

EverUpdate Client Agent Developer's Guide

Table of Contents

Update Overview	1
Update Process Details	2
Advanced Client Development	3
Using the Update API's	3
Simple Scenario	3
Scenario with Custom Configuration	4
Scenario with Callbacks	4
Advanced Scenario	5
Client Configuration	5
Config File	6
App.Config	7
Logging Configuration	8
Redistributing/Deploying Your Client Application	9
Appendix A: UML	11

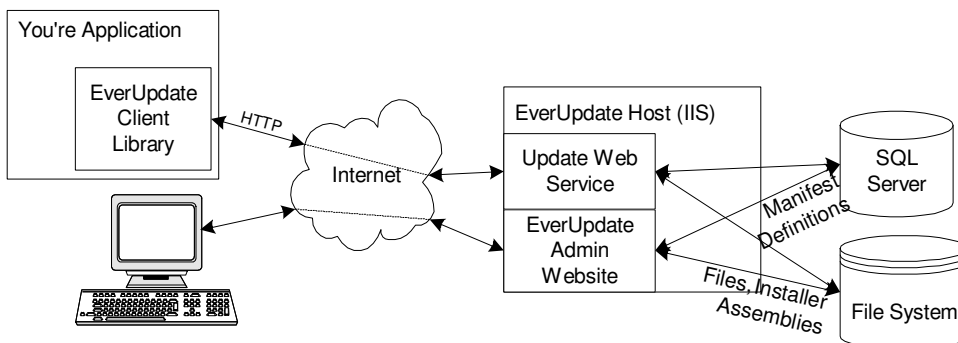
Update Process Overview

EverUpdate consists of an agent Client Library (in the form of .Net assemblies) and hosted services for managing and delivering patch and update Manifests and Manifest Items, which may be files.

As illustrated in Figure 1, you manage Manifest definitions via a remote web browser that accesses the EverUpdate Administration website (see AdminSite). You define Manifests which contain Manifest Items. The Manifests and related Items describe, for a given application, the patches or updates that are available and their contents. Manifest Items currently consist of Registry Entries and Files. Files can be executables that should be run during the patch process, custom .Net Installer assemblies that should be invoked, or simple files that should be placed on the hard drive or which replace existing files, DLL's, assemblies, etc. Files can optionally be defined as locked files, in which case they replace existing locked files during the next reboot.

The download of Manifests and Manifest Items is performed automatically by the Client Library via HTTP requests to the EverUpdate Host Web Service.

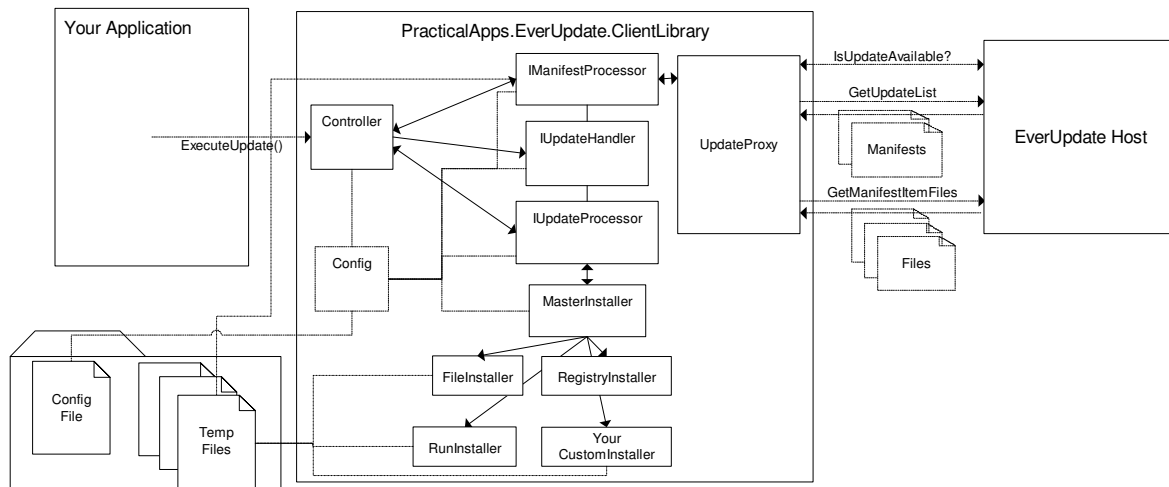
Figure 1



Update Process Details

Figure 2 illustrates in more detail what goes on under the covers of the update agent during the update check/apply process.

