

Mobile Updater Application Block

Reference Guide

November 2006



Table of Contents

•	Getting Started	3
1	Welcome to the Mobile Updater Application Block	3
2	Importance of the updating process in mobile applications.	3
3	How it can help you	4
4	System Requirements	4
5	Mobile Updater Application Block Contents	4
6	More information	5
•	Mobile Updater Design	5
1	How the updating process works	7
•	Understanding Mobile Updater Application Block Manifest	9
•	Extensibility	10
•	Deployment	10
•	Mobile Updater Developer's Reference	12
1	Server Side	12
2	Client Side	16

Getting Started

1 Welcome to the Mobile Updater Application Block

Welcome to the Mobile Updater Application Block! It has been developed by the Q4Tech Engineering Team based on the Microsoft Patterns & Practices Updater Application Block and uses the Connection Monitor and the Configuration Application Block, both part of the Mobile Client Software Factory.

Continuing with the well work done by Patterns & Practices group with the Mobile Client Software Factory, Q4Tech has decided to address a big pending problem in the mobile application development: the application deployment and maintenance.

This application block is not just a port of the desktop version. It includes specific functionality regarding mobile scenario needs. We're cleaned up the code trying to keep it as small as possible and included an Updater Agent in the block which works as a service in the mobile device, letting you to manage the deployment process for your applications from a centralized server.

The design of the application block provides a manageable and scalable solution that strikes a balance between flexibility and complexity. The main design goals of the Mobile Updater Application Block are the following:

- Simplify the addition of self-updating capabilities to a mobile application using standard protocols like TCP/IP and HTTP
- Adapt and extend Updater Application Block 2.0 from Patterns and Practices to the mobile scenario
- Efficient use of network bandwidth
- Processing of complex updates
- Incorporating mobile device status variables to the updating process
- Support background updating for mobile applications through an Updater Agent

2 Importance of the updating process in mobile applications.

The deployment and updating process is a very important issue in mobile application development. During the development of a mobile application you can find not only different platforms to implement, you can also find a geographically huge distribution of clients and update those clients with bug-fixes and new releases can be extremely complex.

There are several issues to address talking about updating processes. You need to identify what are the files to deploy for each device, you need to identify when the device needs to download a new version for an specific application, and you also need to know if it's safe to start the update installation in the device based on the battery level or storage free space. Additionally you can have a running application which needs to complete some tasks (because it can be performing critical business tasks) before to be updated.

This block is intended to address all this problems. Our main focus was the client side of the updating process. For the client side the block includes basic functionality and a web service contract which allows you to add business specific functionality based on a device Id coming from the device.

As his desktop antecessor, the Mobile Updater Application Block is designed to support the most common scenarios for self-updating applications:

- ✚ It supports applications that need to be kept up to date with current executable versions.
- ✚ It supports applications that use multiple plug-ins that users can download and activate on their desktops.
- ✚ It supports applications that rely on reference data (for example, a large set of documents) that need to be cached on the client and occasionally updated from a server.

3 How it can help you

The Mobile Updater Application Block helps you in the following ways:

- ✚ It helps you implement a “pull model” for automatically download updates for .NET Compact Framework applications.
- ✚ It helps you perform post-download configuration tasks without requiring user intervention.
- ✚ It helps you perform pre-download and pre-configuration tasks based on device status queries.
- ✚ It helps you drive the download and configuration tasks from your application if it's already running avoiding data loosing.

4 System Requirements

The Mobile Updater Application Block requires the following software:

- ✚ Microsoft Visual Studio 2005
- ✚ Microsoft Windows Mobile 5.0 SDK for PocketPC
- ✚ Microsoft Windows Mobile 5.0 SDK for Smartphone
- ✚ Patterns & Practices Mobile Client Software Factory

5 Mobile Updater Application Block Contents

With this release you can find the following contents:

- ✚ **The Mobile Updater Application Block source code.** You'll find the source code for the Mobile Updater Application Block, the Updater Agent and a basic implementation of the **ManifestListProvider** web service, called SampleWebService. This web service is needed to test any of the included sample applications.
- ✚ **Documentation.** There are two documents included. This is the reference Guide, written for giving a basic overview about the updater design. There is also a Quick Start guide, with step-by-step instructions to run the included reference implementations.
- ✚ **Reference implementations.** This release includes three reference implementations according with the following scenarios:
 - ✚ Two versions of a Windows Mobile 5 Pocket PC application using the updater agent and the basic manifest list provider. This reference implementation includes how to configure the agent and how to handle update events from an

updatable application. Each version comes with a CAB Installer project to generate the corresponding setup package.

- ✚ Two versions of a Windows Mobile 5 Smartphone application using the updater agent and the basic manifest list provider. This reference implementation includes how to configure the agent and how to handle update events from an updatable application. Each version comes with a CAB Installer project to generate the corresponding setup package.
- ✚ A CAB sample application supporting update events handling from the updater agent through Event Broker.
- ✚ **Mobile Updater Agent** with support for **Windows Mobile 5.0** devices including **PocketPC** and **Smartphone**.
- ✚ Patterns & Practices Mobile Client Software Factory used application blocks:
 - ✚ Configuration Application Block
 - ✚ Connection Monitor

6 More information

In order to get a more information and support about the Mobile Updater Application Block, you can see:

- ✚ MobileBlocks project @ CodePlex
 - ✚ <http://www.codeplex.com/Wiki/View.aspx?ProjectName=MobileBlocks>
- ✚ Q4Tech website: <http://www.q4tech.com/>
- ✚ Mobile Updater Design section (following section in this document)
- ✚ Quick starts document
- ✚ Reference implementations
- ✚ Developer Reference section
- ✚ Microsoft Patterns & Practices Updater Application Block version 2.0. (<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/updaterv2.asp>).

