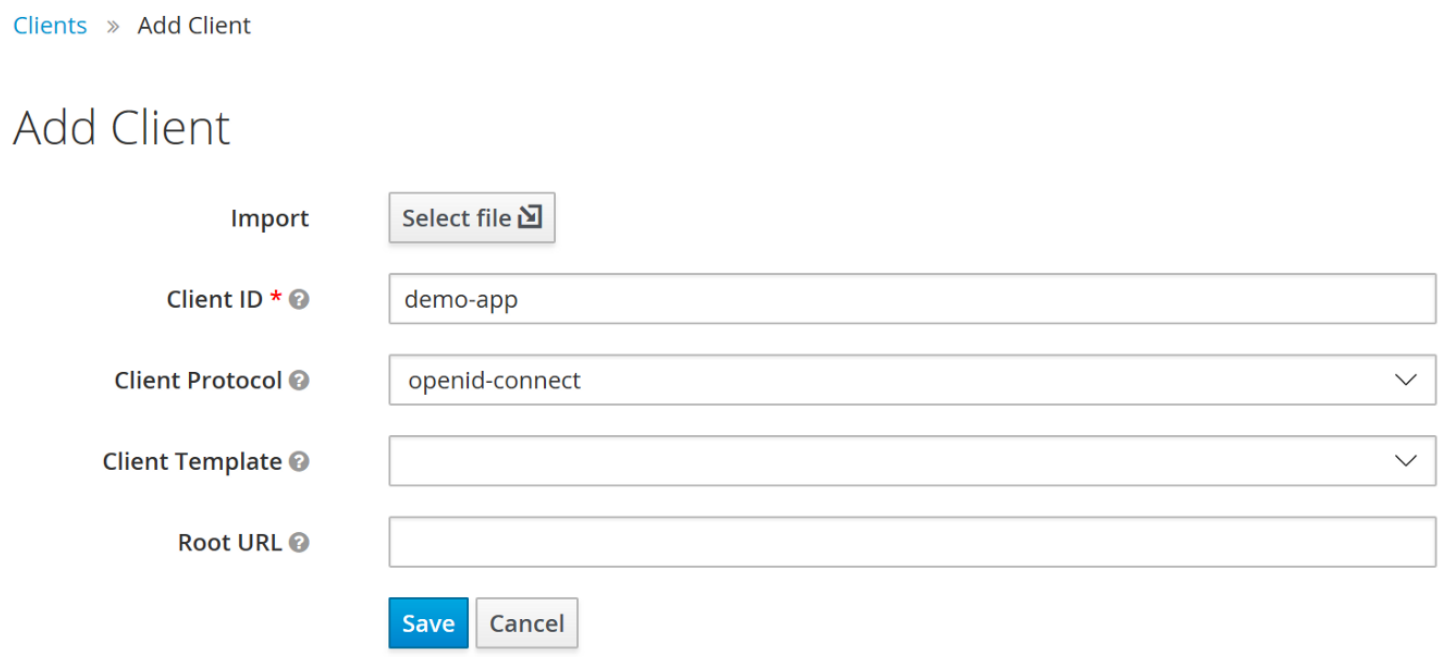


Página de administração do Keycloak

Sua instância do Keycloak agora está funcional e pronta para ser usada.

Configurando o Keycloak

Não vou entrar em muitos detalhes sobre como configurar o Keycloak neste artigo, a única coisa que precisamos fazer para proteger nosso aplicativo é criar um *Client* . Para fazer isso, clique na opção *Clientes* do menu à esquerda. Ele exibirá uma lista de alguns clientes pré-definidos. Clique no botão *Criar no canto superior direito.* No menu exibido, selecione um nome, por exemplo, *demo-app* e mantenha o protocolo do cliente como *openid-connect.*



O menu Adicionar cliente

Em seguida, certifique-se de configurar um URI de redirecionamento válido. Para simplificar, adicione * para permitir qualquer URI de redirecionamento. Habilite também o fluxo implícito marcando a caixa de seleção.

