Hydrogen Sessions

#11 – Storage

Serviços de storage

Em Primavera.Hydrogen.Storage.Abstractions estão disponíveis 3 serviços de storage diferentes (a prazo serão mais):

- ITableStorageService – table storage (structured NoSQL sobre tabelas indexedas por 2 colunas, PartitionKey e RowKey).
- IBlobStorageService – armazenamento de ficheiros (imagens, texto, binários) acessíveis, opcionalmente, num endereço público.
- IIsolatedStorageService – armazenamento local (em disco) com isolamento e proteção.

Estes serviços – o table storage em particular – são muito utilizados nos micro serviços e nem sempre têm sido corretamente utilizados. Por isso vale a pena explicar um pouco melhor o seu funcionamento e características particulares.

Registos dos serviços (implementações concretas)

Estes serviços existem para adaptar as suas implementações concretas. A ideia fundamental é ter N implementações concretas (Azure, AWS, etc.) e poder, de alguma forma, configurar a implementação usada em cada aplicação (inclusive em runtime), mantendo a API do serviço, logo também todo o código custom que depender do serviço.

Neste momento existe apenas, para cada serviço, uma única implementação para o respetivo serviço no Azure. Isso pode mudar logo que se justifique.

Por outro lado, como se pode ver a seguir, o mecanismo de registo ainda não permite alterar a implementação concreta em runtime ou sequer por configuração. Isso deverá mudar quando o ponto anterior tiver que ser resolvido.

As implementações concretas são:

- Primavera.Hydrogen.Storage.Azure.Tables.AzureTableStorageService
- Primavera.Hydrogen.Storage.Azure.Blobs.AzureBlobStorageService
- Primavera.Hydrogen.Storage.Files.Isolated.FilesIsolatedStorageService

Registos dos serviços (implementações concretas)

O registo destes serviços é realizado como todos os outros serviços:

```
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;
}
```

```
16 references | 9/9 passing | Hugo Lourenço | 1 author, 1 change | 1 incoming change
public static IServiceCollection AddAzureBlobStorage(this IServiceCollection services)
{
    // Validation

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

    // Logging

    services
        .AddLogging();

    // Configuration

    services
        .AddOptionsSnapshot<AzureBlobStorageOptions>();

    // Add service

    services
        .TryAddTransient<IBlobStorageService, AzureBlobStorageService>();

    // Result

    return services;
}
```

```
public static IServiceCollection AddFilesIsolatedStorage(this IServiceCollection services)
{
    // Validation

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

    // Logging

    services
        .AddLogging();

    // Add service

    services
        .TryAddTransient<IIsolatedStorageService, FilesIsolatedStorageService>();

    // Result

    return services;
}
```

IIsolatedStorageService

Este serviço permite instanciar uma “store” para armazenar ficheiros que ficam no file system numa “área protegida”.

https://docs.microsoft.com/en-us/dotnet/standard/io/isolated-storage

```
35 references | Hugo Ribeiro, 29 days ago | 1 author, 3 changes
public partial interface IIsolatedStorageService
{
    #region Properties

    /// <summary>
    /// Gets the machine-scoped isolated store.
    /// </summary>
    8 references | 3/3 passing | Hugo Ribeiro, 29 days ago | 1 author, 1 change
    IIsolatedStorageScope Machine
    {
        get;
    }

    /// <summary>
    /// Gets the user-scoped isolated store.
    /// </summary>
    8 references | 3/3 passing | Hugo Ribeiro, 29 days ago | 1 author, 1 change
    IIsolatedStorageScope User
    {
        get;
    }

    #endregion
}
```

```
public partial interface IIsolatedStore : IDisposable
{
    #region Methods

    /// <summary>
    /// Creates a file in the isolated store.
    /// </summary>
    /// <param name="path">The path and file name.</param>
    /// <param name="contents">The file contents.</param>
    79 references | 36/36 passing | Hugo Ribeiro, 136 days ago | 1 author, 1 change
    void CreateFile(string path, byte[] contents);

    /// <summary>
    /// Deletes the specified file from the isolated store.
    /// </summary>
    /// <param name="path">The path and file name.</param>
    43 references | 18/18 passing | Hugo Ribeiro, 136 days ago | 1 author, 1 change
    void DeleteFile(string path);

    /// <summary>
    /// Determines whether the specified path refers to an existing file in the isolated store.
    /// </summary>
    /// <param name="path">The path and file name.</param>
    /// <returns>
    /// A value indicating whether the specified path refers to an existing file in the isolated store.
    /// </returns>
    62 references | 30/30 passing | Hugo Ribeiro, 136 days ago | 1 author, 1 change
    bool FileExists(string path);

    /// <summary>
    /// Tries to read the specified file from the isolated store.
    /// </summary>
    /// <param name="path">The path and file name.</param>
    /// <param name="contents">The file contents.</param>
    /// <returns>
    /// A value indicating whether the file could be read. If true, then <paramref name="contents"/> contains the file contents.
    /// </returns>
    31 references | 18/18 passing | Hugo Ribeiro, 136 days ago | 1 author, 1 change
    bool TryReadFile(string path, out byte[] contents);

    #endregion
}
```

