02

Conhecendo o problema do cliente

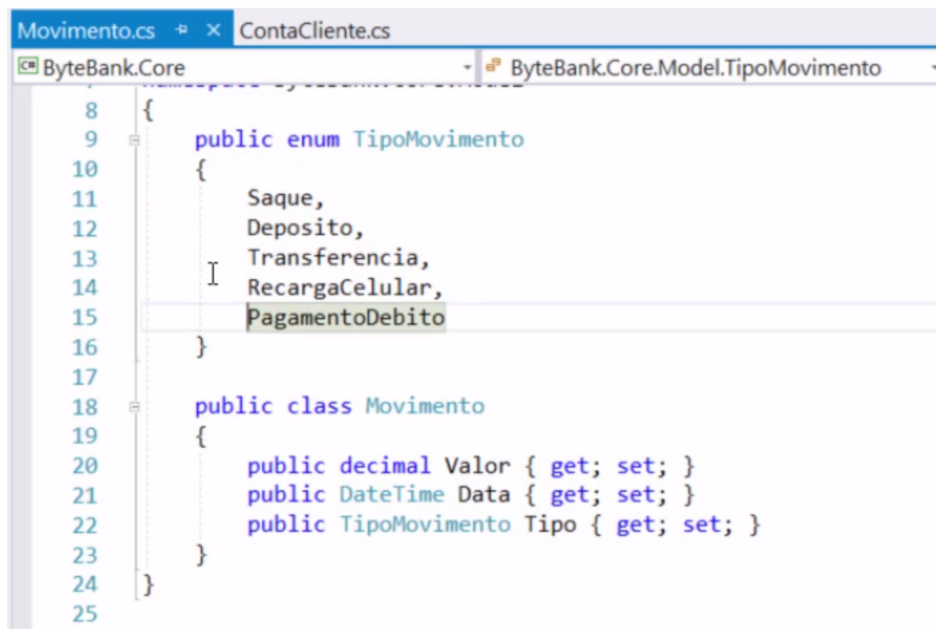
Transcrição

Temos uma aplicação rodando, o **ByteBank**. Trata-se de um banco como os que já estamos acostumados, responsáveis por cuidar de conta corrente, investimentos, movimentações bancárias em geral. A nossa aplicação faz a consolidação destes dados, colhendo toda a movimentação dos clientes ao fim do dia e fazendo vários cálculos financeiros, ajustes, retornando-os posteriormente para os usuários e funcionários do ByteBank.

A app já está construída, vamos abri-la pelo programa Microsoft Visual Studio 2017 (é possível executá-la em versões anteriores também). É um projeto simples, com uma *Solution* e dois projetos, o `ByteBank.Core` e o `ByteBank.View`. A modelagem, repositório e serviço referentes à aplicação estão no primeiro.

Dentro do modelo, encontra-se tudo que envolve a `ContaCliente`, objeto que representa a conta do nosso cliente no ByteBank, como o próprio nome indica. Há `NomeCliente`, que armazena o nome do cliente, o `decimal` que mostra os investimentos, uma lista de `Movimentacoes`.

Pressionaremos a tecla `F12` com o cursor em cima da classe `Movimento`, para verificarmos a implementação com mais detalhes. Nela, existem uma `Data` e um `Tipo` e, por meio deste último, o cliente pode realizar saques, depósitos, transações básicas que fazemos no dia a dia, definidas como vemos abaixo:



Vamos notar que a modelagem está muito simples e poderia ser feita de várias outras formas. Porém, para nosso objetivo e tipo de aplicação, está indo muito bem.

Além do modelo, temos um repositório (`ContaClienteRepository.cs`), com implementação de desenvolvimento que nos retorna todos os clientes, e o serviço (`ContaClienteService.cs`), que faz a consolidação, aquele trabalho "bruto" comentado anteriormente, de cálculos de reajustes financeiros de todos os clientes, investimentos e afins, ao fim de cada dia. Ele possui um método público que nos retorna a *string* com o resultado.