Mobile Updater Design

The Mobile Updater Application Block is designed to provide a "no touch" solution to the problem of keeping mobile applications up to date in enterprise environments. You can use it out-of-the-box to perform the downloading and activation of updates for your applications. It is also designed to be extensible, so you can customize the default application block to perform exactly the functionality that your application requires, whether that is using a different type of download technology or performing a complex set of tasks after the download process is complete.

The mobile application block, which resides at client side, consists of four subsystems, each designed to fulfill a specific role in the application update process. These subsystems include the following:

- ✚ **Update management subsystem.** The update management subsystem consists of the **ApplicationUpdaterManager**, **RegistryManager**, and **UpdaterTask** classes. **ApplicationUpdaterManager** is a facade into the Updater Application

Block. This is a singleton and is responsible for interacting with the **ManifestManager** to determine when the updates are available. It also provides methods to the client application and the agent to start the download and activation processes.

- ✚ **Manifest management subsystem.** The manifest is configuration information for the update to be applied to the client application. It is stored on the server; the information describes the updates that are available and the configuration of those updates. It also defines how the update should be downloaded, what are the conditions required to download and activate it, and the activation processors that should be executed after the files are transferred. Manifest classes access this information and make it available to the other classes in the application block.
- ✚ **Downloader subsystem.** This subsystem is responsible for connecting to the server and downloading the updates to the client computer.
- ✚ **Activation subsystem.** This subsystem is responsible for performing any activation processing, for example, copying the downloaded files to a specified location, deleting temporary files and folders, or executing a .cab file installation process.

Your auto-updating mobile application can use the Mobile Updater Application Block in-process to get the application update stored at server side in the Manifest XML File.

An alternative model is to centralize the update logic in the Updater Agent. This approach is highly recommendable if you have several updatable applications in the device and instead of adding auto-updating logic in each application, you can leave that logic as responsibility of the Updater Agent. Following this approach, the Updater Agent will get a manifest list from the server through a web service (which can be modified according with your own logic) and check for updates regarding that manifest list. Due to the Agent can be running in background, and probably it can need to update an application which is currently running, you can subscribe your application to the Updater Agent Notification Subsystem in order to handle updating events (like new updates found or new updates downloaded). This procedure lets you decide if the update can be activated, or even downloaded.

The Updater Agent consists of four components. It uses the configuration application block and the connection monitor from the Mobile Client Software Factory, and uses the Mobile Updater Application Block to get the updates and activate them. Those four components are:

- ✚ **Device id subsystem.** This subsystem is responsible for getting the device id to be supplied as a parameter to the manifest list provider web service in order to get the manifest list for one specific device. The device id can be a configurable id in the app.config file for the updater agent or the hardware device id. You can select what provider to use in your configuration file.
- ✚ **Updater Configuration Subsystem.** This subsystem is responsible for setup the application block to start the updating process. It provides the corresponding downloader and default configuration to the mobile application block.
- ✚ **Updater Agent Management Subsystem.** This subsystem is responsible for controlling the agent behavior. It monitors connectivity status through the **Connection Monitor**, gets the manifest list, starts the updating process and notifies running applications for updating events through the **NotificationManager** using the **Updater Agent Common Notification Subsystem**.
- ✚ **Updater Agent UI.** Additionally, the updater agent provides a user interface which allows the user to monitor the updater status, force check for updates, and

shutdown the agent. This UI is implemented using the model-view-presenter pattern.

Figure 1 shows the high level design of the Mobile Updater Application Block.