IIsolatedStorageService

Eis um exemplo de utilização deste serviço:

```
1 reference
private static bool UseIsolatedStorage()
{
    IIsolatedStorageService service = ServiceProvider.GetRequiredService<IIsolatedStorageService>();

    IIsolatedStore store = service.User.GetStore(StoreScope.Application);

    try
    {
        if (!store.FileExists("TheFile.txt"))
        {
            byte[] content = "File contents".GetBytes();

            store.CreateFile("TheFile.txt", content);
        }

        return true;
    }
    catch (IsolatedStorageException ex)
    {
        Console.WriteLine("Something went wrong: {0}", ex);
        return false;
    }
}
```

O serviço tem um conceito de store que corresponde a um "espaço de armazenamento" isolado. Pode ser de 3 tipos (scopes):

- Application
- Assembly
- Domain

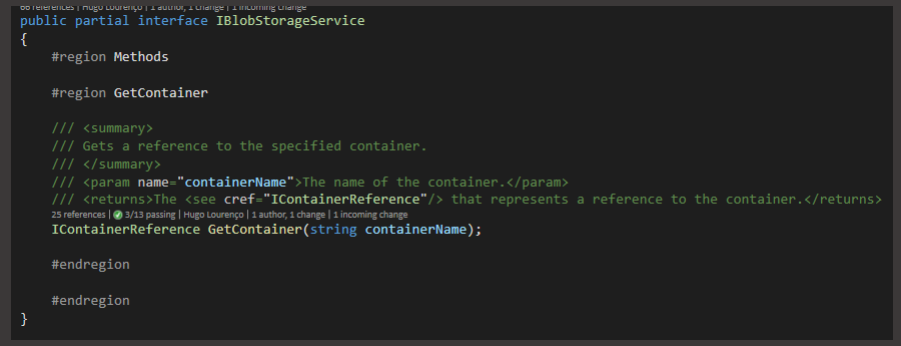
Qualquer erro será identificado por uma exceção do tipo IsolatedStorageException.

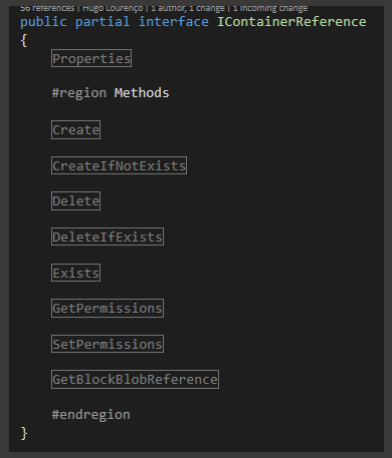
IBlobStorageService

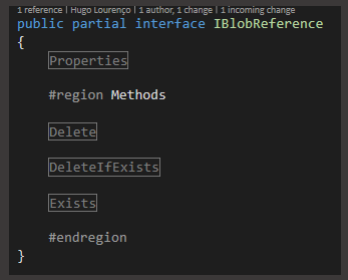
Este serviço permite armazenar ficheiros raw (blobs) na cloud, organizados em containers. Os blobs podem ficar acessíveis, opcionalmente, num endereço público (por exemplo, para referenciar a partir de HTML).