Criando o aplicativo Asp.Net Core + Angular

Para inicializar a criação do aplicativo Asp.Net Core + Angular, desde o .Net Core 2.0, agora existe um gerador que cria um aplicativo de página única com Angular diretamente da `dotnet` linha de comando.

```
mkdir demo-app
cd demo-app
dotnet nova instalação angular do
npm
```

É importante chamar o `npm install` comando após o término do gerador, ele não é executado automaticamente.

Execute o aplicativo usando `dotnet run` e verifique se tudo está funcionando conforme o esperado.

Protegendo a API

Agora vamos configurar o lado da WebApi para poder protegê-lo com base na autenticação e nas funções. Essa parte foi a mais confusa para mim, a princípio pensei que o próprio aplicativo teria que gerenciar os tokens e eu não conseguia entender como isso seria feito. Depois de algumas tentativas e erros e finalmente entendi que era função do Keycloak gerar e gerenciar os Tokens de autenticação e que a aplicação deveria simplesmente receber aquele token e “verificar” com o Keycloak se ele está de fato correto e vindo dele. Implementar essa parte é muito simples.

Primeiro, instale o pacote Microsoft.AspNetCore.Authentication.JwtBearer Nuget na solução.

```
dotnet adicionar pacote Microsoft.AspNetCore.Authentication.JwtBearer
```

Agora, abra o `Startup.cs` arquivo e adicione o seguinte ao `ConfigureServices` método

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
```

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.SpaServices.Webpack;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Http;

namespace demo-app
{
    public class Startup
    {
        public Startup(IHostingEnvironment env, IConfiguration configuration)
        {
            Configuration = configuration;
            Environment = env;
        }

        public IConfiguration Configuration { get; }
        public IHostingEnvironment Environment { get; }

        // This method gets called by the runtime. Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMvc();

            services.AddAuthentication(options =>
            {
                options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
                options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
            }).AddJwtBearer(o =>
            {
                o.Authority = Configuration["Jwt:Authority"];
                o.Audience = Configuration["Jwt:Audience"];
                o.Events = new JwtBearerEvents()
                {
                    OnAuthenticationFailed = c =>
                    {
                        c.NoResult();

                        c.Response.StatusCode = 500;
                        c.Response.ContentType = "text/plain";
                        if (Environment.IsDevelopment())
                        {
                            return c.Response.WriteAsync(c.Exception.ToString());
                        }
                        return c.Response.WriteAsync("An error occured processing your authentication.");
                    }
                };
            });
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
                app.UseWebpackDevMiddleware(new WebpackDevMiddlewareOptions
                {
                    HotModuleReplacement = true
                });
            }
            else
            {
                app.UseExceptionHandler("/Home/Error");
            }

            app.UseStaticFiles();

            app.UseAuthentication();

            app.UseMvc(routes =>
```

```
            {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{action=Index}/{id?}");

                routes.MapSpaFallbackRoute(
                    name: "spa-fallback",
                    defaults: new { controller = "Home", action = "Index" });
            });
        }
    }
}
```