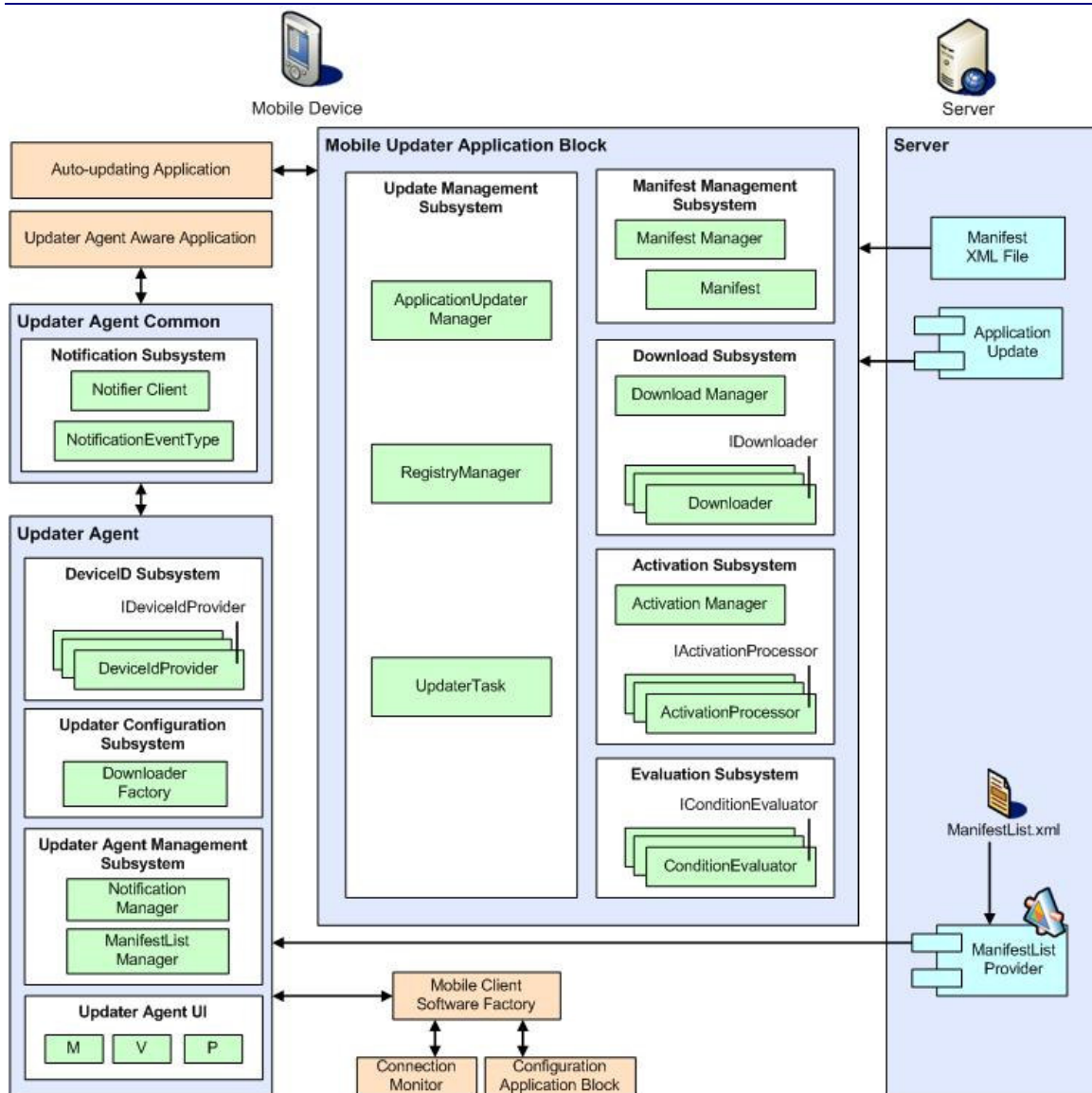


Figure 1. High level design of the Mobile Updater Application Block

1 How the updating process works

The simplest use case of the Mobile Updater Application Block consists of the following steps:

- ✎ The client application requests the **ApplicationUpdaterManager** to check if any updates are available.
- ✎ The **ApplicationUpdaterManager** uses the **ManifestManager** to retrieve the XML manifest for this application from a specified server location.
- ✎ If updates are available, the **ApplicationUpdaterManager** calls the **DownloadManager**, which uses the **HTTPDownloader** to transfer the files from server to client.
- ✎ After all the relevant files are downloaded, the **ActivationManager** uses the specified **ActivationProcessors** to activate the files.

The simplest use case of the application block using the **Updater Agent** consists of the following steps:

- ✎ The **Updater Agent** requests the **ManifestListManager** to retrieve the manifest list.
- ✎ The **ManifestListManager** requests the manifest list from the **ManifestList Provider** web service.
- ✎ The **ManifestListProvider** loads the **ManifestList.xml** file and returns it as XmlDocument containing a server location for each manifest.
- ✎ The **ManifestListManager** requests the **ApplicationUpdaterManager** to check if any updates are available for each manifest in the manifest list.
- ✎ The **ApplicationUpdaterManager** uses the **ManifestManager** to retrieve the XML manifest for each application from a specified server location.
- ✎ If updates are available, the **ApplicationUpdaterManager** calls the **DownloadManager**, which uses the **HTTPDownloader** to transfer the files from server to client.
- ✎ After all the relevant files are downloaded, the **ActivationManager** uses the specified **ActivationProcessors** to activate the files.

