Hydrogen Sessions

#5 – Serviços

Serviços e injeção de dependências

Entenda-se por serviço qualquer tipo (ou instância) que é registado no mecanismo de injeção de dependências nativo do ASP.NET Core (service collection, service container).

Esse mecanismo está extensamente descrito na documentação:

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection

Um serviço pode ser qualquer coisa: um interface (o normal), uma classe de configuração, mesmo uma instância de um objeto.

A construção de componentes como serviços tem como grandes vantagens:

- A aplicação do padrão de desenho DI e do princípio IOC: https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/architectural-principles#dependency-inversion
- Simplificação da instanciação desses serviços (IServiceProvider.GetService<XXXX>()).
- Simplificação dos testes unitários (ou até permitir cenários muito difíceis de testar sem DI).

Serviços no Hydrogen

Praticamente todos os componentes do Hydrogen estão construídos como serviços, para serem registados no container.

Alguns componentes (exemplo: TokenClient), pela sua natureza, fogem a essa regra (por exemplo, porque podem ser invocados a partir de "ambientes" onde não exista um container configurado). Noutros casos, ainda que o componente não seja um serviço, existe um serviço que permite obter instâncias do componente (ver, por exemplo, SmtpClientServiceCollectionExtensions).

Portanto, por defeito, qualquer nova funcionalidade no Hydrogen deve ser construída como um serviço.

Consideremos, como exemplo, um dos serviços mais utilizados, o ITableStorageService:

```
/// <summary>
/// Defines the interface of the table storage service.
/// The table storage service is a structured NoSQL data store in the cloud.
/// The table storage is a key/attribute store with a schema-less design.
/// New instances of the service should be resolved via dependency injection to realize
/// the concrete implementation.
/// </summary>
/// <remarks>
/// ATTENTION:
/// Beware that the service will communicate via a REST API to the actual storage service used.
/// Each instance of the service returned will maintain its own instance of the API client,
/// even if the configuration used is exactly the same.
/// Correctly managing the instances of the services in use will impact the application
/// performance.
/// </remarks>
70 references | Hugo Lourenço | 1 author, 1 change | 1 incoming change
public partial interface ITableStorageService
{
    #region Methods

    #region GetTable

    /// <summary>
    /// Gets a reference to the specified table.
    /// </summary>
    /// <param name="tableName">The name of the table.</param>
    /// <returns>The <see cref="ITableReference"/> that represents a reference to the table.</returns>
    25 references | 8/14 passing | Hugo Lourenço | 1 author, 1 change | 1 incoming change
    ITableReference GetTable(string tableName);

    #endregion

    #endregion
}
```

IServiceCollection extensions

Para permitir ao "utilizador" registar esse serviço na aplicação, criam-se métodos de extensão sobre IServiceCollection para registar a implementação concreta do serviço, tipicamente associando-lhe opções de configuração que permitam modificar determinados comportamentos e/ou fornecer inputs necessários.

```
/// <summary>
/// Adds a service that allows creating instances of <see cref="ITableStorageService"/> using
/// configuration defined by <see cref="AzureTableStorageOptions"/>.
/// This service will use Azure Table Storage.
/// </summary>
/// <param name="services">The service collection.</param>
/// <returns>
/// The <see cref="IServiceCollection"/> instance.
/// </returns>
/// <remarks>
/// <see cref="ITableStorageService"/> is registered as a transient service.
/// </remarks>
12 references | 5/5 passing | Hugo Ribeiro, 24 days ago | 2 authors, 2 changes | 1 incoming change
public static IServiceCollection AddAzureTableStorage(this IServiceCollection services)
{
    // Validation

    SmartGuard.NotNull(() => services, services);

    // Logging

    services
        .AddLogging();

    // Configuration

    services
        .AddOptionsSnapshot<AzureTableStorageOptions>();

    // Add transient service

    services
        .TryAddTransient<ITableStorageService, AzureTableStorageService>();

    // Result

    return services;
}
```

Este é o método principal, aquele que regista o serviço e pressupõe a existência em runtime de determinadas opções de configuração (AzureTableStorageOptions).

