

Answer on Question#38498- Programming, C#

1. Pete is the Chief Executive Officer (CEO) of FlyHigh Airlines. Over the years, the organization has expanded its operations to several countries around the world. This has led to a massive growth in the number of day-to-day transactions that the organization needs to manage, such as ticket booking, flight planning, and aircraft maintenance. However, the existing software that the organization uses, to manage its operations, is incapable of scaling up to the increased operational loads. In addition, the software does not have any provision for the management of new facilities that the organization has introduced for its passengers, such as booking of e-tickets. This is hampering the efficiency of the organization.
Therefore, Pete decides to revamp the existing software to enable it to: Scale up to the increased operational needs of the organization. Support new features such as booking of e-tickets. Pete contacts SoftSols Inc., a software development organization, to revamp the software. T

Solution.

Application Auto-update via Online Files in C#



Introduction

If you want to provide your users with a way of updating an application, you have a few different choices, some of which are:

- Get users to manually download and install updates - notifying them by email or relying on them to check online.
- Using ClickOnce.
- Allow the application to check for updates online and either update automatically or present the user with a one click update facility - without using ClickOnce.

This article demonstrates the last method in the above list - providing your applications with a one click update or an automatic update facility without using ClickOnce.

The update is done through a class and application which installs online updates and is illustrated through the **UpdateMe** application (included as a download with this article).

Without too much difficulty, you should be able to use the class and **update** application, as shown via the **UpdateMe** application, to get automated updates out to the users of your software.

I have kept the **UpdateMe** application as simple as possible so that you will be able to make use of the code without needing to spend too much time in sifting through what will be relevant for your own application.

To make use of the **UpdateMe** application, you will need to have the ability to store files online(read the next line though).

Note: At the time of this article first being published, I have placed the relevant files (version and update) in a public *Dropbox* folder, pointed to by the default values in the **UpdateMe** textboxes- all things being impermanent, this folder may not be available for long so I suggest you use your own webspace.

Tip: One advantage of using the class, methods and application described and included in this article is that you can store the version information and download files on a free file sharing site, such as **Dropbox**, and not have to spend a penny on hosting the update information or download yourself.

Please see the *readme.txt* document included in the download for how to set up the version and download file in your own webspace.

Background

This code originated from the TeboCam webcam security application I have written which is soon to be published as open source.

The implementation takes form in the following three key components:

- A version file that resides online - this is used by the **update** class to check for online updates and also contains information as to the location of the online updates.
- A class - **update** - containing the methods for querying the online version file. For this article, the class is wrapped within the example **UpdateMe** application.
- An application - **update** - which downloads the updates from an online folder and which restarts a named application (**UpdateMe** in this case).

This article is principally about the **update** class and its use within the example **UpdateMe** application.

The order of processing is as follows:

- The main application (in this case **UpdateMe**) downloads the version file and inspects it to see if an update is available.
- If an update is available, the **update** application is called passing it the location and name of the update file as well as an optional command line and the application name to start after the update is completed. The main application then closes.
- The **update** application ensures that the main application is not running and terminates it if it is, then downloads the file and starts the application specified with the optional command line parameters.

It really is that simple (he says after spending the weekend tidying up the code...).

Initially, when I wrote this code, someone asked "**How do you update the update application?**" - so back to the code I went - and the solution turned out to be quite simple:

Any files relating to the **update** application are saved online with a prefix – the **update** class contains a method that simply renames these files to overwrite the **update** application.

So with one button, it is possible to update your main application as well as the **update** application itself.

There is more information relating to the method that handles this lower down in the article.

There are two modes to the update:

- You can download the file immediately – this is the method we use to download the version file to check if any updates are available online.
- You can close the program for an application update.

This application makes use of the **lonic zip** open source library for unzipping files.

I also make use of the methods, etc. that I have found through Google – if you find anything for which I need to give credit to others, for please do let me know.

Using the Code

The **update** class contains the methods for interrogating the version file.

The version file, which you will need to place online, takes the format of a pipe delimited file.

The order of processing is:

- Check if new **update** application files exist, with the defined prefix in their names, and rename these files if they exist – this enables us to update the **update** application.
- Get the update information from the online version file.
We then have a choice if the version file contains information pointing to a new file:
- Download the file without restarting the application, or
- Call the **update** application to download the file.

Getting the update availability information: The **getUpdateInfo** method downloads a pipe delimited file, splitting the columns, into a **List** object.

One point to note is the line number to start reading the data from – this is a zero based index meaning the first line is 0.

In the example below, we start reading from the second line - I prefer to have a header line as when I amend the version file I can know by looking at the file which information goes where.

```
info = update.getUpdateInfo(downloadsurl.Text, versionfilename.Text,
    Application.StartupPath + @"\", 1);
```

As the result of reading the version file is a **List** object, returned by the **getUpdateInfo** method, you can place whatever you need to in this file for your update information.

