CSC 415 ONLINE PHOTOALBUM: THE SEQUEL ASP.NET VERSION

GODFREY MUGANDA

In this project, you will convert the Online Photo Album project to run on the ASP.NET platform, using only generic HTTP handlers.

Because most of you are unfamiliar with C#, I am providing a series of steps to get you started with the use of C# in the project. I suggest classes in the .NET framework that are likely to be useful, and provide other hints that will be helpful to you.

ASP.NET allows applications to be configured with initialization parameters, but the process is somewhat more cumbersome than that of Java. Rather than use that, you should set path to the temp folder in the `Application_Start` method and store it with the keyword `"location"` in application scope.

1. The Assignment

1. Start by creating an empty ASP.NET web application by creating a project named ASPNETOnlinePhotos. Right click on the project node and add an `images` folder as well as a subfolder `storephotos`.

2. Add `Photo` class

```
using System;
namespace ASPNETOnlinePhotos
{
    public class Photo
    {
        public String Name { get; set; }
        public  String Description { get; set; }
    }
}
```

3. Create a folder for use as persistent storage. The folder should be named after your last name with a suffix of 1 to distinguish it from the folder for the last project. For example, if your last name is "Muganda" your folder would be named `C:\temp\Muganda1`.

4. Next, add a global application class and override the two methods

```
protected void Application_Start(object sender, EventArgs e)
{

}

protected void Application_End(object sender, EventArgs e)
{
```

```
}
```

to perform the tasks previously assigned to the context initialized and context destroyed methods of the the context listener in the Java version of the project.

5. We need to be able to work with a map, or dictionary, of `String` to `Photo`. You can find information on dictionary objects at

`https://msdn.microsoft.com/en-us/library/xfhwa508(v=vs.110).aspx`

Make sure you scroll all through the web page and peruse the code example at the end.

Now we need the ability to serialize and deserialize the dictionary. Begin by creating a map in the global application class's `Application_Star()` method and putting it into application scope:

```
protected void Application_Start(object sender, EventArgs e)
{
   // Create a dictionary and add 2 photos
   Dictionary<String, Photo> photoMap =
                      new Dictionary<string, Photo>();
   photoMap.Add( "pic1.jpg",
                 new Photo { Name = "pic.jpg",
                             Description = "This is pic 1"
                           }
               );
   photoMap.Add( "pic2.jpg",
                 new Photo { Name = "pic.jpg",
                             Description = "This is pic 2"
                           }
               );

   // Place in Application scope
   this.Application["photoMap"] = photoMap;
}
```

Now we are going to add a debugging *generic handler* that will access the `photoMap` dictionary and serialize it to a HTTP response so we can see it in the browser.

Note that a generic handler is not the same as an ASP.NET handler, so make sure you add the right handler.

To serialize, we use the .NET facility to serialize objects to JSON format. This is documented at

```
https://msdn.microsoft.com/en-us/library/system.web.script.
             serialization.javascriptserializer(v=vs.110).aspx
```

which you can reach by googling `MSDN JavascriptSeriliazer`. The example of that page shows that to serialize `myObject`, you simply write

```
var serializer = new JavaScriptSerializer();
var serializedResult = serializer.Serialize(myObject);
```

Putting our new-found knowledge to use, we modify the generic handler we just added like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Script.Serialization;

namespace ASPNETOnlinePhotos
{
    public class DebugHandler : IHttpHandler
    {
        public void ProcessRequest(HttpContext context)
        {
            context.Response.ContentType = "text/plain";

            // Grab the map from Application state
            Dictionary<string, Photo> photoMap =
            context.Application["photoMap"] as Dictionary<String, Photo>;

            // Serialize to JSON+
            var serializer = new JavaScriptSerializer();
            var serializedResult = serializer.Serialize(photoMap);

            // Write to the response
            context.Response.Write(serializedResult);
        }

        public bool IsReusable
        {
            get
            {
                return false;
            }
        }
    }
}
```

6. We still need to write the serialized dictionary to disk file. To do this, we need to use a `StreamWriter` object. The constructor

```
StreamWriter(String path)
```

creates an object that will write to the file with the given path. The `StreamWriter` class is a subclass of `TextWriter`, which provides various `Write` and `WriteLine` methods to write primitive values, strings, and even objects. When writing object, the `ToString()` method of the object is called to return the string to be written.

To write the serialized dictionary to the file, modify the `Application_End` method like this:

```
protected void Application_End(object sender, EventArgs e)
{
   // Grab the dictionary from the Application context
```

```
    // Serialize it, and write the result to a file.

