Tasks

Estimated time for completion: 30 minutes

Overview:

In this lab you will be working with an existing WPF that displays random pictures from a website. Currently the application performs all of the content retrieval synchronously. As the website can be very slow this locks the UI for unacceptable amounts of time. Your job is to use the new Task API to perform the picture retrieval asynchronously

Goals:

- Learn how to create tasks
- Learn how to chain tasks together
- Learn how to change the task scheduler such that work takes place on the UI thread

Lab Notes:

N/A

Part 1: Examine the existing code

In the first part of the lab you will take a quick tour of the existing application so that you can see the context of where your changes will be made

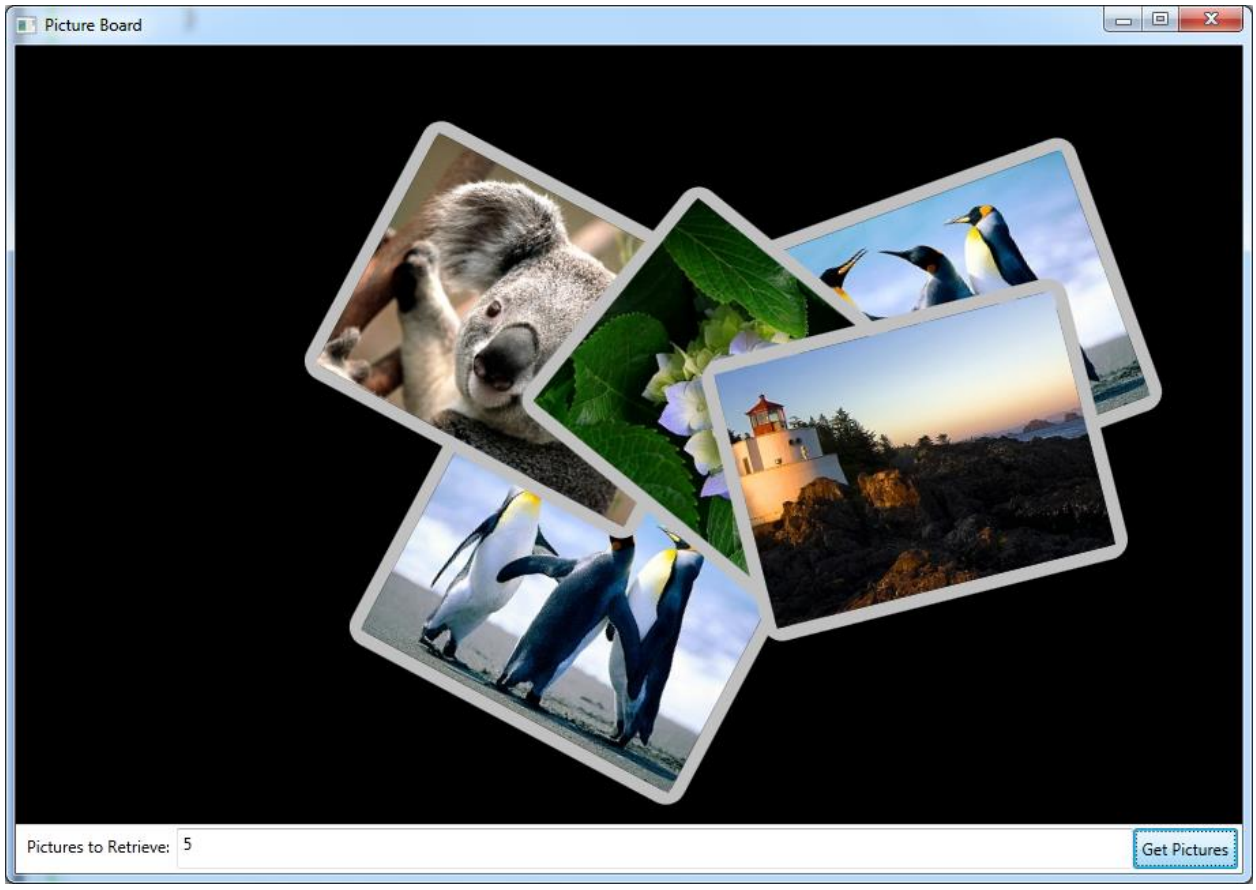
1. In Visual Studio 2010, open the starter solution **before/tasks.sln.** You will notice that it contains two projects: a REST based WCF service that returns the pictures and the WPF `PictureBoard` that displays them to the user
2. Open **program.cs** in the `SlowPictures` service project. Find the `PictureService` class's `GetPictures` implementation. You will see that there is a delay of up to 10 seconds for each picture to be delivered
3. Now go to the `PictureBoard` application and open **MainWindow.xaml** in the designer. You will see that the UI consists of a a textbox where you enter the number of pictures to retrieve, a button that performs the retrieval and a display area where the pictures will be displayed
4. **PictureItem.xaml** is a user control that provides a frame for each picture
5. The button has a `RoutedCommand` (defined in **Commands.cs**) that is wired up in the main window constructor. Find this in **MainWindow.xaml.cs** – you will see that when the button is clicked the `OnGetPictures` method is called
6. Go to the `OnGetPictures` method. This is where all the hard work takes place in the application. It takes the number of pictures requested and synchronously gets each one, creating a `PictureItem` and setting its `Source` to be the downloaded image. It then

places the item at a random position and angle on the main window's `Canvas` (called `board`)

