Configurações externas

Programas modernos, especialmente programas executados na cloud, geralmente têm muitos componentes que são distribuídos por natureza. A difusão das definições de configuração nesses componentes pode levar a erros difíceis de solucionar durante o deployment de uma aplicação. Isto torna-se útil especialmente quando aplicado a uma arquitetura de microsserviços. O facto de termos configurações externas permite-nos alterar as mesmas sem ter que perder tempo a altera-las individualmente para cada serviço e testa-las de forma a ver se estão a funcionar corretamente, sem ter que voltar a dar rebuild e redeploy aos mesmos, torna mais fácil a centralização da gestão e a distribuição de dados de configuração hierárquica para diferentes ambientes e regiões geográficas e permite controlar a disponibilidade dos recursos em tempo real. Para além disto, ao armazenar todas as configurações dos nossos serviços num só lugar prevenimos erros na alteração de um setting transeversal a vários serviços, que pode levar ao funcionamento errado dos mesmos, criando assim um Single Source of Truth para as configurações e tornando a nossa aplicação mais robusta e escalável.

Tendo em conta a atual arquitetura dos serviços da Primavera esta abordagem de configurações externas é uma mais valia para a empresa. Um dos casos de uso que pode ser benificiado por esta abordagem é o de existir mais do que um servidor para alguns dos ambientes do processo de release de um serviço e ao alterar a configuração uma vez esta vai ser aplicada a todos os serviços nos diferentes servidores aos quais esta se aplique. Para contextualizar na prática, por exemplo, se um serviço no total tiver 5 servidores ao longo de todo o processo de release será necessário alterar as configurações 5 vezes nas webapps dos diferentes servidores das diferentes etapas de realese. Os benificios continuam a aumentar se multiplicarmos este processo pelos ~20 serviços que a Primavera possui atualmente. Tudo isto aliado à estrutura organizacional funcional das equipas, que leva a que o desenvolvimento e o deployment destes sejam feitos por equipas diferentes, gera o potencial de trazer bastantes benificios e agilizar o processo de atualização das configurações.

Para além disto se futuramente o deployment e hosting dos serviços passar das webapps para os containers esta abordagem continua a ser válida.

De forma à implementação ser bem sucedida é preciso criar uma divisão de namespaces para as configurações intuitiva e geral que seja fácil de aplicar aos serviços, a curto e longo prazo, e que espelhe a realidade da arquitetura dos microsserviços. O objetivo é que não seja necessário haver um refactoring futuro dos namespaces das configurações e que se possa manter sempre o mesmo standard.

Dentro das configuracões externas, e tendo em conta a realidade organizacional atual da Primavera, existem duas abordagens que se destacam mais para a implementação desta funcionalidade, sendo elas o Serviço App Configuration do Azure e o ConfigMaps dos Kubernetes.

O serviço App Configuration do Azure fornece um serviço para gerir centralmente as configurações de aplicações e as feature flags através de key-value pairs com mais alguns parâmetros configuráveis. É ideal para microsserviços baseados no Serviço Kubernetes do Azure, Azure Service Fabric ou outras aplicações em containers implantados numa ou mais geografias, aplicações serverless, que incluem o Azure Functions ou outras aplicações de computação sem estado controlados por eventos, e pipelines de continuous deployment.

Para mais informações - https://docs.microsoft.com/en-us/azure/azure-app-configuration/overview

Já o ConfigMaps permite desassociar artefatos de configuração do conteúdo da imagem para manter as aplicações em containers portáteis. É possível alcançar isto através da definição individual das configurações, da leitura desta de ficheiros de configuração ou da leitura de diretorias de ficheiros de configuração. Esta abordagem faz mais sentido quando já existe uma maior adoção e maturidade da tecnologia de containers por parte da organização porque a implementação desta torna-se mais simples e eficaz.

Para mais informações - https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/

Após uma comparação de ambas as abordagens, e tendo em conta a realidade atual da Primavera tal como o facto de usar a stack tecnológica da Microsoft, usar vários serviços do azure e de ainda não ter implementado a tecnologia de containers, a que se enquadra melhor é a do serviço App Configuration do Azure.

Abaixo será descrito o processo de implementação de ambas as abordagens.