    Dictionary<String, Photo> photoMap
        = this.Application["photoMap"] as Dictionary<String, Photo>;
    var serializer =  new JavaScriptSerializer();
    var serializationResult = serializer.Serialize(photoMap);

    // Write to the file
    var photoFileStream = new StreamWriter
                (@"C:\Temp\Muganda1\photosdata.json");
            photoFileStream.WriteLine(serializationResult);
            photoFileStream.Close();
}
```

The ASP.NET development server used with Visual Studio does not fire the `Application_Start` event when you exit Visual Studio, and there seems to be no way other way end the application. However, if you rebuild the application, the currently deployed version will end before the new version is deployed. Therefore, to see the file written to disk, you need to rebuild the application.

Note the difference between JSON serialization and Java Seriliazation. JSON Serialization is text-based, while Java Seriliazation is binary serialization.

7. You will need the ability to work with files. You may want to check if a file exists, or delete a file, or copy one a source file to a destination where overwriting of the destination file is not permitted, or copy a source file to a destination file with the option of overwriting an exsiting file. The .NET `File` class at

```
https://msdn.microsoft.com/en-us/library/system.io.file(v=vs.110).aspx
```

has methods for doing all these and more:

```
public static bool Exists(string path)

public static void Delete(string path)

public static void Copy(string sourceFileName,
                        string destFileName)

public static void Copy(string sourceFileName,
                        string destFileName,
                        bool overwrite)
```

8. You will need to be able to read a text file when you want to read the serialized map. To do that, use a `StreamReader` object via the constructor

```
StreamReader(String path)
```

This class is a subclass of the `TextReader` class, which counts among its list of methods such useful methods as

```
public virtual string ReadLine()
public virtual string ReadToEnd()
```

The first method reads and returns the next line from the stream, and the second method reads everything remaining in the stream and returns it in the form of a string.

At this point you should check your understanding of these concepts. Modify the `Application_Start}` method so it checks the `Temp` folder to see if the file storing the dictionary exists. If so, it reads the contents of the file as a single string, deserializes the JSON string, and puts the resulting map into application scope. If the file does not exist, it creates an empty dictionary and puts it into application scope. Use the debug handler to verify that your code is working correctly.

9. At this point you are ready to add a generic handler that will do the photo upload:

```
namespace ASPNETOnlinePhotos
{
       public void ProcessRequest(HttpContext context)
       {
            switch(context.Request.HttpMethod.ToUpper())
            {
                case "GET":
                    context.Response.Redirect("PhotoUpload.html");
                    break;

                case "POST":  handlePost(context);
                    break;
            }
        }

        void handlePost(HttpContext context)
        {
        }
        public bool IsReusable
        {
            get
            {
                return false;
            }
        }
    }
}
```

In the case of GET, the handler can use the method

```
public void Redirect(string url)
```

of the `HttpResponse` object to redirect the request to a static HTML page that contains the form for doing the photo upload.

The switch statement case of POST will handle the postback of the photo upload form. To handle the multipart form-data encoding, use the

```
public HttpFileCollection Files { get; }
```

property of the `HttpRequest` object. You can index this collection with the name of input element to get the posted part in the form of a `HttpPostedFile` object.

```
public HttpPostedFile this[string name] { get; }
```

or equivalently, you can use the following method on the `Files` collection:

```
public HttpPostedFile Get(string name)
```

You can then use the properties and methods of the `HttpPostedFile` class, documented at

```
https://msdn.microsoft.com/en-us/library/system.web.httppostedfile
            (v=vs.110).aspx
```

to access the posted file.

The ASP.NET file upload API is a little different from the Java Servlet file upload specification. Only input elements with type attribute of `"file"` give rise to a `HttpPostedFile` object on the server side. To access the description, treat it as a regular form parameter and use `context.Request["description"]`.

10. Add one more handler, to allow users to view uploaded photos.

2. Miscellaneous Hints

The .NET platform has a `StringBuilder` class that may be useful. C# has a `foreach` statement that is very similar to Java's enhanced `for` loop.

C# also has a `String.Format()` method that provides the ability to embed values into format strings. In addition, the `TextWriter` class also has a `Write()` and `WriteLine()` methods that accept a format string and additional values to embed into the string.

I discovered, however, that .NET format strings throw an exception if the format string contains HTML. I am not sure why, and was not able to find a reason online. If you need to embed dynamically generated values into a HTML template, you will have to split the HTML template into parts. These parts serve as partial templates. In writing the response, you will have to alternate the writing of the partial templates with the writing of dynamically-generated data.

3. Due Date

Due Monday of Week 8. Submit the zipped up Visual Studio folder.