Note-se o seguinte:

- O método regista também dependências de que a implementação necessite (AddLogging()). Neste caso, dada a forma com o serviço de Logging funciona, não há qualquer problema em registá-lo múltiplas vezes.
- Regista ainda a classe de configuração de que depende (AzureTableStorageOptions).
- Finalmente, regista a implementação concreta (AzureTableStorageService).
- O serviço é registado como transiente com o método TryAddTransient (o que permite que o utilizador registe previamente outra implementação que não será substituída, o que é particularmente útil para testes unitários, por exemplo).

IServiceCollection extensions

Os métodos de extensão dependerão da natureza do serviço e também dos parâmetros necessários. No entanto, devem sempre existir pelo menos 2: o anterior e um outro que aceite um delegate para modificar a configuração em runtime (em código).

```
/// <summary>
/// Adds a service that allows creating instances of <see cref="ITableStorageService"/> using the
/// specified configuration options.
/// This service will use Azure Table Storage.
/// </summary>
/// <param name="services">The service collection.</param>
/// <param name="configureDelegate">The configuration delegate.</param>
/// <returns>
/// The <see cref="IServiceCollection"/> instance.
/// </returns>
/// <remarks>
/// <see cref="ITableStorageService"/> is registered as a transient service.
/// </remarks>
23 references | 7/7 passing | Hugo Ribeiro, 24 days ago | 2 authors, 2 changes | 1 incoming change
public static IServiceCollection AddAzureTableStorage(
    this IServiceCollection services,
    Action<AzureTableStorageOptions> configureDelegate)
{
    // Validation

    SmartGuard.NotNull(() => services, services);
    SmartGuard.NotNull(() => configureDelegate, configureDelegate);

    // Configure (after static configuration is loaded)

    services.PostConfigure(configureDelegate);

    // Add

    return services.AddAzureTableStorage();
}
```

Note-se a forma como o delegate é "configurado" usando o PostConfigure, para ser executado depois da configuração ser obtida usando o snashot configurado no método anterior.

Desta forma, este método pode simplesmente chamar o anterior para registar o serviço.

Serviços nas aplicações

Importa recapitular como é que os serviços são configurados e usados nas aplicações (nos micro serviços).

Todos os serviços devem ser registados na etapa “ConfigureServices” do startup da aplicação.

No caso dos micro serviços, há dois momentos principais (mas não só) em que são registados serviços, particularmente aqueles disponibilizados pelo Hydrogen: AddDependencies() e AddAdditionalServices().

```
/// <summary>
/// Called to add dependencies to the service collection.
/// </summary>
/// <param name="services">The service collection.</param>
/// <param name="hostConfiguration">The host configuration.</param>
/// <remarks>
/// The method is called from <see cref="ConfigureServices(IServiceCollection)"/>.
/// </remarks>
1 reference | Hugo Lourenço | 1 author, 1 change
protected virtual void AddDependencies(IServiceCollection services, HostConfiguration hostConfiguration)
{
    // Add dependencies

    // Distributed cache

    this.AddDistributedCache(services, hostConfiguration);

    // Table storage

    this.AddAzureTableStorage(services, hostConfiguration);

    // Search

    this.AddAzureSearch(services, hostConfiguration);

    // Pipelines

    this.AddPipelines(services, hostConfiguration);
}
```

```
/// <summary>
/// Called when configuring services to configure the distributed cache service.
/// </summary>
/// <param name="services">The service collection.</param>
/// <param name="hostConfiguration">The host configuration.</param>
1 reference | Hugo Ribeiro, 14 days ago | 2 authors, 2 changes
protected virtual void AddDistributedCache(IServiceCollection services, HostConfiguration hostConfiguration)
{
    // Validation

    SmartGuard.NotNull(() => services, services);
    SmartGuard.NotNull(() => hostConfiguration, hostConfiguration);

    // REDIS cache (resilient)

    services
        .AddRedisCache();

    services
        .AddResilientDistributedCache();
}
```