Serviço App Configuration do Azure

No portal do Azure

A primeira tarefa é ir ao portal do azure, procurar o serviço "App Configuration" e criar uma nova App Configuration store ou utilizar uma já existente. Caso seja criada uma nova store devemos adicionar algumas configurações. É possível fazer isto manualmente ou através do import das configurações de outras App Configurations stores, App services ou ficheiros de configuração (p.e. appsettings.json). Consequentemente, é também possível fazer o export de configurações de uma App Configurations stores para outras App Configurations stores, App services ou ficheiros de configuração.

Na aplicação

Pelo facto de estarmos a aceder a um serviço do Azure que contém informação importante e que queremos manter secreta precisamos de usar um secret manager para guardar a nossa connectionString da instância da App Configuration store a ser utilizada. A ferramenta do secret manager armazena dados confidenciais para o trabalho de desenvolvimento fora da árvore dos nossos projetos. Esta abordagem ajuda a impedir a partilha acidental de secrets da aplicação no código-fonte.

A forma de implementação deste depende do ambiente em que estamos inseridos. Neste exemplo será especificado como adicionar um secret manager para o ***ambienete de desenvolvimento local***.

Primeiramente temos que adicionar o elemento UserSecretsId aos ficheiro .csproj e atribuir-lhe um GUID.

```
<Project Sdk="Microsoft.NET.Sdk.Web">

    <PropertyGroup>
        <TargetFramework>netcoreapp2.1</TargetFramework>
        <UserSecretsId>79a3edd0-2092-40a2-a04d-dcb46d5ca9ed</UserSecretsId>
    </PropertyGroup>

    <ItemGroup>
        <PackageReference Include="Microsoft.AspNetCore.App" />
        <PackageReference Include="Microsoft.AspNetCore.Razor.Design"
Version="2.1.2" PrivateAssets="All" />
```

```
    </ItemGroup>

</Project>
```

Para podermos usar o serviço App Configuration do Azure no nosso projeto, precisamos de instalar o NuGet package `Microsoft.Azure.AppConfiguration.AspNetCore`. Posteriormente deverá ser corrido o comando `dotnet restore` para restaurar os packages do projeto.

Após isto devemos adicionar um secret ao secret manager e dar-lhe um nome. Neste caso o nome será ConnectionStrings:AppConfig.

Nota: Este comando tem que ser executado na mesma diretoria que o ficheiro .csproj do projeto a desenvolver. É também preciso ter atenção à shell que estamos a utilizar pois algumas fazem o truncate da connectionString, a menos que ela esteja entre aspas.

```
dotnet user-secrets set ConnectionStrings:AppConfig <your_connection_string>
```

Já no código da aplicação, e após este setup inicial, devemos atualizar o método `CreateWebHostBuilder` no Program.cs para usar o App Configuration.

Nota: Dependendo do sitio onde inserimos o método para adicionar o AppConfiguration podemos, ou não, dar overwrite às configurações dos ficheiros locais. Isto acontece caso este método esteja depois da adição dos ficheiros de configuraçãoes local (p.e. config.AddJsonFile($"./GeneratedCode/appsettings.gen.json", optional: false, reloadOnChange: true)) e caso os settings de ambos os lados tenham o mesmo nome.

Dentro do método do AppConfiguration, para além de inserirmos a connectionString para conectar à nossa App Configuration store, que neste caso foi criado através do secret manager, podemos também configurar outros métodos, tais como o `ConfigureRefresh` que serve para especificar as configurações usadas para atualizar os dados de configuração no armazenamento de App App quando uma operação de atualização é acionada. Para realmente acionar uma operação de atualização, é necessário configurar um middleware de atualização para que o aplicativo atualize os dados de configuração quando ocorrer qualquer alteração.

registar configurações na cache e definir alguns parâmetros, como por exemplo o tempo de renovação do valor, podemos também selecionar apenas algumas das configurações existentes, entre outras. É neste ponto que se vai verificar se a implementação dos namespaces para as configurações foi bem sucedido ou não

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .ConfigureAppConfiguration((hostingContext, config) =>
        {
            var settings = config.Build();
                    config.AddAzureAppConfiguration(options =>
                    {
                        options.Connect(settings["ConnectionStrings:AppConfig"])
                               .ConfigureRefresh(refresh =>
                               {

refresh.Register("TestApp:Settings:testeExternalConfig");
                                    refresh.Register("TestApp:Settings",
refreshAll: true);
```