Understanding Mobile Updater Application Block Manifest

The manifest

Following you can see a sample manifest file:

```
<?xml version="1.0" ?>
<manifest xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  manifestId="{D56F0089-385C-4c7c-BEE4-121C55900C2B}"
  mandatory="False"
  xmlns="urn:schemas-microsoft-com:PAG:updater-application-
  block:v2:manifest">
  <description>Sample Application - Version 1</description>
  <application applicationId="{ED83BE04-9ACE-4a72-A1B1-2C06EF92E699}">
    <entryPoint file="SampleApplication.exe" parameters="" />
    <location>\Program Files\Sample Application</location>
  </application>
  <files base="http://ppp_peer/SampleWebService/Applications/{ED83BE04-
  9ACE-4a72-A1B1-2C06EF92E699}/" hashComparison="No">
    <file source="SampleApp.cab" transient="False" />
  </files>
  <activation>
    <tasks>
      <task
        type="Microsoft.ApplicationBlocks.Updater.ActivationProcessors.CABUpdateProce
        ssor, Microsoft.ApplicationBlocks.UpdaterCF" name="ApplicationDeployProcessor"
        />
    </tasks>
  </activation>
  <preDownloadConditions>
    <conditions>
      <condition
        type="Microsoft.ApplicationBlocks.Updater.Evaluators.AvailableStorage,
        Microsoft.ApplicationBlocks.UpdaterCF" name="AvailableStorage" space="300K"
        storageCard="false" />
    </conditions>
  </preDownloadConditions>
  <installConditions>
    <conditions>
      <condition
        type="Microsoft.ApplicationBlocks.Updater.Evaluators.BatteryLevel,
        Microsoft.ApplicationBlocks.UpdaterCF" name="BatteryLevel" minLevel="40"
        overrideLevelOnAcPower="true" />
    </conditions>
  </installConditions>
</manifest>
```

The manifest is a configuration file for application updates. It contains identifying information for the application to be updated, the file list, the downloader which should the updater use to get the files and the activation tasks.

The Mobile Updater includes two new supported sections in the manifest: the **preDownloadConditions** section and the **installConditions** section. Both sections are composed by evaluators. An evaluator is a new concept which allows you to evaluate some specific device status indicator, as the available storage or battery level. Each

evaluator provides its own parameter list. Using evaluators you can specify in your manifest that you don't want to download the updates for this application if the battery level is less than 40%, or you don't want to activate the manifest if there are less than "2 M" of available storage space in your device.

If the updater doesn't complete the download or the install because one evaluator is not accomplished, it'll still running the evaluators on every check for updates process and when all the evaluators for the manifest are accomplished it will continue with the suspended process.

Extensibility

You can extend the updater implementing your own downloader, activation processor or even your own condition evaluator.

Downloaders

The current version of the Mobile Updater includes one downloader: the **HTTPDownloader**. It allows you to download any file in the You can extent it adding a new downloader just implementing the **IDownloader** interface according with your own needs.

ActivationProcessors

You can also extend the Mobile Updater adding new activation processors implementing the **IActivationProcessor** interface and including the new processors in your manifest.

Device Status Conditions

This version includes two built-in condition evaluators: **AvailableStorage** and **BatteryLevel**. You can implement your own device status condition evaluators implementing the **IConditionEvaluator** interface. A possible custom condition evaluator to be implemented depending on your deployment scenario can be a connectivity evaluator, if you don't want to apply the updates if the device is disconnected.

Deployment

You can deploy the Mobile Updater Application Block on mobile devices that are running Windows Mobile 5.0. Before deploying the Mobile Updater Application Block, you must install the Microsoft Compact Framework version 2.0.

You can deploy the whole Mobile Updater Application Block (including the source code, documentation, and Quick Starts) on a development workstation by running the MobileUpdater.msi installation program.

In production environments, you should use the following deployment method:

Preparing the Shared Update Location

Create a central server-based location application updates can be downloaded from. The specific kind of location (such as Web server or network share) depends on the downloader you intend to use. The Mobile Updater includes only one default HTTPDownloader. In order to use that downloader, your update location should be on a Web server. You can create a single location for all application updates or use a

different location for each application. If you are publishing updates for multiple applications in a single location, you should generally use a subfolder for each application.

For each update, you must create a folder that contains the manifest file and the application updates. A good practice is to name the root folder after the application name and create a subfolder for the application updates named after the version number. For example, the root folder for MyApp should be named MyApp and the application updates for version 2.0.0.0 should be in a subfolder named 2.0.0.0.

Figure 2 shows a server deployment for two updateable applications.

