Don’t keep state in your XAML item templates; put the state in the items

devblogs.microsoft.com/oldnewthing/20231226-00

December 26, 2023

Raymond Chen

When you apply a template to an item in XAML, the template is used to create an element (known as a “container”) which is ued to represent the item in the user interface. However, the pairing between items and containers can change over time, so you need to make sure you keep your state in the item and not in the container.

Some XAML collections are virtualizing collections, meaning that they generate XAML elements only when needed for display (or anticipated to be needed for display). If you create a XAML ListView with ten thousand items, the ListView will create containers only for the 50 items that are visible, plus a few hundred containers for the items that are just out of view, so they can be scrolled in quickly. But the other 9,000+ items will not have containers yet.

As you scroll through the collection, the items that you scroll too far away from lose their containers and become virtualized out. Meanwhile, items that scroll into view (or nearly so) become devirtualized and gain a container.

In the following diagram, we have a collection of 15 items. Items 7 through 9 are in view, so containers are created for those items so they can be displayed on the screen. We say that these items are “devirtualized” or “realized”. Furthermore, a few items that are just outside the view are also devirtualzed/realized. In the example, I’ll realize one screen’s worth before and after the view, so items 4 through 6, and items 10 through 12. In total, items 4 through 12 have been realized: 4 through 6 are ready for display if the user scrolls upward, 7 through 9 are on the display right now, and 10 through 12 are ready for display if the user scrolls downward. Let’s say that the items which are in view or just outside the view are in the “realization region”.

Item 1

Item 2

Item 3

Item 4	Container	
Item 5	Container	Almost in view
Item 6	Container	
Item 7	Container	
Item 8	Container	In view
Item 9	Container	
Item 10	Container	
Item 11	Container	Almost in view
Item 12	Container	
Item 13		
Item 14		
Item 15		

If the user scrolls downward one item, then item 4 falls out of the “almost in view” zone, and the item loses its container. Meanwhile, item 13 enters the “almost in view” zone, and it gains a container. Items 5 through 12 retain their containers since they are still within the realization region.

Item 1		
Item 2		
Item 3		
Item 4	(No container)	
Item 5	Container	
Item 6	Container	Almost in view
Item 7	Container	
Item 8	Container	
Item 9	Container	In view
Item 10	Container	
Item 11	Container	

Item 12	Container	Almost in view
Item 13	Container	
Item 14		
Item 15		

The relization region moves around as the user scrolls through the collection.

When an item scrolls out of the realization region, it is disconnected from its container, and the container returns to a small cache of “available containers”. When an item scrolls into the realization region, it is connected to a container (from the cache, if available).

The connection between an item and its container is only temporary. It’s like one of those political dramas with constantly-changing alliances.

This means that you cannot rely on the containers remembering anything for you, since you lose it when the item scrolls out of view. For example, here’s an example of a problematic item template:

```
<DataTemplate x:DataType="Widget">
  <StackPanel>
    <TextBlock Content="{x:Bind Name}" />
    <ListBox x:Name="ReportsList" />
    <Button Content="Load reports" Click="LoadReports" />
  </StackPanel>
</DataTemplate>
```

The idea here is that the list starts out empty, but if you click the “Load reports” button, then the code-behind populates the `ReportsList`.

If you do this, then strange things will happen when you start scrolling through the view. You might click “Load reports” for item 1, and then scroll all the way to the bottom, and when you scroll back up to item 1, those reports you loaded are gone! Even stranger, the reports you loaded for item 1 somehow show up in item 75.

What happened is that when you scrolled too far away, item 1 lost its container, and when you scrolled back to item 1, it got a different container, and the reports you loaded aren’t present in the new container.

This also explains why the reports you loaded into item 1 mysteriously appeared in item 75: The container that had been used for item 1 was recycled for item 75.

The problem is that you were recording state in the container, rather than recording it in the item itself. Containers can get lost and recycled, and that applies to any state you had stored in them.

The way to fix this is to keep the list of loaded reports in the item, not in the container:

```
<DataTemplate x:DataType="Widget">
  <StackPanel>
    <TextBlock Content="{x:Bind Name}" />
    <ListBox ItemsSource="{x:Bind LoadedReport}" />
    <Button Content="Load reports" Click="LoadReports" />
  </StackPanel>
</DataTemplate>
```

Storing the loaded reports in the item allows them to survive changes in container.

And while you're there, you can simplify matters by making `LoadReports` a method on the item rather than a method on the page.

```
<DataTemplate x:DataType="Widget">
  <StackPanel>
    <TextBlock Content="{x:Bind Name}" />
    <ListBox ItemsSource="{x:Bind LoadedReport}" />
    <Button Content="Load reports" Click="{x:Bind LoadReports}" />
  </StackPanel>
</DataTemplate>
```

If there are some parts of the data template that you can't (or don't want to) make driven by data-binding, you can reset the container to a "clean" state in your `ContainerContent-Changing` event handler. Details are available in the GridView and ListView optimization guide.