Figure 2



Using the simplest approach, your application simply obtains a `Controller` object and calls the `ExecuteUpdate()` method.

The `Controller` object then obtains objects implementing the `IManifestProcessor` and `IUpdateProcessor` interfaces (usually `ManifestProcessorImpl` and `UpdateProcessorImpl`), and passes them to an `IUpdateHandler`, which coordinates the process of checking for and applying updates via the `Processors`. This process includes:

- (optionally) prompting for whether to perform the check
- checking with the EverUpdate host for available update Manifests
- (optionally) prompting for whether the update(s) should be downloaded
- downloading the update (as defined by the Manifest Items in each Manifest)
- (optionally) prompting for whether the update(s) should be applied
- applying the update(s)

For each update Manifest, the files defined by file-type Manifest Items are stored in a temporary location on the hard drive

If the update is being applied, then the `IUpdateProcessor` invokes the `MasterInstaller`, passing the update Manifest.

The `MasterInstaller` then invokes the appropriate specific installer for each Manifest Item. These include:

- the `FileInstaller`, which simply deploys a file-type Manifest Item to the target location (including optional replacement file or locked file, which requires a reboot)
- the `RegistryInstaller`, which handles additions or updates to Registry entries
- the `RunInstaller`, which executes a file-type Manifest Item (allowing you, for instance, to deploy more complex Setup.exe executables)

- your own custom Installer Assemblies, which comply with the .Net Installer framework, allowing you to develop Installers that perform custom tasks outside of those support directly by the EverUpdate installers.

The behavior of various aspects of the update process is controlled via properties in the client's Config file.

If multiple manifests are available, `UpdateProcessorImpl` applies them in order based on the Version and FinalVersion properties. They will be sorted by Version, then Final Version if the Versions are equal, then applied in that order.

If the Version and FinalVersion properties of both objects are string representations of Decimal values, they will be compared numerically ("01.1" would be greater than "1.0"). For each property, if either of the values cannot be to decimal, they are compared as strings using `String.CompareTo()` ("01.1" would be less than "1.0").

The determination of which Manifests are candidates for application is made by on the host by comparing the Application and Version reported by the client (as obtained from the Config file) with those of the Manifests. Any Manifest with a Version greater than the current version as defined in the Config file is considered available.

Following the application of a candidate Manifest, the Version property of the Config file is updated to the FinalVersion property of the Manifest.

See Appendix A: UML for detailed UML diagrams or key classes and sequence diagrams.

Advanced Client Development

Using the Update API's

Simple Scenario

The minimal amount of code you need to perform an update check is as follows:

```
// include a reference to the EverUpdate Client package at the top of
// your code
using PracticalApps.EverUpdate.Client;

...

// In the portion of your code where you want to perform an update check...
// get a Controller (which manages the whole check/apply process for us)
Controller ctrl = new Controller();

// Execute the update check/apply process
ctrl.ExecuteUpdate();
```

The first step creates a `PracticalApps.EverUpdate.Client.Controller` object. The next line calls the `ExecuteUpdate()` method of that object, which performs the update check and apply process, as controlled by the default config settings.

Scenario with Custom Configuration

By default, the `Controller` object will access the default config settings (see **Config File Parameters**, below). Before you invoke `ExecuteUpdate()`, however, you can override how the config settings are changed, or modify the settings themselves.

```
...

/// Initialize the Configuration
/// Here we call ClientUtils.GetConfig(filename) to get the config from a custom
/// settings file instead of the default behavior of automatically loading
/// settings from config.bin or config.exl (depending on which is in the
/// current path).
config = ClientUtils.GetConfig("MyConfigFile.exl");
// override a config property
config.Keys["Company"] = "Acme, Inc.";

// get a Controller, initializing it with the specific Config object
Controller ctrl = new Controller(config);

// Execute the update check/apply process
ctrl.ExecuteUpdate();

...
```