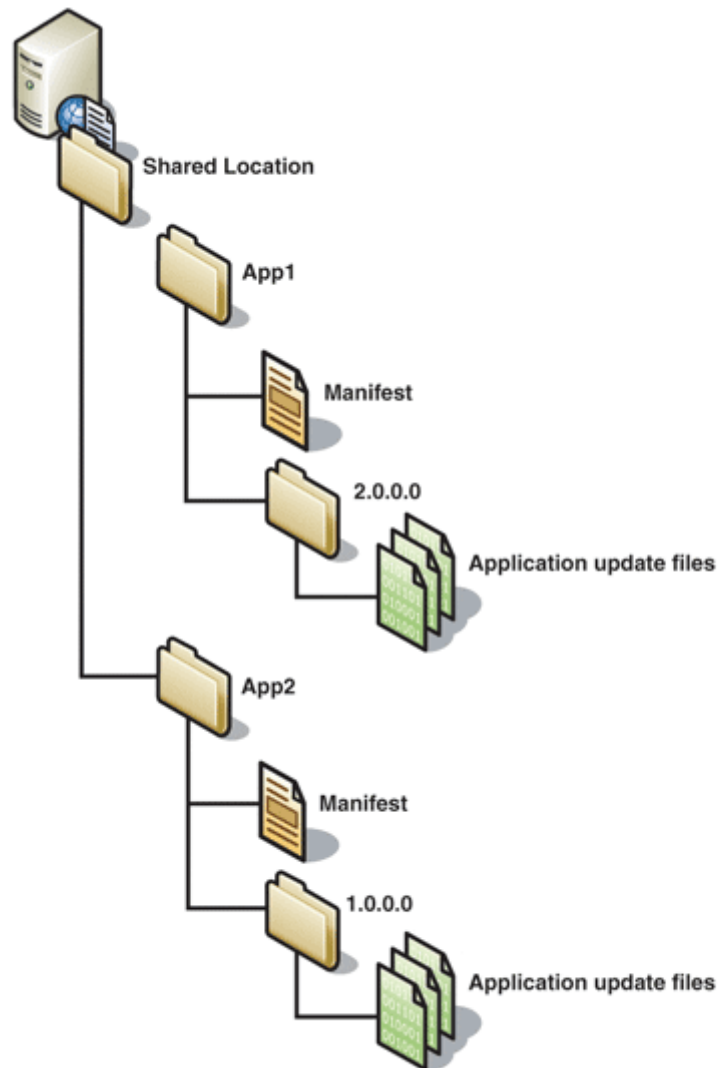


Figure 2. Server deployment

👉 Deploying an auto-updating client application

If your application is not using the Mobile Updater Agent, you'll need to deploy the following files with it:

- 👉 q4tech.engineering.ipc.dll
- 👉 q4tech.engineering.updateragent.common.dll

👉 Deploying the Mobile Updater Agent and Agent-aware applications

If your application is using the Mobile Updater Agent, you'll need to deploy the following files with it:

Additionally, the device should have the Mobile Updater Agent already installed, or you can deploy it.

If you want to install the Mobile Updater Agent you'll need to deploy the following files:

- 👉 microsoft.applicationblocks.mobile.updater.dll
- 👉 microsoft.practices.mobile.configuration.dll
- 👉 microsoft.practices.mobile.connectionmonitor.dll
- 👉 q4tech.engineering.ipc.dll
- 👉 q4tech.engineering.updateragent.common.dll
- 👉 UpdaterAgent.exe
- 👉 app.config

Mobile Updater Developer's Reference

1 Server Side

How to implement server side logic

👉 Server-side Responsibilities

The server-side user application will be responsible of delivering a manifest list tailored for each device.

The current implementation is made through a Web Service, although minor modifications could be made to the client block, so other communication protocols could be used, as explained below.

Specifically, this single-operation web service receives a unique device ID and has to render an XML document as a response in concordance with the XSD schema specified in ManifestList.XSD.

The uniqueness of the device ID is given by the implementation of the client's **device id provider** (see below for details), so the ID could be unique in an absolute sense, i.e. every device available in the market, or local inside an organization or a given computer system.

👉 The application list:

👉 XSD description

The provided XSD is the schema for the manifest list that the server application should return. It can be extended to suit whatever modification was necessary,

adding attributes or even entire elements and groups of elements. Figure 3 shows the application list XSD.

NOTE: The schema is provided as a guide only. No strict validation occurs in the whole process.

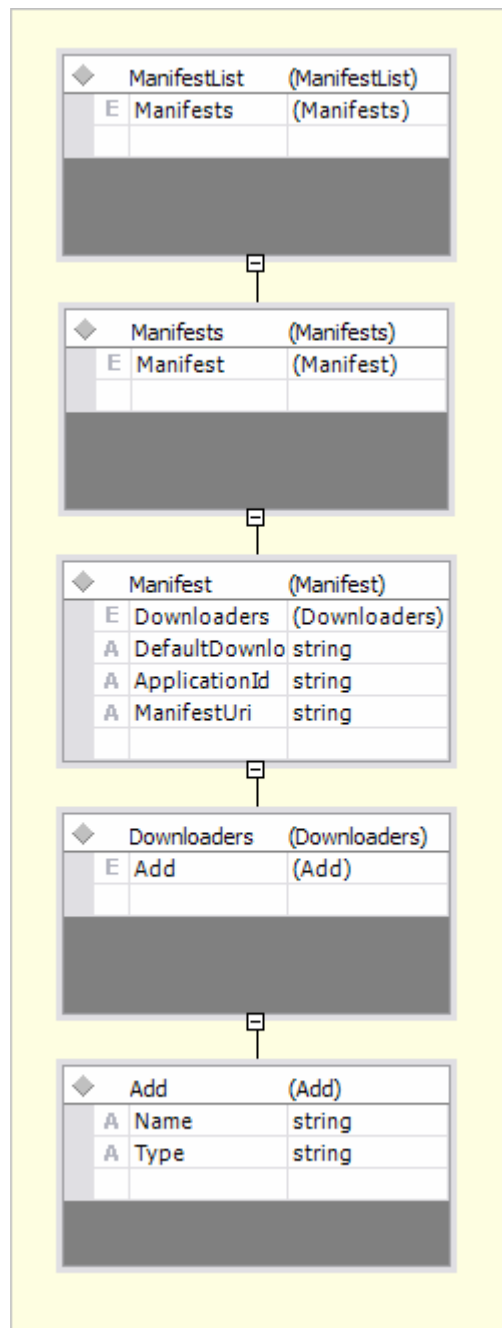


Figure 3. XSD

The **Root element**, named **ManifestList** contains no attributes, and a single, required appearance of a **Manifests** element.