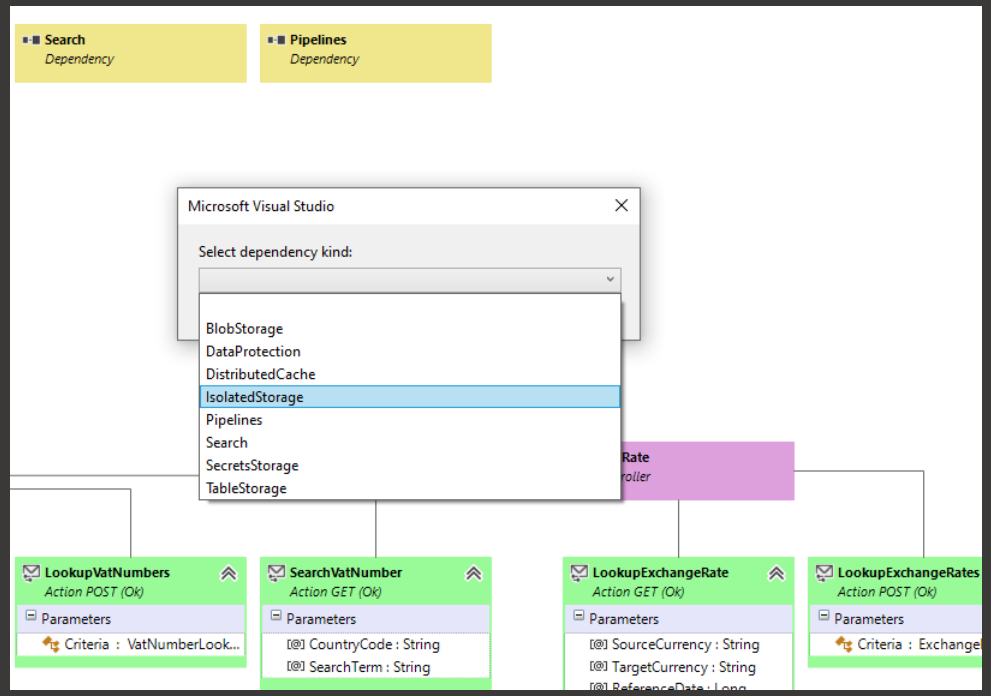
Serviços nas aplicações

O segundo é o método que deve ser “overriden” na implementação do micro serviço para registar os serviços “custom”:

```
/// <summary>
/// Called to add additional (custom) services to the service collection.
/// </summary>
/// <param name="services">The service collection.</param>
/// <param name="hostConfiguration">The host configuration.</param>
/// <remarks>
/// The method is called from <see cref="ConfigureServices(IServiceCollection)"/>.
/// </remarks>
4 references | Hugo Ribeiro, 14 days ago | 2 authors, 2 changes
protected virtual void AddAdditionalServices(IServiceCollection services, HostConfiguration hostConfiguration)
{
    // Validation

    SmartGuard.NotNull(() => services, services);
    SmartGuard.NotNull(() => hostConfiguration, hostConfiguration);

    // Add HttpClient

    this.AddHttpClient(services, hostConfiguration);

    // Add endpoint analyzer

    services
        .AddEndpointAnalyzer();

    // Add configuration analyzer

    services
        .AddConfigurationAnalyzer();
}
```

Serviços nas aplicações

Regra geral também, alguns dos serviços do Hydrogen são tratados pelo SDK como “dependências” que podem ser adicionadas ao modelo do microserviço para que sejam adicionadas automaticamente ao código gerado.



Isso acontecerá em função da utilidade do serviço, por um lado, e da possibilidade do developer decidir ou não se determinado serviço deve ser registado ou não no seu micro serviço (alguns são mesmo obrigatórios).