I suggest at first to keep it simple with the version, download URL and download filename being the only vital pieces of information required for an update.

```
app|version|release date|url|file
updateme|1.2|24/09/2011|http://changeThisUrl.com/downloadFiles/|updateme.zip
```

One thing to note with keeping the information, in this version file online, is that it allows you to direct users' applications to pick up the new application from whichever location you desire.

Tip - When running the **UpdateMe** application, make changes to the "This Version No." textbox to test if the update is picked up.

With regards to installing the updates, two methods exist:

- **installUpdateNow** - This will install the downloaded file(s) without the application restarting, and
- **installUpdateRestart** - This will install the downloaded file(s) via the update application and results in the application restarting.

The **installUpdateRestart** method is the business end of the **UpdateMe** application - installing an update and restarting the new version of the **UpdateMe** application through the **update** application.

This method starts the **update** application - passing parameters indicating which file to download and which process to start once the download of the file has completed.

```
public static void installUpdates(string downloadsURL, string filename,
    string destinationFolder, string processToEnd, string postProcess,
    string startupCommand, string updater)
{
    string cmdLn = "";

    cmdLn += "|downloadFile|" + filename;
    cmdLn += "|URL|" + downloadsURL;
    cmdLn += "|destinationFolder|" + destinationFolder;
    cmdLn += "|processToEnd|" + processToEnd;
    cmdLn += "|postProcess|" + postProcess;
    cmdLn += "|command|" + @" / " + startupCommand;

    ProcessStartInfo startInfo = new ProcessStartInfo();
    startInfo.FileName = updater;
    startInfo.Arguments = cmdLn;
    Process.Start(startInfo);
}
```

The **webdata** class is used by both the **UpdateMe** application and the **update** application. This class contains the methods for downloading data from the online location.

One area I think worth pointing out is the use of the custom **bytesDownloaded** event. If you are fairly new to .NET, then custom events are something which I would strongly recommend using and understanding (that is understanding how to use them and why they are so handy).

To enable the progressbar in the **update** application to correctly display how far through the data download we are - we will need to be able to specify the bytes downloaded together with the total number of bytes in the download.

We start by defining a delegate and a class that holds the arguments for the custom event - the class **ByteArgs** inherits from the **EventArgs** class.

 [Collapse](#) | [Copy Code](#)

```
public delegate void bytesDownloaded(ByteArgs e);

public class ByteArgs : EventArgs
{
    private int _downloaded;
    private int _total;

    public int downloaded
    {
        get
        {
            return _downloaded;
        }
        set
        {
            _downloaded = value;
        }
    }

    public int total
    {
        get
        {
            return _total;
        }
        set
        {
            _total = value;
        }
    }
}
```

When we start downloading the data, we create a new instance of the **ByteArgs** class and assign the downloaded and total values.

One point to note when we call the event - is that we test if it is **null** first - if we do not test if the event is **null** and the event is not consumed, we will get a **NullReferenceException**. This is important because the **update** application consumes the **bytesDownloaded** event for the purpose of the progressbar whilst the **UpdateMe** application does not.

 [Collapse](#) | [Copy Code](#)

```
//Let us declare our downloaded bytes event args
ByteArgs byteArgs = new ByteArgs();

byteArgs.downloaded = 0;
```

```
byteArgs.total = dataLength;  
  
//we need to test for a null as if an event is not consumed, we will get an exception  
if (bytesDownloaded != null) bytesDownloaded(byteArgs);
```

Earlier in this article, I mentioned the question - "**How do you update the update application?**" A method within the **Update** class called **updateMe** resolves this issue:

 [Collapse](#) | [Copy Code](#)

```
update.updateMe(updaterPrefix, Application.StartupPath + @"\"");
```

We simply pass the prefix of the **update** application files to this method together with their location - the prefix allows us to download these files while the **update** application is running without running into any file sharing/lock issues.

Calling this method when **UpdateMe** starts up allows us to install any new files related to the **update** application.

Tip: you will notice that the *um.zip* file included as a download with this article has files with the **M1234_** prefix - these files will have their prefix removed and will replace or supplement any files in the application folder once the update has been published when running the **UpdateMe** application.

Points of Interest

Initially, I used an HTML webpage to hold the update information – the code would inspect this page for updates.

This was causing problems at times with connections failing - so I decided to place the update information in a pipe delimited online file that can be downloaded, for some reason the file download option was more reliable – 103 bytes of download (the version file should be around this size) is not going to use up too much bandwidth.

One area that is not covered by this article is the issue of cleaning up legacy files from the application folder - this can potentially be handled when the **updateMe** method runs in your main application (what I mean by a legacy file is a file which was used by a previous version of your application and is no longer of use).

Alternatively you can leave legacy files on the users' machines - I admit this is not ideal however with the risk of removing files the user deliberately placed in the application folder it may be better to allow these legacy files to persist.

I really do hope that this will be of help to others and let me know what can be improved – please.