Ao selecionarmos `ByteBank.View` no menu à direita da tela, veremos o projeto construído em cima do WPF. Vamos abrir `MainWindow.xaml`. As telas e janelas foram construídas nesta linguagem de marcação chamada `xaml`, muito semelhante ao

`xml`. Na realidade, não precisaremos nos preocupar muito com isto, pois montar as telas em `.xaml` no WPF é tarefa do designer.

Precisamos saber, no entanto, que existe um botão denominado `BtnProcessar`. Quando clicado, ele será responsável pelo processamento. No mesmo arquivo, teremos também uma lista, chamada `LstResultados`.

Vamos iniciar a aplicação clicando em "*Start*" na barra de edição do Visual Studio. Vemos a app com o logo, um resumo em forma de texto como resultado, e o retângulo em que este resumo deve aparecer é a lista (`LstResultados`) citada anteriormente. Mais abaixo, veremos o botão `BtnProcessar`.

A reclamação do cliente é exatamente esta: a app está lenta, demorando demais para ser executada. Analisaremos o que realmente ocorre quando clicamos em "Fazer Processamento", e identificar o razão. Em seguida, fecharemos a aplicação. De volta ao Visual Studio, selecionaremos `MainWindow.xaml.cs`, onde se localiza o código a ser executado.

No construtor, só criamos o repositório (`ContaClienteRepository();`) e o serviço (`ContaClienteService();`). O `BtnProcessar_Click` é o código executado ao se clicado no botão pelo usuário. O primeiro passo consiste em obtermos todas as contas dos clientes, armazenadas na variável `var contas`. Montaremos uma lista vazia com o resultado usando `var resultado`.

Pressionaremos `F12` em cima de `AtualizarView` para verificar sua implementação, e veremos que ele monta o resumo com a quantidade de clientes processados em determinado tempo, atualizando a lista de resultados (`LstResultados`), assim como a mensagem de texto. A linha abaixo é chamada para limpar a tela de qualquer resquício de processamento feito anteriormente.

```
AtualizarView(new list<string>(), TimeSpan.Zero);
```

Para obtermos as métricas, teremos um contador (`var inicio = DateTime.Now;`), ou seja, vamos armazenamos o início do processamento. Agora, de fato, faremos o processamento, por meio de:

```
foreach (var conta in contas)
{
  var resultadoConta = r_Servico.ConsolidarMovimentacao(conta);
  resultado.Add(resultadoConta);
}
```

