01

Criando uma barra de progresso

Transcrição

Progredimos bastante no curso e nossa aplicação finalmente está responsiva, durante um trabalho muito pesado que acontece em *background* paralelamente. Estamos utilizando bem os recursos da CPU e o código está elegante - com o `AsyncAwait`, surgido na versão `5` do C#. Deixamos de nos preocupar com o `TaskScheduler`, com sua recuperação e reuso.

Agora, vamos dar uma olhada no funcionamento da aplicação, clicando em "*Start*" e em "Fazer Processamento" quando a app for aberta. Com a aplicação responsiva, enquanto uma tarefa pesada é processada, teremos a interface gráfica liberada para darmos algum retorno ao usuário. Faremos uso desse recurso. Afinal, o grande benefício de termos uma interface responsiva é dar um *feedback* ao usuário, informando-o sobre o andamento do processamento.

Até então, ao clicarmos em "Fazer Processamento", lê-se a mensagem "Processamento de 0 clientes em 0.0 segundos!". Se eu fosse um usuário, pensaria que a aplicação está quebrada, com *bug*. Será que o usuário ficará todo este tempo parado, olhando para a tela, sem *feedback* de que algo está ocorrendo, até que o resultado seja mostrado na tela? Acho que não, até porque neste caso, é bem simples.

Para o usuário final, isto pode ser muito mais vasto, com base em um número maior de clientes, podendo ser um trabalho mais duradouro. Vamos melhorar isto?

Começaremos tirando a frase "Processamento de 0 clientes...", que é resultado da utilização do `AtualizarView`. Criaremos um método dedicado para limpar a *view* (`LimparView`), sem recebermos nenhum parâmetro, pois já sabemos o que iremos colocar na tela. Usaremos o `LstResultados` e seus `ItemsSource`, recebendo uma referência nula. O `TxtTempo`, responsável por aquela mensagem, será atualizado para nulo também. Agora que temos um método dedicado, vamos usá-lo.

```
private void LimparView()
{
  LstResultados.ItemsSource = null;
  TxtTempo.Text = null;
}
```

Substituiremos a linha `AtualizarView(new List<string>(), TimeSpan.Zero);` por `LimparView();`. Para mostrarmos o *feedback* ao usuário, várias práticas são muito bem aceitas no mundo da experiência do usuário e de interface gráfica. Podemos usar "`50%` completos", subindo este número para `55%`, `60%`, alterando-se a mensagem anterior para "`1` cliente de um total de `5` consolidado", atualizando-a. Há também a possibilidade de uso de uma barra de progresso.

Todas estas opções são similares no que diz respeito ao valor total, utilizado para calcular o progresso do que já foi feito e também do que resta. Uma barra de progresso funcionaria muito bem em nossa aplicação e, para colocá-la, precisaremos editar o arquivo `MainWindow.xaml` - presente em `ByteBank.View`. Costumeiramente, cabe ao designer mexer neste arquivo, porém é algo bem simples de se fazer. No documento, antes da linha `<ListView Name="LstResultados"`, colocaremos a barra de progresso.

Uma `ProgressBar` possui algumas propriedades interessantes e fundamentais: o valor de progresso que a barra possui, o mínimo, que neste caso é `0` (ou seja, `0` clientes consolidados é o valor mínimo, ninguém foi consolidado), e o máximo, que indica a localização em que a barra deve aparecer quando completa.

Quando estamos montando a tela, não sabemos se teremos `10`, `100` ou `1.000.000` clientes. Assim, definiremos o valor mínimo, sendo que o `value` pode ter qualquer atribuição de valor, porém não podemos determiná-lo também, por se tratar do tempo de execução. Atualizaremos esta propriedade depois, bem como `Maximum`, quando descobriremos quantas contas serão consolidadas.

Em relação à parte visual, definiremos uma altura (em pixels) para que a barra não fique tão pequena na tela. O `DockPanel` exige a definição de como seu elemento deve ser "dockado" no painel. Vamos colocá-lo posicionado embaixo ("*Bottom*"):

```
<!-- Barra de Progresso para dar feedback ao usuário -->
<ProgressBar Name="PgsProgresso"
             Minimum="0"
             Height="23"
             DockPanel.Dock="Bottom"/>
```

Voltando ao arquivo `MainWindow.xaml.cs`, teremos que atualizar o método `LimparView()`, uma vez que o valor da barra de progresso deverá ser limpo também.

```
private void LimparView()
{
  LstResultados.ItemsSource = null;
  TxtTempo.Text = null;
  PgsProgresso.Value = 0;
}
```

A próxima atualização será feita após a recuperação das contas a serem consolidadas. É neste momento que saberemos quantas contas serão processadas, também vamos descobrir o valor máximo da barra de progresso. Atualizaremos este valor, que será simplesmente um `Count` da conta.

```
private async void BtnProcessar_Click(object sender, RoutedEventArgs e)
{
  BtnProcessar.IsEnabled = false;

  var contas = r_RepositorioGetContaClientes();

  PgsProgresso.Maximum = contas.Count();

  LimparView();

  //...
}
```

Vamos executar a aplicação; verificaremos que logo acima do botão onde se lê "Fazer Processamento", está a barra de progresso. Clicaremos neste botão, e aquela mensagem de antes não aparecerá mais. Em seguida, atualizaremos o valor da barra conforme o progresso vai acontecendo e cada cliente é consolidado.