Serviços nas aplicações

A resolução do serviço pode ser depois realizada em runtime em qualquer classe da aplicação, seja por injeção no construtor, seja através do IServiceProvider. Exemplos:

```
public AzureBlobStorageService(AzureBlobStorageOptions options, ILogger<AzureBlobStorageService> logger)
{
    // Validation

    SmartGuard.NotNull(() => options, options);
    SmartGuard.NotNull(() => logger, logger);

    // Set properties

    this.Options = options;
    this.Logger = logger;

    // Initialize client

    this.InitializeClient();
}
```

```
2 references | Hugo Lourenço | 1 author, 1 change
private IDistributedCache Cache
{
    get
    {
        return this.ServiceProvider.GetRequiredService<IDistributedCache>();
    }
}
```

```
4 references | Hugo Lourenço | 1 author, 1 change
private IVatNumberManager Manager
{
    get
    {
        return this.HttpContext.RequestServices.GetRequiredService<IVatNumberManager>();
    }
}
```

Transient, singleton, scoped

Em princípio, o tempo de vida de um serviço depende da forma como é registado:

- Um transient vive durante o tempo em que é utilizado (ou seja, sempre que é resolvido, obtém-se uma instância nova).
- Um singleton vive durante o tempo de vida da aplicação (ou seja, é sempre resolvida a mesma instância).
- Um scoped vive durante o tempo de vida do scope (se o scope for a request HTTP, durante o processamento dessa request, a instância será sempre a mesma).

Há situações especiais em que isto é, na verdade, mais complexo e pode levar a comportamentos inesperados (que já aconteceram). As recomendações usualmente aceites são:

- Devem preferir-se serviços transientes, porque o seu comportamento é mais previsível e são mais fáceis de implementar (sem cair nas tais situações especiais).
- Os serviços singleton devem ser utilizados para manter comportamento/estado muito estático. Os benefícios de performance de um singleton em relação a um transiente são muitas vezes apenas aparentes (poupa-se na resolução do serviço à custa de mais memória consumida e cenários de GC pendurado).
- Os serviços scoped devem ser evitados exceto em situações muito específicas (exemplo: contextos para a base de dados) porque são mais difíceis de implementar e de utilizar.

Configuração e lazy loading

A resolução dos serviços também tem a ver com a resolução das opções de configuração de que estes dependem.

Um aspeto muito importante nas aplicações .NET Core que executam no Azure é a possibilidade de modificar os app settings no portal do Azure e essas alterações terem efeito imediato na aplicação (sem restart).

Essa é mais uma razão para evitar serviços singleton, se estes mantiverem em memória as opções de configuração.

Este aspeto também tem a ver com a tentação de fazer lazy loading dos serviços com construções do género da seguinte:

```
private IMyService myService;
public IMyService MyService
{
  get
  {
    if (this.myService == null)
    {
      this.myService = IServiceProvider.GetRequiredService<IMyService>();
    }
    return this.myService
  }
}
```

Notas:

Se o serviço onde esta resolução estiver for transiente, o efeito de lazy loading é nulo, porque a instância do serviço pai é sempre nova (logo o field é null).

Se for singleton, então pode ser ainda pior, porque se IMyService for uma classe de configuração, então ela só vai ser carregada uma vez (e qualquer alteração em runtime no portal não teria efeito).

Boas Práticas

#1 - Nome do serviço

O serviço deve ter um nome significativo e simples. Regra geral, deve ter um sufixo “Service” (ex.: IBlogStorageService).

#2 - Classe com a implementação concreta

A classe deve ter um nome que indique algo sobre a implementação e deve ser baseado no nome do serviço (ex.: AzureBlobStorageService).

A classe deve ser sempre interna, para impedir a sua instanciação direta (a visibilidade para os testes unitários deve ser configurada com o atributo [InternalsVisibleTo]).