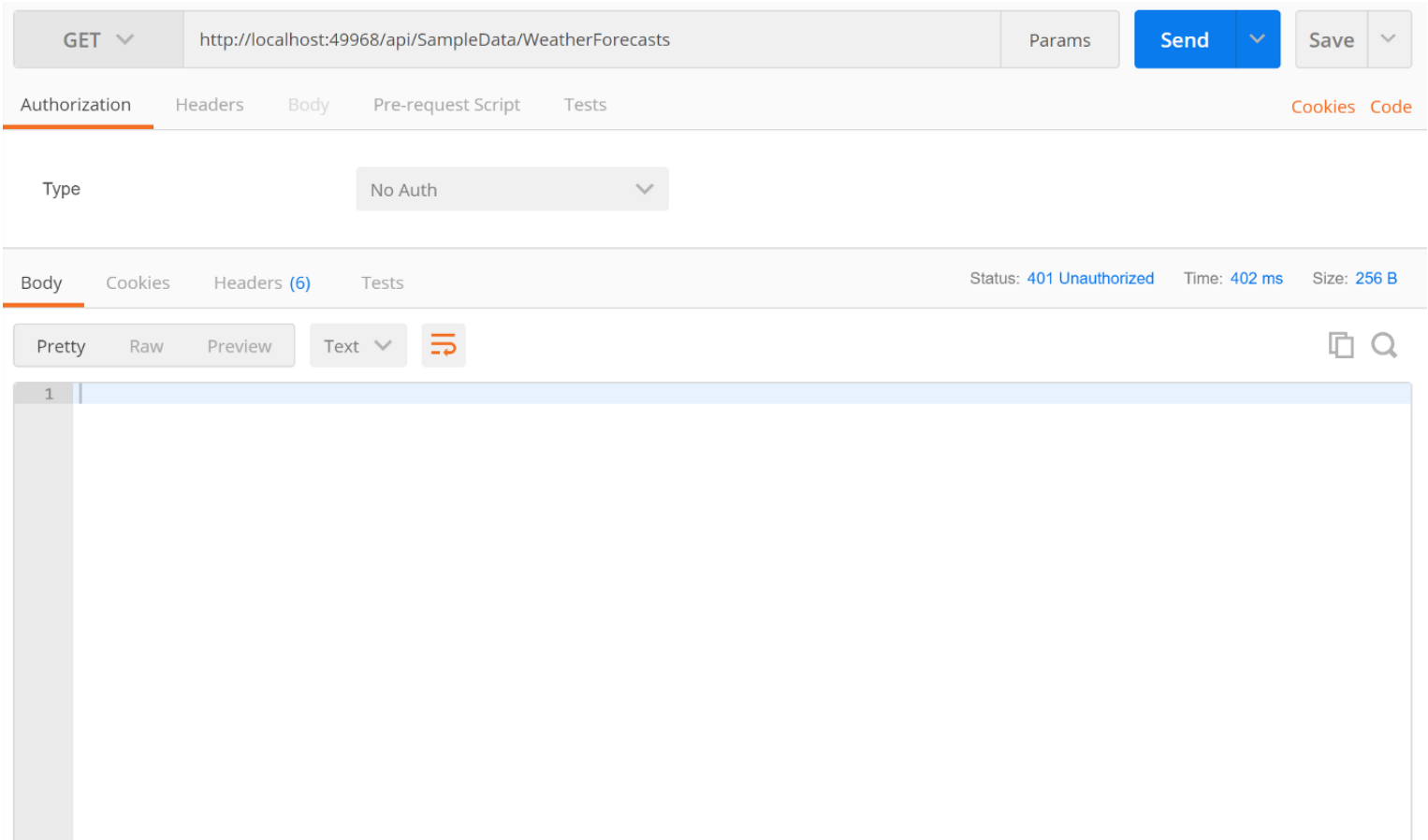
Você deve então adicionar a Autoridade e o Público em sua configuração. Autoridade é o URL de sua instância e domínio do Keycloak, em nosso caso `https://localhost:8080/auth/realms/Master`, e o Público é o nome do ID do cliente que criamos anteriormente: demo-app. Adicione isso ao seu arquivo de configuração, por exemplo, ou às variáveis de ambiente.

Agora, tudo o que precisamos fazer para proteger um endpoint de API é adicionar um `Authorize` atributo ao método endpoint. Isso adicionará uma verificação de que o método foi chamado com um cabeçalho de autenticação contendo um token JWT válido. Vamos aproveitar o código de demonstração existente e adicionar um atributo ao `SampleDataController` criado pelo scaffolding.

```
[Authorize]
[Route(“api/[controller]”)]
public class SampleDataController : Controller {...}
```

Você pode adicionar o atributo à classe (como eu fiz aqui), caso em que ele protegerá todos os métodos no controlador ou apenas em um método para proteger apenas um endpoint da API.