```
//refresh.SetCacheExpiration(TimeSpan.FromMinutes(5));
                                })
                                .Select(KeyFilter.Any, LabelFilter.Null)
                                .Select(KeyFilter.Any, "prod");
                    });
                });
        })
        .UseStartup<Startup>();
```

```
namespace Primavera.Lithium.Nitrogen.WebApi.CustomCode.Models
{
    public class Settings
    {
        /// <summary>
        /// Model that contains the scopes for a product module
        /// </summary>
        public string testeExternalConfig { get; set; }
    }
}
```

Antes do MVC

```
/// <summary>
/// Called when configuring services to configure additional services.
/// </summary>
/// <param name="services">The service collection.</param>
/// <param name="configuration">The service collection.</param>
/// <remarks>
/// The method is called from <see cref="ConfigureServices(IServiceCollection)"/>.
/// </remarks>
public virtual void AddAppConfiguration(IServiceCollection services)
{
    if (Configuration.GetValue<bool>("UseAzureAppConfiguration"))
            services.Configure<Settings>
(Configuration.GetSection("TestApp:Settings"));

    Console.WriteLine(Configuration["IdentityProviderConfiguration:Uri"]);

    Configuration["IdentityProviderConfiguration:Uri"] =
Configuration.GetValue<string>("IdentityProviderConfiguration:Uri");

    Console.WriteLine(Configuration["IdentityProviderConfiguration:Uri"]);
}
```

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
```

```
    app.UseAzureAppConfiguration();

    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    app.UseMvc();
}
```

```
public class HomeController : Controller
{
    private readonly Settings _settings;
    public HomeController(IOptionsSnapshot<Settings> settings)
    {
        _settings = settings.Value;
        Console.WriteLine(_settings.testeExternalConfig);
    }
}
```

Para mais informações consultar:

- https://docs.microsoft.com/en-us/azure/azure-app-configuration/overview
- https://docs.microsoft.com/en-us/azure/azure-app-configuration/quickstart-aspnet-core-app?tabs=core2x
- https://docs.microsoft.com/en-us/azure/azure-app-configuration/enable-dynamic-configuration-aspnet-core?tabs=core2x

ConfigMap dos Kubernetes

```
config.AddJsonFile("config/appsettingsteste.json", optional: true, reloadOnChange: true);
```

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: demo-config
data:
  appsettingsteste.json: |-
    {
      "Logging": {
        "LogLevel": {
          "Default": "Error",
          "System": "Error",
          "Microsoft": "Error"
```

```
        }
      }
    }
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ntr-deployment
  labels:
    app: ntr
spec:
  replicas: 3
  selector:
    matchLabels:
      app: ntr
  template:
    metadata:
      labels:
        app: ntr
    spec:
      containers:
      - name: ntr
        image: primaverabss.azurecr.io/dev/ntr:v4
        ports:
        - containerPort: 80
        volumeMounts:
        - name: config-volume
          mountPath: NTR/config
      volumes:
      - name: config-volume
        configMap:
          name: demo-config
      imagePullSecrets:
      - name: primaverabss
```

Infelizmente, o refresh interno nas alterações no provedor de arquivos principais do .NET não funciona com ficheiros symlink. O mapa de configuração não aciona o refresh da configuração como seria de esperar. Isto parece acontecer porque a descoberta de alterações do .NET core depende da data da última modificação do ficheiro. Como o ficheiro que estamos a monitorizar não foi alterado (a referência do symlink mudou), nenhuma alteração foi detetada.

Até hoje este problema ainda não foi resolvido, porém, por agora, podemos tirar proveito do sistema de configuração extensível do .NET Core e implementar um provedor de configuração baseado em ficheiros que detecta alterações com base no conteúdo do arquivo.

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .ConfigureAppConfiguration(c =>
```

```
        {
            c.AddJsonFile(ConfigMapFileProvider.FromRelativePath("config"),
                "appsettings.json",
                optional: true,
                reloadOnChange: true);
        })
        .UseStartup<Startup>();
```