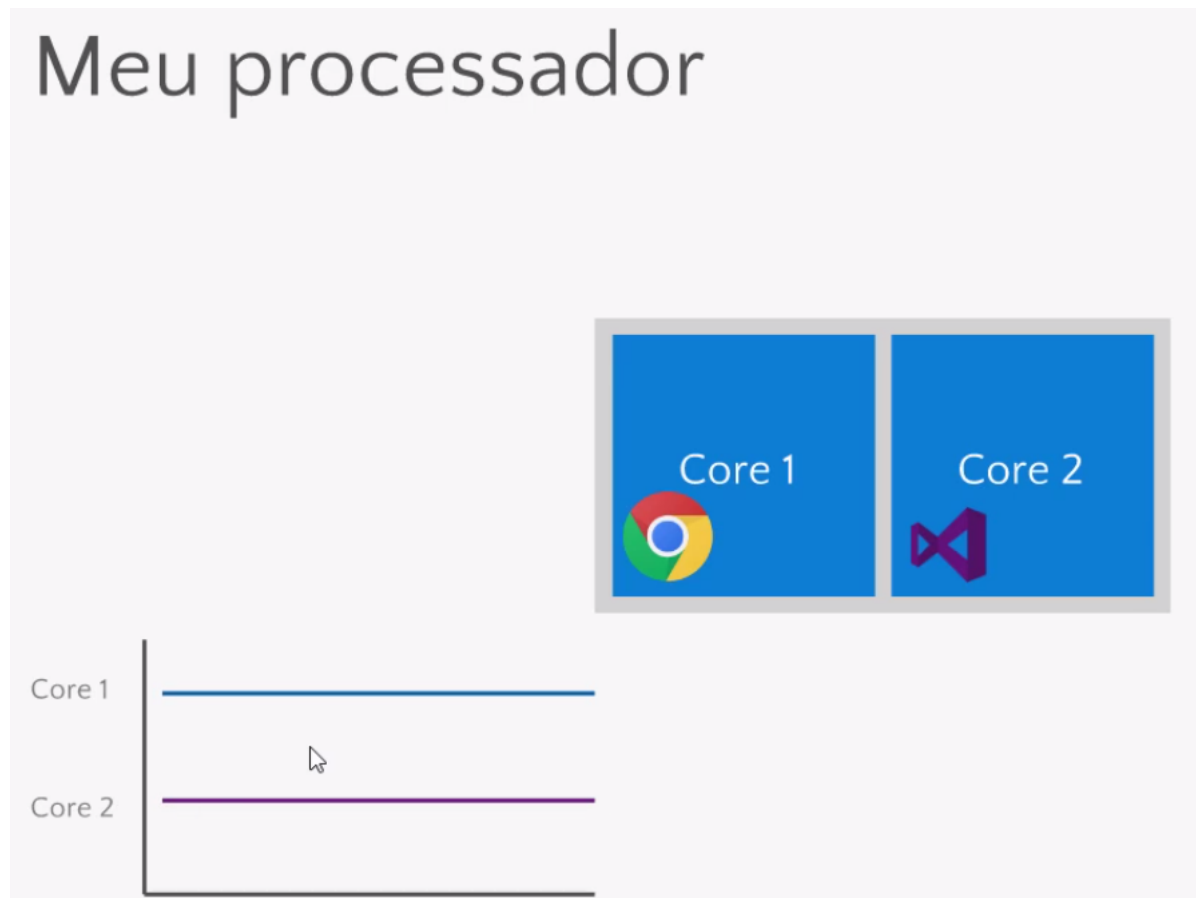
Utilizando-se um laço de repetição para cada conta na lista de contas, chamaremos o serviço, faremos a consolidação de seus movimentos, adicionando-se o resultado do serviço à lista de resultado. Marcamos também o tempo de término (`var fim = DateTime.Now;`) e tela será atualizada com o resultado obtido.

Observe que este processamento é feito de forma totalmente independente para cada cliente. É isto que acontece na vida real, afinal, não queremos que o saldo bancário de uma pessoa seja influenciado pelo saldo de outra.