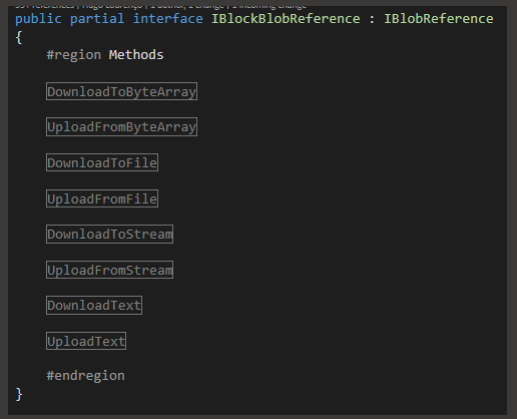
https://azure.microsoft.com/en-us/services/storage/blobs/

NOTA: Embora existam vários tipos de blobs. Atualmente o serviço suporta apenas block blobs.









IBlobStorageService

Eis um exemplo da utilização deste serviço:

```
private static async Task<bool> UseBlobStorageAsync()
{
    IBlobStorageService service = ServiceProvider.GetRequiredService<IBlobStorageService>();

    try
    {
        IContainerReference container = service.GetContainer("testing-1");

        await container
            .CreateIfNotExistsAsync(ContainerPublicAccessType.BlobsOnly)
            .ConfigureAwait(false);

        IBlockBlobReference blob = container.GetBlockBlob("blob-1");

        await blob
            .UploadTextAsync("File contents")
            .ConfigureAwait(false);

        string result = await blob.DownloadTextAsync()
            .ConfigureAwait(false);

        return true;
    }
    catch (AzureBlobStorageException ex)
    {
        if (ex.ErrorCode == BlobStorageError.InvalidContainerName)
        {
            Console.WriteLine("The container name is invalid.");
            return false;
        }

        Console.WriteLine("Something went wrong: {0}", ex);
        return false;
    }
}
```

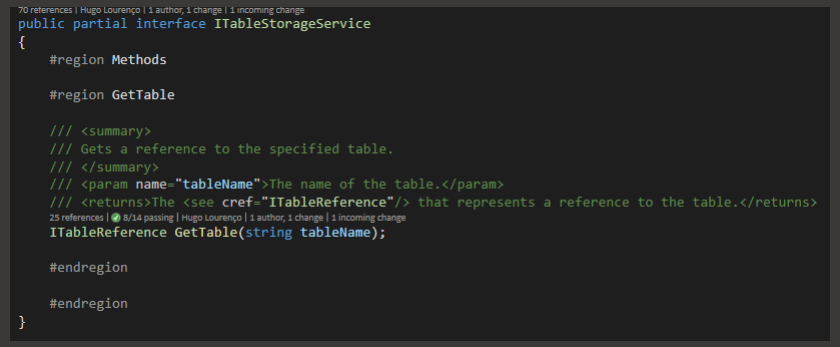
Não é visível neste exemplo, mas o blob fica disponível num endereço público (por causa da permissão dada na criação do container). Esse endereço está em blob.Uri.

Qualquer erro será identificado por uma exceção do tipo AzureBlobStorageException. Note-se a propriedade ErrorCode.

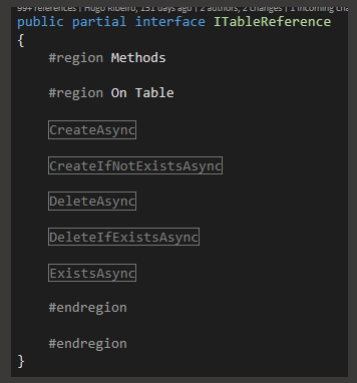
ITableStorageService

O serviço de table storage permite armazenar dados “semi-estruturados” em tabelas indexadas por 2 colunas (PartitionKey e RowKey).