Criamos o método assíncrono `ConsolidarContas()` antes, com uma lista de tarefas em que cada uma é responsável pela consolidação de um cliente. É neste momento que reportaremos o progresso ao usuário, utilizando uma barra de progresso. Voltaremos a expressão lambda para a forma mais complexa, com corpo, as chaves (" `{}` ") e o ponto e vírgula (" `;` "), bem como a sobrecarga `StartNew` de uma *task* que retorna valor.

Precisaremos guardar o valor obtido pela consolidação em uma variável, retornando o seguinte código:

```
private async Task<string[]> ConsolidarContas(IEnumerable<ContaCliente> contas)
{
  var tasks = contas.Select(conta =>
    Task.Factory.StartNew(() =>
    {
      var resultadoConsolidacao = r_Servico.ConsolidarMovimentacao(conta);
      PgsProgresso.Value++;
      return resultadoConsolidacao;
      })
  );

return await Task.WhenAll(tasks);
}
```

Feito isto, vamos executar a app. Veremos que apareceu um erro já esperado, trata-se do mesmo que tivemos no início do curso. Há uma *thread* que não é a principal (GUI) tentando acessar um objeto da interface gráfica, o que não é permitido. Qual foi a solução encontrada naquele caso? Recuperamos o `TaskScheduler` da *thread* principal, usando `taskSchedulerGui` para fazer uma atualização na tela.

```
private async Task<string[]> ConsolidarContas(IEnumerable<ContaCliente> contas)
{
  var taskSchedulerGui = TaskScheduler.FromCurrentSynchronizationContext();
  var tasks = contas.Select(conta =>
    Task.Factory.StartNew(() =>
    {
      var resultadoConsolidacao = r_Servico.ConsolidarMovimentacao(conta);
      PgsProgresso.Value++;

      Task.Factory.StartNew(
        () => PgsProgresso.Value++,
        CancellationToken.None,
        TaskCreationOptions.None,
        taskSchedulerGui
      );

      return resultadoConsolidacao;
      })
    );

    return await Task.WhenAll(tasks);
}
```

Criamos uma nova *task* utilizando o `Factory`, com o qual usamos o valor da barra de progresso. Precisaremos usar uma sobrecarga diferente no `StartNew`, que nos permita o uso de outro `TaskScheduler`. Anteriormente, só era possível utilizá-lo por causa do `ContinueWith()`, que possui uma sobrecarga que recebe uma *action* como primeiro parâmetro, e o `TaskScheduler` como segundo.

No entanto, como estamos criando uma nova tarefa e não queremos utilizar o `ContinueWith`, uma vez que `Task.Factory.StartNew()` é nossa *task* que dá um retorno relevante quando fazemos uma tarefa `WhenAll`. Criaremos uma *task*, cuja única responsabilidade será atualizar a *view*. Dentre as sobrecargas disponíveis para utilização, optamos por uma que recebe uma *action*.

Aprenderemos melhor sobre `CancellationToken` mais adiante, por ora, utilizaremos seu valor *default*, fazendo o mesmo com `TaskCreationOptions`. Depois, utilizaremos o parâmetro `TaskScheduler`.

Rodaremos novamente a aplicação, verificando as alterações feitas e acompanhando o processamento pelo Gerenciador de Tarefas... Deu outro erro! Para corrigi-lo, comentaremos `PgsProgresso.Value++;`, deixando o código da seguinte forma:

```
private async Task<string[]> ConsolidarContas(IEnumerable<ContaCliente> contas)
{
  var taskSchedulerGui = TaskScheduler.FromCurrentSynchronizationContext();
  var tasks = contas.Select(conta =>
    Task.Factory.StartNew(() =>
    {
      var resultadoConsolidacao = r_Servico.ConsolidarMovimentacao(conta);

      // Não utilizaremos atualização do PgsProgresso na Thread de Trabalho
      // PgsProgresso.Value++;

      Task.Factory.StartNew(
        () => PgsProgresso.Value++,
        CancellationToken.None,
        TaskCreationOptions.None,
        taskSchedulerGui
      );

      return resultadoConsolidacao;
      })
    );

    return await Task.WhenAll(tasks);
}
```

Executaremos a aplicação de novo. É possível ver pelo Gerenciador de Tarefas que o processamento foi iniciado usando os `8` cores, terminando todos mais ou menos ao mesmo tempo.

O ByteBank finalizou a execução, então, vamos clicar em "Fazer Processamento" mais uma vez, sendo finalizado novamente depois de alguns instantes. A app dá um retorno muito melhor ao usuário.

Retornando ao código, observaremos que voltamos a explícitar o uso do `TaskScheduler` em `ConsolidarContas`. Vamos ver como deixar o código mais limpo.