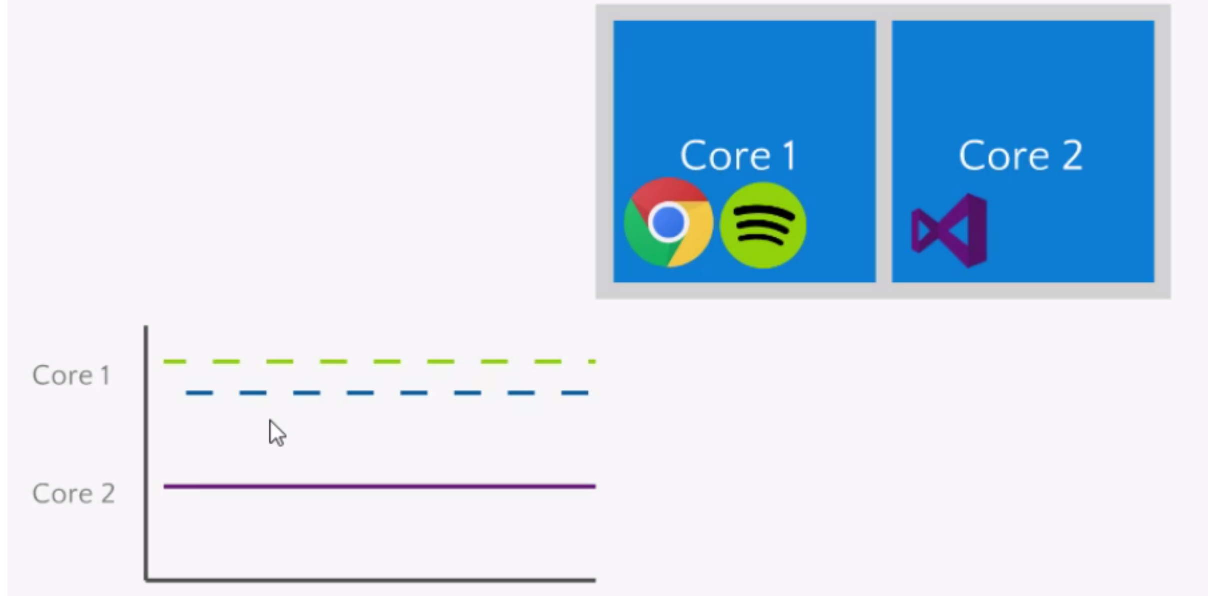
Antes de vermos exatamente o caso do cliente, vamos pensar em como as aplicações são executadas em nosso computador. Neste exemplo, teremos um computador *dual core*, um processador com dois núcleos de processamento. Montei um gráfico simples para mostrar o processamento de cada núcleo (*core*) em relação ao tempo (eixo `x` do gráfico).

Quando abrimos uma aplicação, neste caso o Google Chrome, o sistema operacional precisa executá-lo em um dos *cores*, o `1`, por exemplo, que fica em atividade pelo tempo necessário para renderizar a página, carregar, fazer downloads de imagens, enquanto o outro *core* não possui nenhuma aplicação por enquanto.