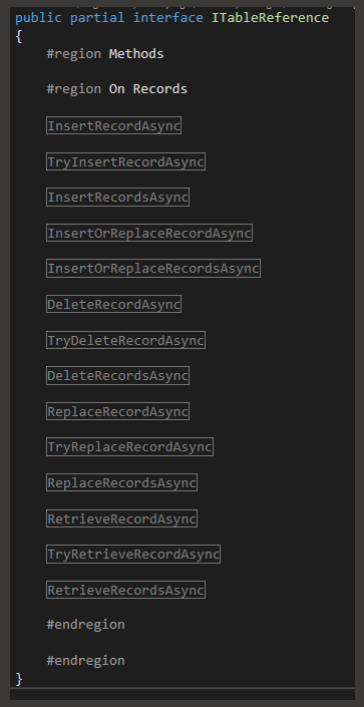
https://docs.microsoft.com/en-us/azure/cosmos-db/tutorial-develop-table-dotnet



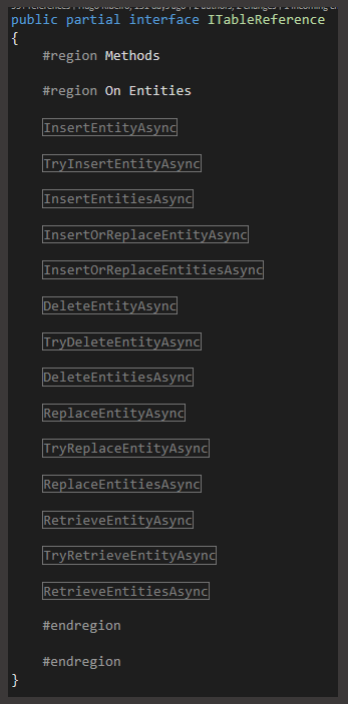
```
70 references | Hugo Lourenço | 1 author, 1 change | 1 incoming change
public partial interface ITableStorageService
{
    #region Methods

    #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
}
```



```
public partial interface ITableReference
{
    #region Methods

    #region On Table

    CreateAsync

    CreateIfNotExistsAsync

    DeleteAsync

    DeleteIfExistsAsync

    ExistsAsync

    #endregion

    #endregion
}
```



```
public partial interface ITableReference
{
    #region Methods

    #region On Records

    InsertRecordAsync

    TryInsertRecordAsync

    InsertRecordsAsync

    InsertOrReplaceRecordAsync

    InsertOrReplaceRecordsAsync

    DeleteRecordAsync

    TryDeleteRecordAsync

    DeleteRecordsAsync

    ReplaceRecordAsync

    TryReplaceRecordAsync

    ReplaceRecordsAsync

    RetrieveRecordAsync

    TryRetrieveRecordAsync

    RetrieveRecordsAsync

    #endregion

    #endregion
}
```



```
public partial interface ITableReference
{
    #region Methods

    #region On Entities

    InsertEntityAsync

    TryInsertEntityAsync

    InsertEntitiesAsync

    InsertOrReplaceEntityAsync

    InsertOrReplaceEntitiesAsync

    DeleteEntityAsync

    TryDeleteEntityAsync

    DeleteEntitiesAsync

    ReplaceEntityAsync

    TryReplaceEntityAsync

    ReplaceEntitiesAsync

    RetrieveEntityAsync

    TryRetrieveEntityAsync

    RetrieveEntitiesAsync

    #endregion

    #endregion
}
```

ITableStorageService

O serviço permite operar sobre os registos de duas formas:

- Através de um TableRecord (não estruturado)
- Através de uma TableEntity (estruturado).

Ambos os modos têm as suas vantagens e desvantagens.

Eis um exemplo do primeiro caso:

```
1 reference
private static async Task<bool> UseTableStorageWithRecordsAsync()
{
    ITableStorageService service = ServiceProvider.GetRequiredService<ITableStorageService>();

    try
    {
        ITableReference table = service.GetTable("sampletable1");

        await table
            .CreateIfNotExistsAsync()
            .ConfigureAwait(false);

        TableRecord record = new TableRecord("key1", "key2");
        record.Properties.Add("Value1", 10);
        record.Properties.Add("Value2", false);
        record.Properties.Add("Value3", "text");

        await table
            .InsertOrReplaceRecordAsync(record)
            .ConfigureAwait(false);

        IList<TableRecord> records = await table
            .RetrieveRecordsAsync("key1")
            .ConfigureAwait(false);

        return true;
    }
    catch (AzureTableStorageException ex)
    {
        if (ex.ErrorCode == TableStorageError.InvalidTableName)
        {
            Console.WriteLine("The table name is invalid.");
            return false;
        }

        Console.WriteLine("Something went wrong: {0}", ex);
        return false;
    }
}
```

Qualquer erro será identificado por uma exceção do tipo AzureTableStorageException. Note-se a propriedade ErrorCode.