Vamos executar o aplicativo novamente e tentar chamar o `WeatherForecast` método do `SampleDataController`. Estou usando o Postman para fazer as chamadas da API REST.



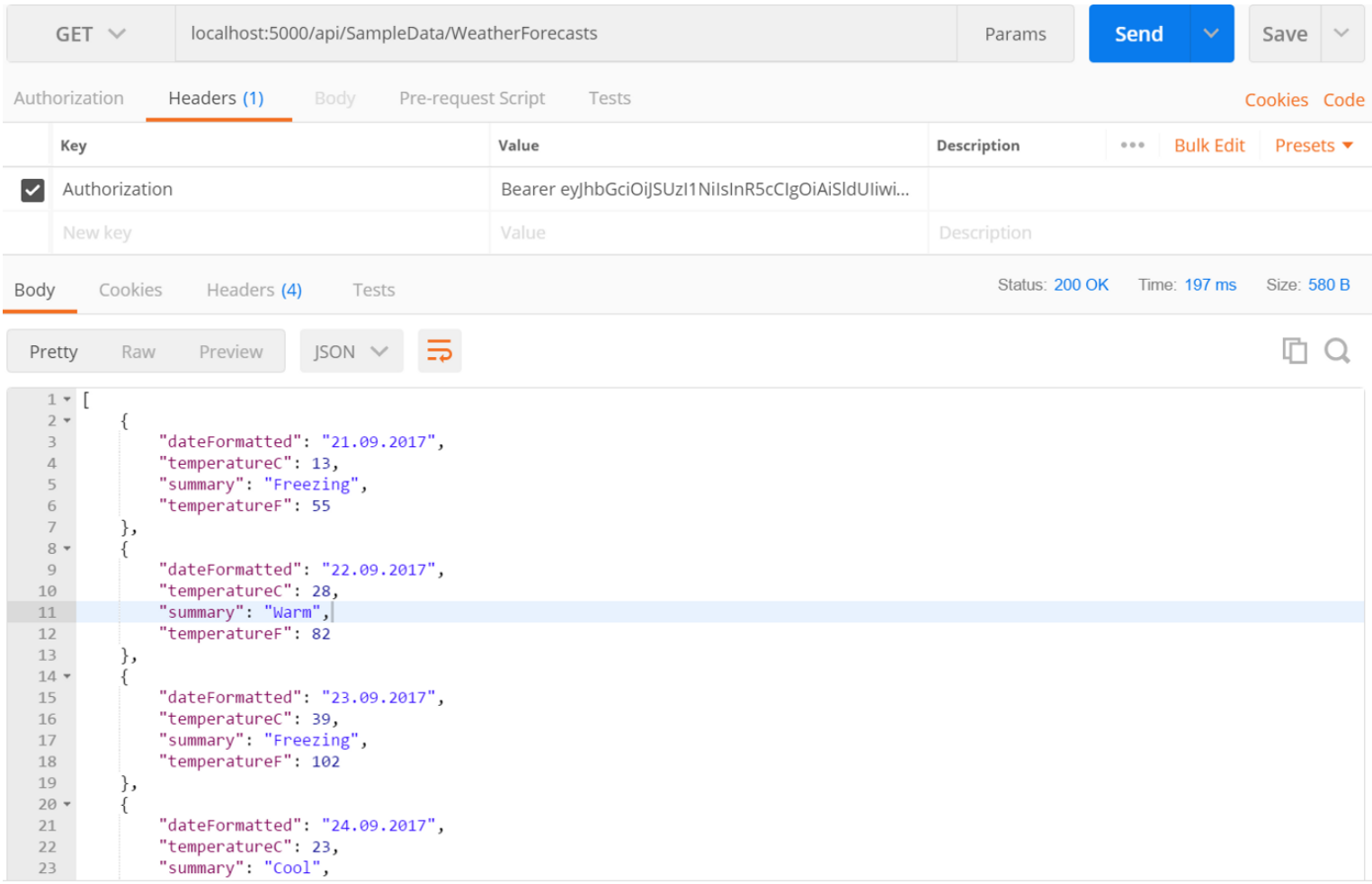
Resultado da chamada com o atributo [Autorizar] (401 Resposta não autorizada)

