Instituto Superior de Engenharia de Lisboa
Licenciatura em Engenharia Informática e de Computadores
Ambientes Virtuais de Execução
2020

A biblioteca dotnet **Jsonzai** para processamento de dados em formato JSON (https://www.json.org/). Esta biblioteca disponibiliza uma classe `JsonParser` que pode ser usada para transformação de uma *string* JSON numa instância de uma classe compatível (e.g. `Student`) conforme ilustrado no exemplo seguinte:

```
string src = "{name: \"Ze Manel\", nr: 6512, group: 11}";
Student std = (Student) JsonParser.Parse(src, typeof(Student));
Assert.AreEqual("Ze Manel", std.Name);
Assert.AreEqual(6512, std.Nr);
Assert.AreEqual(11, std.Group);
```

A class `JsonParser` usa uma instância de uma classe auxiliar `JsonTokens` para percorrer os elementos da *String* JSON fonte. O algoritmo de `JsonParser` é recursivo, criando instâncias de classes ou arrays e preenchendo respectivamente os seus campos ou elementos com valores primitivos ou instâncias de outras classes ou arrays e assim sucessivamente.

O nome das propriedades JSON pode ser definido indiferentemente em maiúsculas ou minúsculas. A classe correspondente não pode ter propriedades com nomes que se distingam entre si apenas por terem letras maiúsculas ou minúsculas.

Exemplo de entidades de domínio dos testes:

- `Classroom` que agrega um conjunto de instâncias de `Student`
- `Account` com um saldo (*balance*) e um conjunto de movimentos de conta (*transactions*)
- Um outro exemplo ao seu critério.

As propriedades da classe destino podem ter nomes distintos dos nomes usados na representação em JSON. Por exemplo, uma propriedade em JSON pode ter o nome `birth_date` e em C# `BirthDate`. Para resolver a correspondência entre propriedades de nome distinto existe uma anotação `JsonProperty` que pode ser usada sobre propriedades de uma classe em C# indicando o nome correspondente em JSON (e.g. `[JsonProperty("birth_date")]`).

Algumas classes como `DateTime`, `Guid` ou `Uri` não podem ser inicializadas através das suas propriedades. Para tal, é disponibilizada uma anotação `JsonConvert` que permita estabelecer de que forma o `JsonParser` deve instanciar estas classes. Por exemplo uma propriedade `DueDate` do tipo `DateTime` pode ter uma classe associada `JsonToDateTime` que sabe criar uma instância de `DateTime` a partir da sua representação em *string*. Para tal a propriedade `DueDate` pode indicar essa correspondência através da seguinte anotação:
`[JsonConvert(typeof(JsonToDateTime))] DateTime DueDate { get; set; }`

A classe `JsonParsemit` tem o mesmo comportamento de `JsonParser`, mas **NÃO usa reflexão na atribuição de valores às propriedades**. Note, que **continua a ser usada reflexão na**

leitura da *metadata*, deixando apenas de ser usada reflexão em operações como `<property>.SetValue(…)`.
A atribuição de valores a propriedades é realizada directamente com base em código IL emitido em tempo de execução através da API de `System.Reflection.Emit`.
Para tal, `JsonParsemit` gera em tempo de execução implementações de classes, em que **cada classe** tem a capacidade de fazer a **atribuição de um valor a uma determinada propriedade**.

A aplicação consola **JsonzaiBenchmark** compara o desempenho do método `Parse()` entre as classes `JsonParsemit` e `JsonParser`.

O método `IEnumerable SequenceFrom(string filename, Type klass)` retorna uma sequência *lazy* para os dados JSON contidos no ficheiro `filename`. Este método assume que o ficheiro fonte tem como elemento raíz um *array* JSON, dando excepção caso o elemento raíz seja de outro tipo JSON, como por exemplo um objecto.

As configurações que são suportadas através do *custom attribute* `JsonConvert` podem ser adicionadas também através de métodos públicos genéricos de `JsonParser`:

- Estes métodos são genéricos em conformidade com o tipo das propriedades e dos conversores.
- A operação de conversão é passada num parâmetro através de um *delegate*.