Hydrogen Sessions

#2 - Extension Methods

Extension Methods

Os extension methods são uma “construção” que foi adicionada ao C# na versão 3.0 (Visual Studio 2008) com o propósito de adicionar comportamento (métodos) a um tipo qualquer sem necessidade de o modificar ou compilar.

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods

Originalmente, o conceito foi criado em grande medida para suportar os operadores LINQ.

Como são um tipo de método estático, não podem ser adicionados a tipos estáticos.

São particularmente úteis para adicionar comportamento a tipos de livrarias externas.

São muito utilizados no lugar de classes helper, que se usavam muito antes.

Exemplos

```
public static bool EqualsNoCase(this string text, string value)
{
    // Null

    if (text is null)
    {
        return value is null;
    }

    // Same reference

    if (ReferenceEquals(text, value))
    {
        return true;
    }

    // Result

    return string.Compare(text, value, StringComparison.OrdinalIgnoreCase) == 0;
}
```

```
internal static bool IsBadRequest(this CloudException exception)
{
    if (exception != null && exception.Response != null)
    {
        return exception.Response.StatusCode == HttpStatusCode.BadRequest;
    }

    return false;
}
```

Boas práticas

#1 - Nome da classe

É regra geral organizarem-se os métodos de extensão de acordo com o tipo que estendem e coloca-los todos numa única classe com o nome do tipo estendido mais “Extensions”.

Se o método de extensão é para o tipo HttpRequest, cria-se uma classe com o nome HttpRequestExtensions.

#2 - Namespaces

Se analisarmos o código fonte de muitos componentes do próprio .NET Core, observaremos que as classes com os métodos de extensão são criadas frequentemente com o mesmo namespace do tipo estendido. Ou seja, para System.Net.Http.HttpRequestMessage, surge a classe System.Net.Http.HttpRequestMessageExtensions (mesmo que numa livraria em que o namespace base seja Microsoft.Extensions.Logging).

Esta abordagem tem a vantagem óbvia de facilitar o intellisense mas tem um problema fundamental que levou a que no Hydrogen não se siga esta abordagem: causa conflitos sem resolução possível pelo compilador quando 2 livrarias diferentes acabam a definir dois tipos exatamente com o mesmo nome.

Por isso, no Hydrogen, as classes de extensão usam o namespace da livraria em que estão, não do tipo que estendem (ver, por exemplo: HttpRequestMessageExtensions em IdentityModel.Client).

No Hydrogen o ficheiro de determinada classe deve estar numa pasta com o nome do namespace. A única exceção são as classes de extensão sobre IServiceCollection e os tipos associados, usadas no mecanismo de injeção de dependências, que ficam numa pasta com o nome DependencyInjection mas não têm esse sufixo no namespace. Isto serve exclusivamente para separar essa lógica da restante.

Boas práticas

#3 - Helper classes

Como foi dito antes, com os métodos de extensão não deveria ser necessário criar helper classes (exemplo: ConvertHelper), exceto quando se pretende estender classes que são estáticas (exemplo: ConsoleHelper).

Todos conhecemos algures um developer que resolvia tudo com helper classes. Em muitos aspetos, isso é só uma forma mais sofisticada de spaghetti code.

#4 - Métodos desnecessários

Um erro comum que ocorre neste tema é a criação de métodos que não são necessários de todo, ou porque não são necessários mesmo ou porque já existem outros métodos de extensão que fazem o mesmo ou quase. Não é fácil encontrar estes casos, desde logo porque o intellisense não vai ajudar muito. Mas é preciso procurar, antes de criar o novo método.

Um exemplo deste caso é o método IDictionaryExtensions.AddOrUpdate(). O comportamento do IDictionary já é esse.

```
myDic[key] = value;
myDic.AddOrUpdate(key, value);
```

Boas práticas

#5 - “Overextension”

Um método de extensão deve ser extremamente útil (usado muitas vezes) ou então deve ser implementado como um método privado da classe que o chama. Para observar o princípio da simplicidade.

Quantas vezes será necessário fazer Dispose explícito dos elementos de um dicionário como está implementado em IDictionaryExtensions.Dispose()? Já agora, neste caso, para o consumidor, isto não cria a ilusão que o IDictionary em si é disposable?

#6 - Public vs Internal

Todos os métodos públicos são parte da API e não vão poder mudar mais. Mas um método internal, não.

Os métodos de extensão podem ser implementados por defeito como internal, junto das classes (na livraria) onde são úteis. Deve seguir-se essa abordagem quando a utilidade generalizada de determinado método for discutível.

Boas práticas

#7 - Intellisense

A abordagem dos namespaces descrita torna mais difícil encontrar os métodos de extensão (é preciso primeiro adicionar o using correto). Ainda assim, o inverso causaria dificuldades maiores.

Outro problema prende-se com o nome da classe de extensão. Torna mais difícil encontrar o tipo original (experimentem fazer intellisense sobre IDic numa classe que tem o using para Primavera.Hydrogen mas não tem para System.Collections.Generic).

#8 - Documentação

Como acontece para tudo o resto, é fundamental descrever devidamente na documentação o que faz cada método de extensão. É fundamental descrever devidamente os parâmetros e o resultado. A secção remarks pode adicionar informação pertinente sobre o comportamento do método.