Com o Google Chrome e o Visual Studio abertos simultaneamente, porém em núcleos distintos, o sistema operacional gerencia o processador de forma que seu uso seja otimizado ao máximo:



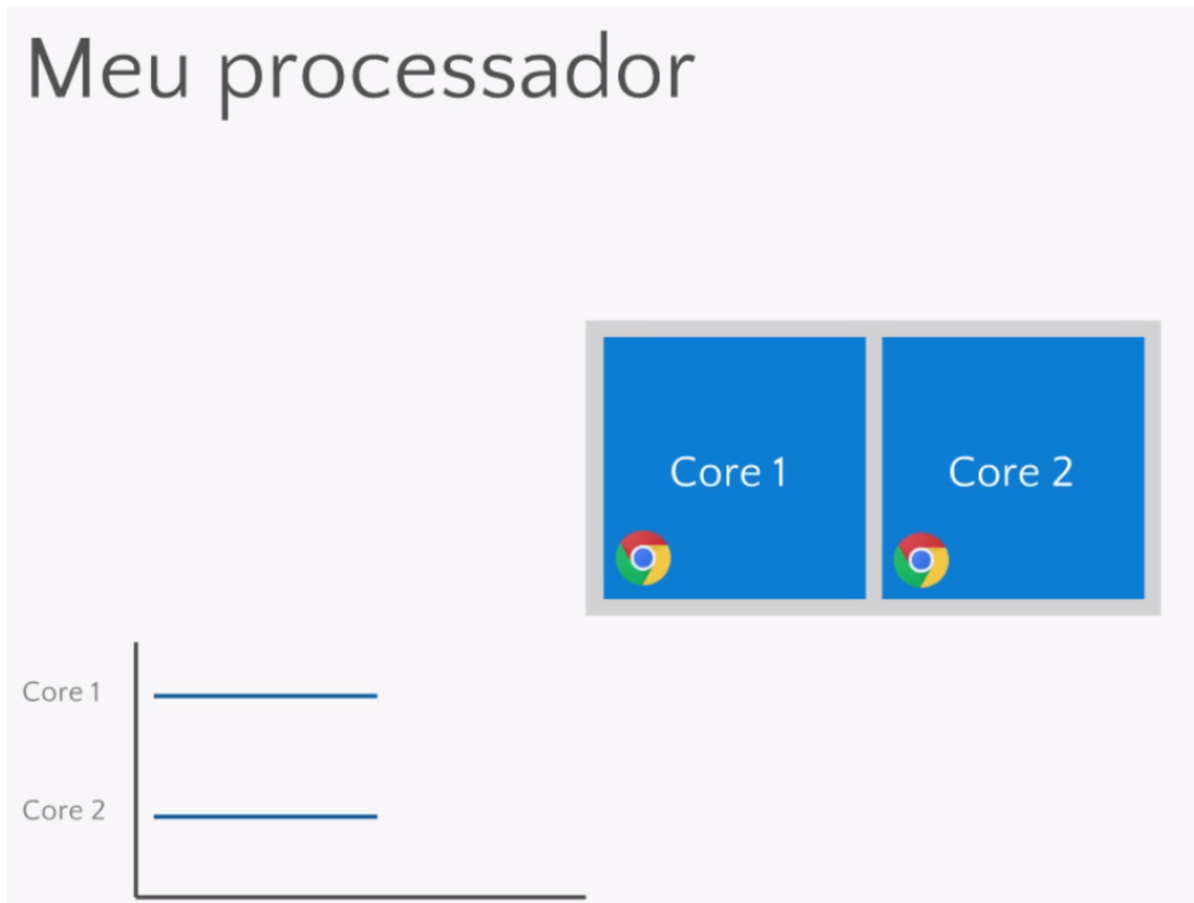
Se estamos escrevendo código e ele está abrindo a janela de autocompletar, por exemplo, tudo isto é feito em outro *core* (`2`). O que acontece se abrirmos mais de uma aplicação em um mesmo núcleo? No exemplo, utilizamos o Spotify. Não temos um terceiro núcleo para colocá-lo. Neste caso, o sistema operacional ficará intermediando o tempo de uso de cada *core* para mais de uma aplicação.