Podemos continuar nosso teste com o Postman solicitando um token ao Keycloak e adicionando-o à nossa solicitação. Clique na guia Autorização (na lista suspensa "GET") e selecione OAuth 2.0. Clique no botão *Obter novo token de acesso* e insira as seguintes opções:

Nome do token: demo
URL de autenticação: http://localhost:8080/auth/realms/master/protocol/openid-connect/auth
URL do token: http://localhost:8080/auth/realms/master/protocol/openid-connect /token
ID do cliente: aplicativo de demonstração

Em seguida, clique no botão *Solicitar token* . Deve haver uma janela do navegador exibida com o login do Keycloak. Insira suas credenciais e clique em Login. O Keycloak exibirá o token recém-criado na lista de *token existente* . Selecione-o, selecione *Adicionar token a: cabeçalho* e clique em *Usar token* . Ele será adicionado ao cabeçalho em um parâmetro de autorização.

Vamos chamar o método novamente e verificar se está funcionando conforme o esperado.



O resultado da chamada com o token JWT válido passado

Agora que a API está segura, vamos fazer a parte do Angular.

Implementando fluxo implícito em Angular

O fluxo implícito do OpenId connect é baseado no redirecionamento do usuário para a página de login do provedor de identidade e usa URIs de redirecionamento para verificar a identidade do cliente. É o único "fluxo" que não requer o uso de um segredo. Isso significa que é o único fluxo válido que pode ser usado por um aplicativo puramente do lado do cliente, como estamos escrevendo com Angular, que não pode realmente manter um segredo.

Para implementar o fluxo implícito, vou usar uma biblioteca de código aberto. Existem duas bibliotecas que encontrei e que são certificadas pela fundação OpenId, e elas têm um nome muito parecido: angular-auth-oidc-client de Damien Bowden e angular-oauth2-oidc de Manfred Steyer . Ambos são muito semelhantes em sua implementação e ambos os projetos são bem apoiados. Aqui, usarei a biblioteca angular-oauth2-oidc. Muitas das explicações e exemplos de código vêm do Readme da biblioteca.

Vamos começar instalando a biblioteca usando o NPM

```
npm instala angular-oauth2-oidc --save
```

Então, você precisa importar o módulo no seu `app.module` caso, vamos adicioná-lo ao `app.module.shared.ts` arquivo. Basta adicionar a importação do `OAuthModule` e adicioná-lo à `NgModule` propriedade de importação.

```
import { OAuthModule } from 'angular-oauth2-oidc';
[...]

@NgModule({
  imports: [
    [...]
    HttpModule,
    OAuthModule.forRoot()
  ],
  declarations: [
    AppComponent,
```

```
      HomeComponent,
      [...]
    ],
    bootstrap: [
      AppComponent
    ]
  })
  export class AppModule {
  }
```

app.module.shared.ts hosted with ❤ by **GitHub** **view raw**

Em seguida, vamos criar um arquivo na `ClientApp/app` pasta e chamá-lo `auth.config.ts.` Ele irá exportar a configuração para o Oidc.

```
import { AuthConfig } from 'angular-oauth2-oidc';

export const authConfig: AuthConfig = {

  // Url of the Identity Provider
  issuer: 'http://localhost:8080/auth/realms/master',

  // URL of the SPA to redirect the user to after login
  redirectUri: window.location.origin,

  // The SPA's id.
  // The SPA is registerd with this id at the auth-server
  clientId: 'demo-app',

  // set the scope for the permissions the client should request
  // The first three are defined by OIDC.
  scope: 'openid profile email',
  // Remove the requirement of using Https to simplify the demo
  // THIS SHOULD NOT BE USED IN PRODUCTION
  // USE A CERTIFICATE FOR YOUR IDP
  // IN PRODUCTION
  requireHttps: false
}
```