7. Make sure the service has the rights to listen on its URL by running at an admin command prompt

```
NETSH HTTP ADD URLACL URL=http://+:9000/ USER=<Your user name>
```

8. Run the service and then run the application. Request five pictures and notice that the application locks up while the pictures are retrieved. Your job will be to change the `OnGetPictures` method to download the pictures asynchronously



Part 2: Using tasks to get the pictures

In this part of the lab you will take the Picture Board synchronous behavior and make it asynchronous using the new Task API from PFx

1. Open **MainWindow.xaml.cs** in the `PictureBoard` project and find the `OnGetPictures` method

2. Select all of the code within the loop from where the `PictureItem` is created until it is added to the `Canvas board` and extract it into a new method called `UpdateUI`. This will generate the method taking a `WebResponse` parameter. We will change this later.
3. We will now proceed to use Tasks to perform the requests for the pictures asynchronously. `WebRequest` already has async methods for making requests so we will use the Async Pattern wrapping functionality in the task factory to create the tasks.
4. Inside the loop, after you have created the `WebRequest`, create a local variable of type `Task<WebResponse>` called `task`. You will need to ass a `using` statement for the namespace `System.Threading.Tasks`
5. Intialize this variable by using the static member of the task factory class `FromAsync`. `FromAsync` takes an `IAsyncResult` (use the result of calling `BeginGetResponse` on the `WebRequest`) and a delegate that takes an `IAsyncResult` and returns a `WebResponse` (pass the `EndGetResponse` method of the `WebRequest` for this parameter)

```
Task<WebResponse> task =
   Task.Factory.FromAsync<WebResponse>(req.BeginGetResponse(null, null),
                                       req.EndGetResponse);
```

6. Compile and test the code so far. You should see the requests hit the service although currently we are doing nothing with the responses
7. Now we will process the responses. The neatest way with the task API is to chain a new task after the request one by using the `ContinuesWith` method on the request task. This takes a delegate of type `Action<Task<WebResponse>>` as a parameter. The `UpdateUI` method almost has the right signature so change the parameter from `WebResponse` to `Task<WebResponse>` and then in the body change the access of the `WebResponse` to use the `Result` property of the task parameter

```
void UpdateUI(Task<WebResponse> t)
{
  PictureItem item = new PictureItem();

  BitmapImage bmp = new BitmapImage();
  bmp.DownloadCompleted += delegate
  {
    t.Result.Close();
  };

  bmp.BeginInit();
  bmp.StreamSource = t.Result.GetResponseStream();
  bmp.EndInit();

  item.Source = bmp;

  double left = rnd.NextDouble() * board.ActualWidth;
  double top = rnd.NextDouble() * board.ActualHeight;
  Canvas.SetLeft(item, left);
  Canvas.SetTop(item, top);
```

```
  item.Angle = rnd.Next(-45, 45);

  board.Children.Add(item);
}
```

8. Use the changed `UpdateUI` method to chain a task to the request task using `ContinuesWith`

```
task.ContinueWith( UpdateUI );
```

9. Compile and run your code in the debugger (by pressing F5). You will notice that you get an exception – this is because you are trying to update the UI from a thread other than the UI thread
10. Change the `ContinuesWith` call take the synchronization context based scheduler as a second parameter by calling the static method of the `TaskScheduler` class `FromCurrentSynchronizationContext`

```
task.ContinueWith( UpdateUI,
                   TaskScheduler.FromCurrentSynchronizationContext() );
```

11. Compile and test your code – you should see the pictures rendered without blocking the UI thread
12. The final issue with the code is that we have no code in place that would deal with an exception on the task code and nowhere to wait on the spawned tasks without blocking the UI thread. Therefore, in the `MainWindow` constructor wire up an event handler for the `static` `UnobservedTaskException` event on the `TaskScheduler`. Show the exception details in a message box and set the exception as observed by using the `SetObserved` method on the event args. You can test this code by deliberately introducing an exception in the `UpdateUI` method (remember that you may have to make more than on request to see the error as the unobserved event handling results from the task finalizer noticing no one has observed the exception

---

Solutions

after\tasks.sln