Portanto, se antes o *Core* `1` executava apenas o Google Chrome de forma constante, agora ele se detém, executa um pouco de um programa, pausa, começa a executar outro, alternando-se de um para o outro tão rapidamente, sem que o usuário final consiga perceber. No entanto, na prática, o tempo de execução de cada aplicação é maior, pois ele terá menos tempo de CPU para a realização deste processamento, afinal, ele estará dividindo seu *core* com outra aplicação.

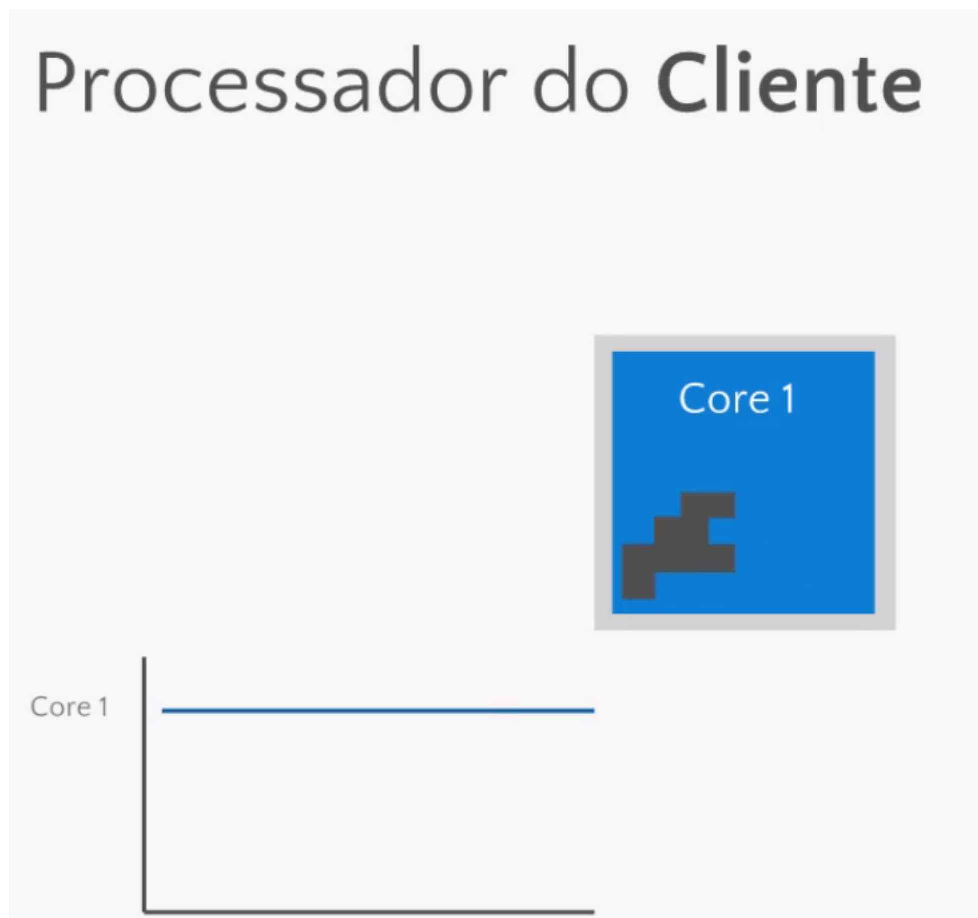
Na situação atual temos apenas uma aplicação sendo executada, sem o Visual Studio ou o Spotify estarem funcionando em paralelo. Inclusive, o ícone do Google Chrome está bem grande nesta representação justamente porque ele possibilita que sejam abertas várias abas, das quais utilizaremos uma, enquanto as outras continuam carregando imagens e sendo renderizadas. Teremos sua performance afetada.

Temos outro *core* livre, sem uso. Então, o que o Chrome faz é executar várias tarefas separadamente, em *cores* distintos, conforme sua disponibilidade. Neste exemplo, vamos supor que temos um site aberto, com outra aba aberta também sendo processada um *core* diferente:



Nós conseguimos carregar duas páginas ao mesmo tempo, com renderizações simultâneas, downloads, otimizando-se o uso do processador, afinal, trata-se de um *dual core*. Apenas o Google Chrome está sendo executado neste momento. Em vez de termos um grande processo que demora este tempo para ser executado, podemos dividi-lo duas linhas menores de execução. Por "linha de execução", vamos pensar do momento em que apertamos "Enter" e a página começa a ser carregada, até o fim de seu carregamento, quebrando-se, assim, o tempo necessário.

No caso do ByteBank, o que está acontecendo com seu código, e por que o cliente está reclamando? Esta é a representação do *core* do cliente, com apenas um processador. O ByteBank é executado no **servidor dedicado**, o que significa que não se trata de uma máquina em que o usuário fica utilizando o Google Chrome, Visual Studio, Spotify, ou qualquer outra aplicação do tipo ao mesmo tempo. O servidor dedicado atua apenas para esta aplicação em específico.



Atualmente, o que acontece é que a aplicação roda em um único *core*. O cliente notou a lentidão e as reclamações chegaram ao chefe, e depois ao gerente de TI, que decidiu pela aquisição de uma máquina mais potente. Passa-se a utilizar então uma máquina com dois núcleos de processamento, e espera-se que o tempo de processamento da aplicação caia pela metade, mas na verdade, ele se manteve, pois quando o ByteBank começou a ser executado, apenas um dos núcleos faz o esperado. A app só realiza uma linha de execução.

Ao abrir a aplicação, o usuário clica em "Fazer Processamento" e, quando chega ao fim deste, tudo acontece na mesma linha de execução. Não fazemos uso de várias execuções paralelas e, por isso, mesmo usando uma máquina mais potente, o cliente não nota o ganho de performance.

O que vamos aprender a fazer agora é justamente quebrar esta aplicação em pedaços menores para otimizarmos o uso do processador da máquina do cliente.