auth.config.ts hosted with ❤ by **GitHub** **view raw**

Em seguida, precisamos configurar o serviço oauth quando o aplicativo for iniciado. Vamos modificar o `app.component.ts` arquivo. Nós apenas injetamos a `OAuthService` do construtor e chamamos um método para configurar a configuração que criamos acima e tentamos o login.

```ts
import { Component } from '@angular/core';
import {
  OAuthService,
  JwksValidationHandler } from 'angular-oauth2-oidc';
import { authConfig } from '../../auth.config';
@Component({
  selector: 'app',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  constructor(private oauthService: OAuthService) {
    this.oauthService.configure(authConfig);
    this.oauthService.tokenValidationHandler =
      new JwksValidationHandler();
    this.oauthService.loadDiscoveryDocumentAndTryLogin();
  }
}
```

app.component.ts hosted with ❤ by **GitHub** **view raw**

O próximo passo é adicionar o menu de login à barra de navegação. Vamos editar o `navmenu.component.ts` arquivo e criar métodos de login e logout.

```ts
export class NavMenuComponent {
   constructor(private oauthService: OAuthService) { }

   login(){ this.oauthService.initImplicitFlow(); }
   logout(){ this.oauthService.logOut(); }

   get givenName() {
      let claims = this.oauthService.getIdentityClaims();
      if(!claims) return null;
      return claims.given_name;
   }
}
```

navmenu.component.ts hosted with ❤ by **GitHub** **view raw**

e o html, adicione-o ao final do menu de navegação.

```html
<li *ngIf="!givenName">
   <button class="btn btn-default" (click)="login()">
      Login
   </button>
</li>
<li *ngIf="givenName">
   <h1>Hello, {{givenName}}</h1>
</li>
<li *ngIf="givenName">
```

```
    <button class="btn btn-default" (click)="logout()">
       Logout
    </button>
</li>
```

navmenu.component.html hosted with ❤ by GitHub view raw

Por fim, vamos adicionar o token de autenticação aos cabeçalhos da chamada à API. Também aproveitei a oportunidade para mover a chamada para a API fora do construtor para um método OnInit, é uma péssima ideia tê-la no construtor de qualquer maneira.

```
import { Component, Inject, OnInit } from '@angular/core';
import { Http, Headers } from '@angular/http';
import { OAuthService } from 'angular-oauth2-oidc';

@Component({
    selector: 'fetchdata',
    templateUrl: './fetchdata.component.html'
})
export class FetchDataComponent implements OnInit {
    public forecasts: WeatherForecast[];

    constructor(
        private http: Http,
        @Inject('BASE_URL') private baseUrl: string,
        private oauthService: OAuthService) { }

    ngOnInit(): void {
        var headers = new Headers({
            "Authorization": "Bearer " + this.oauthService.getAccessToken()
        });

        this.http.get(this.baseUrl + 'api/SampleData/WeatherForecasts',
            { headers: headers })
            .subscribe(result => {
                this.forecasts = result.json() as WeatherForecast[];
            }, error => console.error(error));
    }
}

interface WeatherForecast {
    dateFormatted: string;
    temperatureC: number;
    temperatureF: number;
    summary: string;
}
```

fetchdata.component.ts hosted with ❤ by GitHub view raw

Como você pode ver, é apenas uma questão de obter o token de acesso usando a biblioteca oauth e adicioná-lo ao cabeçalho em uma propriedade Authorization.

Conclusão

Terminamos com este exemplo básico, passamos por todas as coisas importantes e mostramos como usar o Keycloak com um aplicativo Asp.Net Core.

Coloquei o projeto completo disponível no GitHub aqui: https://github.com/Gimly/SampleNetCoreAngularKeycloak

Também tenho um artigo de acompanhamento que o orientará sobre como adicionar autorização usando os grupos do Keycloak.
https://medium.com/@xavier.hahn/adding-authorization-to-asp-net-core-app-using-keycloak-c6c96ee0e655