The **Manifests element** contains no attributes and may contain zero or more (0-unbounded) **Manifest** elements.

Each **Manifest element** represents a current update or version for a given application, and should contain the following attributes:

ApplicationId: Is a string specifying the application ID

ManifestUri: Is the URI the updater is going to use for retrieving the manifest for the application. This URI should be accessible through HTTP

DefaultDownloader: Is a string with the preferred downloader for downloading application binaries. It must be one of the items appearing in the **Downloaders** element for this attribute's element

Also the element must contain one **Downloaders** element.

The **Downloaders element** has no attributes and must contain one or more (1-unbounded) **Add** elements.

Each **Add element** represents a downloader. It has no nested elements and should specify these attributes:

Name: A name for the downloader, for matching with the **DefaultDownloader** attribute

Type: The full type (including the assembly if necessary) for loading the component

👉 Web service signature

As part of the source code, there is the formal WSDL definition for the web service signature.

The following is the operation body, as implemented in the sample web service:

```
[WebMethod]
public XmlDocument GetManifestList(String deviceId)
{
    XmlDocument document = new XmlDocument();
    document.Load(Server.MapPath("ManifestList.xml"));

    return document;
}
```

And here, a sample SOAP 1.2 call:

```
POST /samplewebservice/Service.asmx HTTP/1.1
Host: localhost
Content-Type: application/soap+xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap12:Envelope xmlns:xsi="http://www.w3.org/2001/...
  <soap12:Body>
    <GetManifestList xmlns="http://tempuri.org/">
```

```
<deviceId>string</deviceId>
</GetManifestList>
</soap12:Body>
</soap12:Envelope>
```

👉 How to extend the web service signature

In case you need to add extra functionality to the manifest list web service, as additional validations or more complex identification logic, you'll have to change the web service reference signature. After changing the reference, you may need to modify these lines, at **ManifestListManager.GetManifestList()**:

```
...

XmlNode doc = service.GetManifestList(deviceId);

XmlSerializer serializer =
    new XmlSerializer(typeof(XSD.ManifestList));

return (XSD.ManifestList)serializer.Deserialize(
    new StringReader(doc.OuterXml));
}
```

The first line actually calls the webservice, sending the device ID and retrieving an XML document with the response. If you change the webservice signature, this line has to be modified to suit the new signature.

Later, depending if you change the manifest list schema or not, you may need to modify the next two lines which parse the response.

👉 Kicking off the web service

The manifest list web service can be removed easily and replaced with a variety of methods. The following are only two ideas to work around a better suited solution.

👉 Using HTTP Get for the application list.

The following code retrieves the manifest list directly from a URL, bypassing the web service SOAP enveloping. The example uses a fixed URL address, but you can manage your way to add querystring parameters for a non anonymous fetching.

```
WebRequest req = WebRequest.Create("http://fooURL/list.xml");

WebResponse response = req.GetResponse();

Stream stream = response.GetResponseStream();

XmlDocument doc = new XmlDocument();

doc.Load(stream);

XmlSerializer serializer = new
XmlSerializer(typeof(ManifestList));
```

```
return (ManifestList)serializer.Deserialize(new  
StringReader(doc.OuterXml));
```

✎ Using FTP

You can get the manifest xml list through FTP using third-party components due lack of FTP protocol support in CF. Next to getting the manifest, the parsing will be the same as the previous case.

✎ Publishing manifests

Manifest has to be accessible using HTTP protocol at the same time the manifest list is downloaded and later. There is no support for web service calling, but only plain HTTP. The simpler option for sending data and validating is passing the parameters as querystring elements specified in the manifest list.

✎ Server side for an auto-updating application (no Updater Agent usage at client side).

There is no formal support for sending data when retrieving a manifest. The simpler choice is to include querystring variables in the manifest URI. On the server side, a web site parsing these variables can be the responsible for verifying device identity and other required tasks.

2 Client Side

✎ Adding auto-updating support to your application

The following steps conforms the guidelines to using the updater as an in-process block, either embedded on the main application or in a stand-alone custom updating application.

✎ Referencing the updater from your application

First, you have to add a **reference to the assembly UpdaterCF.dll** in every project that will perform an updating operation. These could be the main application or the updating application (depending on your update model) and any other extension component you develop to perform custom activation task or alternative download methods, for example.

Next, you have to **initialize the updater** component calling the static **Initialize()** method in the **ApplicationUpdateManager** class .

```
public static void Initialize(string basePath,  
IDownloaderFactory downloaderFactory)  
{ ...
```

The first parameter, **basePath** is the base path the updater will use for storing metadata files and temporary binaries.

In **downloaderFactory** you have to supply an **IDownloaderFactory** instance. This is a class that you have to make for you own, and is a simple interface with two methods, as follows:


```
public interface IDownloaderFactory
{
    IDownloader CreateDownloader(string downloaderName);
    IDownloader CreateDownloader();    }
```

The first override, **CreateDownloader(string downloaderName)** has to return a **IDownloader** instance based on the supplied name. The second override simply returns any **IDownloader**, becoming that, the default downloader.

You can copy in your project the simple implementation performed in the Updater Agent. This implementation makes use of the p&p's configuration application block for retrieving the list of downloaders from a config file.