ITableStorageService

E um exemplo do segundo caso:

```
private static async Task<bool> UseTableStorageWithEntitiesAsync()
{
    ITableStorageService service = ServiceProvider.GetRequiredService<ITableStorageService>();

    try
    {
        ITableReference table = service.GetTable("sampletable2");

        await table
            .CreateIfNotExistsAsync()
            .ConfigureAwait(false);

        List<MyEntity> entities = new List<MyEntity>()
        {
            new MyEntity() { Key1 = "partition1", Key2 = "row1", Value1 = "text", Value2 = 20, Value3 = 10.1, Value4 = DateTime.Now },
            new MyEntity() { Key1 = "partition1", Key2 = "row2", Value1 = "text", Value2 = 11, Value3 = null, Value4 = DateTime.Now.AddDays(10) }
        };

        await table
            .InsertOrReplaceEntitiesAsync<MyEntity>(entities)
            .ConfigureAwait(false);

        MyEntity entity = await table
            .TryRetrieveEntityAsync<MyEntity>("key1", "row2")
            .ConfigureAwait(false);

        return true;
    }
    catch (AzureTableStorageException ex)
    {
        if (ex.ErrorCode == TableStorageError.InvalidTableName)
        {
            Console.WriteLine("The table name is invalid.");
            return false;
        }

        Console.WriteLine("Something went wrong: {0}", ex);
        return false;
    }
}
```

```
public class MyEntity : TableEntity
{
    #region Public Properties

    2 references
    public string Value1
    {
        get;
        set;
    }

    2 references
    public int Value2
    {
        get;
        set;
    }

    2 references
    public double? Value3
    {
        get;
        set;
    }

    2 references
    public DateTime Value4
    {
        get;
        set;
    }

    #endregion
}
```

Repare-se como a entidade (POCO) é definida para representar o registo na table storage (derivado de TableEntity).

Outro aspeto importante tem a ver com o insert das entidades. Por defeito, uma única operação de insert tem que operar sobre registos com a mesma partition key (caso contrário ocorre um erro).

Obviamente, os dois modos (records e entidades) também podem ser combinados.

Querying

O serviço permite extrair registos das tabelas das seguintes formas:

- Extração de um registo específico (query sobre PartitionKey e RowKey): RetrieveRecord, RetrieveEntity, TryRetrieveRecord
- Extração de todos os registos de uma partition key: RetrieveRecords(key)
- Extração de todos os registos da tabela: RetrieveRecords() – ATENÇÃO: operação potencialmente muito lenta!
- Execução de queries (sobre quaisquer colunas da tabela)

É fundamental compreender os seguintes aspetos do funcionamento da table storage para não acabar com queries extremamente lentas:

- A tabela é indexada apenas por PartitionKey e RowKey.
- Todos os registos da mesma partition key estão juntos (no mesmo data center, digamos assim), os registos de partition keys diferentes podem estar ou não.
- As operações sobre table storage são sempre paginadas. Uma operação pode implicar N requests ao serviço no Azure (tantas quantas as páginas necessárias).
- O ITableStorageService não fornece meio na API para controlar a paginação. Ele próprio usa o mecanismo das bibliotecas da MS, mas tenta sempre retornar todos os registos.
- As queries sobre qualquer coluna que não a partition key ou a row key implicam sempre um “table scan”.
- De tudo isto se conclui que o serviço é mais adequado quando: (1) as tabelas não têm muitos registos (acima de 1 milhar), exceto quando todas as queries são sobre as colunas indexadas; (2) as queries são muito eficientes sobre PartitionKey/RowKey e muito ineficientes sobre qualquer outra coluna.

Querying

A API de querying permite:

- Definir filtros com condições AND ou OR.
- As condições podem ser combinadas.
- Definir o número máximo de registos a retornar.
- Selecionar as colunas da tabela retornadas no resultado.