#3 - Classe com extensões a IServiceCollection

Esta classe terá normalmente o nome da implementação interna (sem “Service”) mais o sufixo “ServiceCollectionExtensions” (ex.: AzureBlobStorageServiceCollectionExtensions), o mesmo namespace e deve ser separada numa pasta com o nome “DependencyInjection”

Boas Práticas

#4 – Métodos de extensão para registo do serviço

Os métodos devem ter o nome “Add” seguidos do nome da implementação interna (sem o sufixo “Service”) (ex.: AddAzureBlobStorage()).

O primeiro método não deve ter parâmetros (apenas this IServiceCollection) e registar todas as dependências, a classe de configuração e o serviço, por esta ordem.

Deve ser adicionado pelo menos mais um método que aceite um Action<X> (X é a classe de configuração) para permitir a configuração do serviço em código.

#5 – Classe de opções de configuração

A classe de configuração deve ter o nome da implementação interna (sem o sufixo “Service”) mais o sufixo “Options” (ex.: AzureBlobStorageOptions). Deve ser também colocada na pasta “DependencyInjection”.

#6 – Lifetime

Regra geral, os serviços devem ser transientes.

Sempre que for viável, deve ser possível substituir a implementação registada, em código, tanto antes como depois do registo. Por exemplo:

```
services.Configure<AzureTableStorageOptions>(context.Configuration.GetSectic
services.AddTransient<ITableStorageService, MockTableStorageService>();
services.AddAzureTableStorage();
```

```
services.Configure<AzureTableStorageOptions>(context.Configuration.GetSec
services.AddAzureTableStorage();
services.AddTransient<ITableStorageService, MockTableStorageService>();
```

Boas Práticas

#7 - Documentação

É importante que a documentação dos métodos de extensão seja bastante explicativa.

Deve colocar-se nas remarks, informação sobre como o serviço é registado (ver exemplo anterior do ITableStorageService).

Decorators

Os serviços definidos por terceiros usam, muitas vezes, outros serviços (as suas dependências). Em determinadas situações, pode ser necessário substituir essas dependências. Noutros casos, pode ser necessário substituir mesmo a implementação concreta registada do serviço pai.

O Hydrogen inclui uma funcionalidade – os decorators – que permite exatamente isso (ver em Core\DependencyInjection).

É utilizada, por exemplo, para registar a cache distribuída resiliente. Embora o serviço continue a ser IDistributedCache, a instância resolvida será um decorator (ResilientDistributedCache) da implementação concreta registada.

```
11 references | 0/5 passing | Hugo Lourenço | 1 author, 1 change | 1 incoming change
public static IServiceCollection AddResilientDistributedCache(this IServiceCollection services)
{
    // Validation

    SmartGuard.NotNull(() => services, services);

    // Services

    services
        .AddLogging();

    // Configuration

    services
        .AddOptionsSnapshot<ResilientCacheOptions>();

    // Ensure the original service registration

    services
        .AddDistributedMemoryCache();

    // Decorate original service

    services
        .AddTransientDecorator<IDistributedCache, ResilientDistributedCache>();

    // Result

    return services;
}
```

```
/// <inheritdoc />
1 reference | Hugo Ribeiro, 43 days ago | 2 authors, 3 changes | 1 incoming change
public virtual Task SetAsync(string key, byte[] value, DistributedCacheEntryOptions options, CancellationToken token = default)
{
    // Handle exceptions

    try
    {
        // Execute decorated service

        return this.RetryPolicy
            .ExecuteAsync(
                () => this.Decorator.Instance.SetAsync(key, value, options, token));
    }
    catch (Exception ex)
    {
        // Logging
        // Do not log exception because it will end up in the exceptions of AI (not only in the trace)

        this.Logger.LogWarning($"{nameof(IDistributedCache)} instance raised exception. Exception: {ex.GetType().FullName} - {ex.Message}.");

        // Throw

        throw;
    }
}
```