Currently the updater comes with only one **IDownloader** implementation, the **HTTPDownloader** class. If you are going to use this class only, you can make a very simple factory as follows:

```
using System;
using Microsoft.ApplicationBlocks.Updater;
using Microsoft.ApplicationBlocks.Updater.Downloaders;
using Microsoft.ApplicationBlocks.Updater.Utilities;

namespace myproject
{
    public class SimpleFactory : IDownloaderFactory
    {
        public IDownloader CreateDownloader(string
downloaderName)
        {
            return new HTTPDownloader();
        }

        public IDownloader CreateDownloader()
        {
            return new HTTPDownloader();
        }
    }
}
```

As you can see, these implementation's functions returns always a new instance of **HTTPDownloader**, whatever the parameters are.

👉 Getting the manifest

To check for updates for your application you have first to get the **ApplicationUpdaterManager** instance for your application using an application ID. This has to be done for every application you intend to update. Call the **GetUpdater(String applicationId)** static method for this.

```
ApplicationUpdaterManager manager =  
ApplicationUpdaterManager.GetUpdater("inProcessUpdatingApp");
```

You have to use an application ID even if you update a single application, and the **ID you provide must match the one in the update manifests**.

Now you can make the manifest download calling in your instance of the **ApplicationUpdateManager** class the **CheckForUpdates()** method:

```
Manifest[] manifests = manager.CheckForUpdates(new  
Uri("http://www.q4tech.com/inProcessUpdater/manifest.xml"));
```

This will give you an array of manifests as a result. This is due the updater block check for dependant manifests and downloads those too.

👉 Downloading an update

You have to confirm each manifest, setting the **Apply** property to **true**. If there is nothing you wish to check, simply mark all the manifests in a loop:

```
foreach (Manifest manifest in manifests)  
{  
    manifest.Apply = true;  
}
```

The downloading process is asynchronous, so you have to subscribe to a notification event that will indicate when the download is done, and call the **BeginDownload(...)** method to start.

```
manager.BeginDownload(manifests);  
  
// Use async so that we can continue updating other apps.  
manager.DownloadCompleted += onDownloadCompleted;
```

In the example the download will begin immediately and a function of the calling class, **onDownloadCompleted()**, will be called upon completion. When all the files had been downloaded, you must proceed to activation.

👉 Activating an update

You can activate an update at any time after the binary download finishes, but it is usual that this job is done right after the binary fetching process. To begin the activation you must call the **Activate(...)** method in the **ApplicationUpdateManager** class with one or more manifest to activate.

The activation process is asynchronous, so if you want to get feedback about the activation finalization, you'll have to subscribe to the **ActivationCompleted** event.

```
manager.ActivationCompleted += onActivationCompleted;
```

The following sample code is the event handler for the **DownloadCompleted** event.

```
private void onDownloadCompleted(object sender,  
ManifestEventArgs args)  
{  
    if (args.Manifest.Apply)
```

```
((ApplicationUpdaterManager) sender).Activate(new
Manifest[] { args.Manifest });
}
```

👉 Resuming updates

When, under any circumstance, an update is interrupted, the updating state is persisted for further processing. This processing occurs in the next check for updates, and is notified to the application through an event, **PendingUpdatesDetected**.

A classic example of the event subscription and handler is:

```
manager.PendingUpdatesDetected += onPendingUpdatesDetected;

void onPendingUpdatesDetected(object sender,
PendingUpdatesDetectedEventArgs e)
{
((ApplicationUpdaterManager) sender).ResumePendingUpdates();
}
```

The **ResumePendingUpdates** method must be called in order to continue the suspended task.

👉 Giving feedback to the user

As almost all of the processing is asynchronous, your app has to subscribe to various events to get feedback about the updating progress.

In the previous topics there is a description of several events and their usage, and this is the full event list:

- PendingUpdatesDetected

When, for any reason, a previous updating task has been suspended, this event will get raised when an updating process were performed again. See **Resuming Updates**.

- UpdatesSuspended

This event gets fired when at least one condition doesn't meet the expected value at the pre-download evaluation. See **conditions** for more details.

- DownloadStarted

This event gets fired for every file that's being download, previously to the download itself starts.

- DownloadProgress

This event depends heavily on the chosen **IDownload** implementation. Usually the event handler receives information periodically through this event about download progress for each file and in general.

- DownloadError

This event gets fired in the occasion of a download error.

- DownloadCompleted

When all the files where downloaded for a manifest, this event's subscribers receive this notification. See **Downloading An Update**.

- ActivationInitializing

This event gets fired when the activation initialization process occurs. This happens only once per manifest, no matter how much activation tasks a manifest have.

- ActivationInitializationAborted

In the case any activation initialization task error happens, this event will get raised and the whole activation initialization process for the manifest will be aborted.

- ActivationStarted

This event gets fired after the activation initializing phase finishes. If you subscribe to this event, you will have to return a Boolean value in the event handler. If your handler returns **true**, the process continues normally. If **false**, the activation is cancelled.

- ActivationCompleted

This event will be fired after the activation process finishes, either with or without errors. This condition will be indicated using the Boolean **success** parameter.