Eis um exemplo:

```
1 reference
private static async Task<bool> UseTableStorageWithQueriesAsync()
{
    ITableStorageService service = ServiceProvider.GetRequiredService<ITableStorageService>();

    try
    {
        ITableReference table = service.GetTable("sampletable2");

        await table
            .CreateIfNotExistsAsync()
            .ConfigureAwait(false);

        ITableQuery query = new AzureTableQuery()
            .Where(
                AzureTableQueryCondition.And(
                    new AzureTableQueryCondition("PartitionKey", TableQueryComparison.Equal, "key1"),
                    AzureTableQueryCondition.Or(
                        new AzureTableQueryCondition("Value2", TableQueryComparison.Equal, 11),
                        new AzureTableQueryCondition("Value2", TableQueryComparison.GreaterThan, 20))))
            .Select("PartitionKey", "RowKey", "Value1")
            .Take(10);

        IList<TableRecord> records = await table
            .RetrieveRecordsAsync(query)
            .ConfigureAwait(false);

        return true;
    }
    catch (AzureTableStorageException ex)
    {
        if (ex.ErrorCode == TableStorageError.InvalidTableName)
        {
            Console.WriteLine("The table name is invalid.");
            return false;
        }

        Console.WriteLine("Something went wrong: {0}", ex);
        return false;
    }
}
```

Boas práticas

1. Nunca, em momento algum, referenciar os tipos das bibliotecas do Azure (ex.: CloudTable).
2. Preferir o serviço blob storage ao serviço table storage para armazenar estado que não requer (ou requer muito pouco) querying (o serviço é mais económico).
3. Indicar explicitamente sempre, para a blob storage, a visibilidade pública dos containers/blobs.
4. Preferir o serviço de table storage para armazenar tabelas pequenas (é uma péssima solução para implementar, por exemplo, mecanismos de logging).
5. Preferir o modo de entidades sobre o modo de registos da table storage (mas há cenários em que o segundo é mais adequado, por exemplo, quando as colunas são mais dinâmicas).
6. Minimizar as queries sobre colunas para lá da PartitionKey/RowKey.
7. Minimizar os registos devolvidos pelas queries.
8. Não fazer permanentemente validações nas operações como, por exemplo, verificar se a tabela X existe (as operações retornam erros conhecidos em ErrorCode para esses casos).
9. Usar background services para inicializar as tabelas na primeira execução da aplicação.

Pipelines e caching

Como já foi referido antes, os serviços de storage podem beneficiar se forem associados com caching e pipelines.

Existem vários micro serviços com exemplos disso mesmo.

Este é o caso do DLS (Data Lookup Service):

```
/// <inheritdoc />
32 references | Hugo Lourenço | 1 author, 1 change
public virtual async Task<OperationResult<VatNumberInfo>> LookupAsync(string countryCode, string vatNumber)
{
    // Validation

    SmartGuard.NotNullOrEmpty(() => countryCode, countryCode);
    SmartGuard.NotNullOrEmpty(() => vatNumber, vatNumber);

    // Normalize inputs

    countryCode = countryCode.ToUpperInvariant();
    vatNumber = vatNumber.ToUpperInvariant();

    // Logging

    this.Logger.LogDebug($"Looking up VAT number '{countryCode}.{vatNumber}'...");

    // Create the pipeline
    // When the country code is PT we no dot enable the VIES service handler

    IPipeline<VatNumberLookupContext> pipeline = null;

    if (countryCode.EqualsNoCase("PT"))...
    else
    {
        pipeline = new PipelineBuilder<VatNumberLookupContext>()
            .Use<VatNumberLookupHandlerMemory>()
            .Use<VatNumberLookupHandlerStorageVIES>()
            .Use<VatNumberLookupHandlerServiceVIES>()
            .Build(this.ServiceProvider);
    }

    // Execute the pipeline

    VatNumberLookupContext context = new VatNumberLookupContext(countryCode, vatNumber);
    await pipeline.InvokeAsync(context).ConfigureAwait(false);

    // Result

    VatNumberInfo result = context.Result;

    // Return

    return OperationResult<VatNumberInfo>.Success(result);
}
```

A pesquisa segue esta lógica:

- Primeiro procura em cache (HandlerMemory).
- Se não existir, procura na table storage (HandlerStorageVIES).
- Se não existir, invoca o serviço do VIES (HandlerServiceVIES).
- Adiciona o registo à tabela storage (duração de dias).
- Adiciona o registo à cache (duração de horas).
- Retorna o resultado.