- ActivationError

In the case an activation error happens, this event will get raised and the activation process will be interrupted. Also, the **OnError()** method of all activation processors in the chain will be executed.

- ActivationSuspended

This event gets fired when at least one condition doesn't meet the expected value at the pre-download evaluation. See **conditions** for more details.

🔧 Using the Updater Agent

You can use the out-of-the-box Updater Agent application to add update support to your applications. The minimum amount of work to do is, apart from the manifest list web service and the manifests themselves, to make a config file with a few parameters.

🔧 Configuring the updater Agent

The configuration file is called App.Config and is made of several parts:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="UpdaterConfiguration"
type="Q4Tech.Engineering.UpdaterAgent.Configuration.UpdaterCo
nfigurationSection, UpdaterAgent" />
    <section name="Connections"
type="Microsoft.Practices.Mobile.ConnectionMonitor.Configurat
ion.ConnectionSettingsSection,
Microsoft.Practices.Mobile.ConnectionMonitor" />
  </configSections>
  <Connections>
    <ConnectionItems>
      <add Type="CellConnection" Price="8"/>
      <add Type="NicConnection" Price="2"/>
      <add Type="DesktopConnection" Price="1"/>
    </ConnectionItems>
  </Connections>
</configuration>
```

```
</Connections>
<UpdaterConfiguration defaultDownloader="HTTPDownloader"
deviceIdProvider="HardDeviceIdProvider"
manifestListUri="http://ppp_peer/SampleWebService/Service.asmx" >
  <deviceIdProviders>
    <add name="HardDeviceIdProvider"
type="Q4Tech.Engineering.UpdaterAgent.DeviceIdProvider.HardDeviceIdProvider, UpdaterAgent" />
    <add name="ConfigDeviceIdProvider"
type="Q4Tech.Engineering.UpdaterAgent.DeviceIdProvider.ConfigDeviceIdProvider, UpdaterAgent"/>
  </deviceIdProviders>
  <downloaders>
    <add name="HTTPDownloader"
type="Microsoft.ApplicationBlocks.Updater.Downloaders.HTTPDownloader, Microsoft.ApplicationBlocks.UpdaterCF" />
  </downloaders>
</UpdaterConfiguration>
</configuration>
```

The **UpdaterConfiguration** section has two attributes:

defaultDownloader: The downloader to use by default in the binary download

deviceIdProvider: The device ID provider to use when retrieving the manifest list

manifestListUri: Is the web service URI that the agent will use to retrieve the manifest list.

and the following elements:

deviceIdProviders element: holds a collection of “Add” elements. Each one of these specify a device ID provider by friendly name and full type

downloaders element: holds a collection of “Add” elements. Each one of these specify a downloader by friendly name and full type.

👉 Adding agent support to your CF application

As an option, your application can reference a client component of the agent This gives the app the advantage of begin “aware” of the existence of the updating processes, without having to implement the full in-process logic.

The process involves adding a reference, creating an instance class, attaching to the desired events and optionally to respond actively to the events to interrupt the updating process.

👉 Setup (adding references)

The updater-aware app has first to reference the **UpdaterAgent.Common.Dll** assembly.

A **NotifierClient** class must be instantiated and initialized. The class constructor requires the application ID. In the example below, the ID is a **GUID**.

```
using Q4Tech.Engineering.UpdaterAgent.Common;
```

```
...  
  
notifierClient = new NotifierClient("{ED83BE04-9ACE-4a72-  
A1B1-2C06EF92E699}");  
  
notifierClient.Initialize();
```

Note: The class inherits from **IDisposable**, so the **Dispose()** method **must** be called properly when the instance will be no longer used.

👉 Handling the events

There are two events available for subscribing to.

UpdatesAvailable is fired when an update is available for the application, but prior to the beginning of the download. The application may choose to download or not the update, returning **true** or **false** from the event handler.

UpdatingInProcess is raised when the updated has been downloaded and prior to the activation process. Once again, the app may choose to continue or not with the update.

The following sample shows a decision schema based on user input:

```
...  
notifierClient.UpdatesAvailable += onUpdatesAvailable;  
notifierClient.UpdatingInProcess += onUpdatingInProcess;  
}  
  
bool onUpdatingInProcess()  
{  
    return  
        MessageBox.Show("Install?", "Update ready",  
            MessageBoxButtons.YesNo,  
            MessageBoxIcon.Asterisk,  
            MessageBoxDefaultButton.Button2)  
        == DialogResult.Yes;  
}  
  
bool onUpdatesAvailable()  
{  
    return  
        MessageBox.Show("Download?", "Update available",  
            MessageBoxButtons.YesNo,  
            MessageBoxIcon.Asterisk,  
            MessageBoxDefaultButton.Button2)  
        == DialogResult.Yes;  
}
```

👉 Working with not updater agent - aware applications

If an application doesn't attach to any of the events, or if the application even doesn't references to the client, the updater agent assumes that the application can be updated at any time.

In this case the open application situation must be handled carefully. If you are using CAB install files to deploy the versions, it is important to note that the Windows Mobile 5 CAB installer default behavior is the following; If the app that is being updated is open, the system sends a "Close" message to the app, if the app doesn't even respond to this message, the update is stored for after a reboot.