Scenario with Callbacks

Under the covers, `Controller` uses an `UpdateProcessor` to perform the process of checking for updates, and a `ManifestProcessor` for performing the process of applying any updates specified by Manifests discovered in the first step by the `UpdateProcessor` (including download, and application of `ManifestItems`). The `UpdateProcessor` and `ManifestProcessor` are exposed as properties of the `Controller` and you can attach custom handlers to those objects so that your code can be notified as the update check/apply process progresses:

```
// include a reference to the EverUpdate Client package at the top of
// your code
using PracticalApps.EverUpdate.Client;

...

// In the portion of your code where you want to perform an update check...
// get a Controller (which manages the whole check/apply process for us)
Controller ctrl = new Controller(config);

// Attach our own event handler to the Controller's UpdateProcessor. This
// handler will get notified when an update actually occurs, so we can
// update the LastUpdate date/time.
ctrl.UpdateProcessor.UpdateEvtHndlr +=
    new UpdateEventHandler(My_UpdateEvtHndlr);
ctrl.ManifestProcessor.ManifestEvtHndlr +=
    new ManifestEventHandler(My_UpdateEvtHndlr);

// Execute the update check/apply process
ctrl.ExecuteUpdate();

...
```

```

private static void My_UpdateEvtHndlr(object sender, UpdateEventArgs e) {
    // ...Do stuff...

    // for example:
    if (e.Type == UpdateEventType.MANIFESTSTART) {
        ShowYourCustomProgressBar();
    } else if (e.Type == UpdateEventType.ITEMSTART) {
        IncrementYourCustomProgressBar();
    }
}

private static void My_UpdateEvtHndlr(object sender, UpdateEventArgs e) {
    // ...Do other stuff...
}

```

Advanced Scenario

Advanced developers can access certain functions exposed by the update framework directly. In the example below, we directly access the `UpdateProxy` to check for updates, then list the update Manifests available, if any exist. The code fragment below is lifted from the source for the `SampleClient` included in this SDK. It assumes you have a form with various `TextBox` controls defining the Company, Application, and Version, and for displaying the results.

```

// get an UpdateProxy object, initialized with the settings in myConfig
IUpdateProxy proxy = new UpdateProxy(myConfig);

// check for updates
bool result = proxy.IsUpdateAvailable(
    true, this.txtCompany.Text,
    this.txtAppName.Text, this.txtVersion.Text );

// if updates exist, list them
if (result.ToString().Equals("True")) {
    ArrayList result = proxy.GetUpdateList(
        true, this.txtCompany.Text,
        this.txtAppName.Text, this.txtVersion.Text );

    if (result != null) {
        result.Sort();
        this.txtResult2.Text = "";
        for (int i = 0; i < result.Count; i++) {
            string desc = ((Manifest)result[i]).Description;
            this.txtResult2.Text +=
                ((Manifest)result[i]).Name + " : " + desc + "\n";
            if (log.IsDebugEnabled) log.Debug("Update:" + desc );
            foreach (ManifestItem mi in ((Manifest)result[i]).ManifestItems) {
                this.txtResult2.Text += "   " + mi.Name + "\n";
            }
        }
    } else {
        this.txtResult2.Text = "NO UPDATES AVAILABLE";
    }
}
}

```

Client Configuration

There are 3 configuration areas that affect the EverUpdate client:

- The Config File

- Your application's App.config
- Logging configuration (optional)

Config File

The behavior of the EverUpdate client library is controlled via configuration settings. By default, the Config file will be loaded from config.exl, if found, otherwise it will load config.bin. An .exl file is an xml serialization of a config collection, a .bin file is a standard binary serialization of a config collection. It is recommended that when you distribute you update-enabled application to your end-users, you include a .bin file, since it offers a better inherent level of obfuscation, thus making it less likely that the end-user will attempt to tamper with the config settings.

You edit the properties contained in your config file using the ConfigManager application included in this SDK.

Config file parameters:

Parameter Name	Optional	Description
Application	N	The Name of your application. Used to match against the Application name of available Manifests when checking for updates.
ApplyProgressFormAssembly	Y	Assembly containing the custom AbstractApplyProgressFormClass (below)
ApplyProgressFormClass	Y	Name of a custom form class extending AbstractApplyProgressForm that should be invoked to display the progress as a patch is applied (only if ShowApplyProgressForm is true).
BackupFiles	N	Indicates whether files that are being replace during the update should first be backed up to a temporary backup directory.
CheckForUpdatePromptAssembly	Y	Assembly containing the custom CheckForUpdatePromptClass (below)
CheckForUpdatePromptClass	Y	Name of a custom form class extending AbstractCheckForUpdatePrompt that should be invoked to prompt your user for whether they want to check for new updates (only if PromptBeforeCheckForUpdate is true).
Company	N	The name of the Company your application is distributed under. Used to match against the company name of available Manifests when checking for updates.
DownloadProgressFormAssembly	N	Assembly containing the custom DownloadProgressFormClass class (below)
DownloadProgressFormClass	N	Name of a custom form class extending AbstractDownloadProgressForm that should be invoked to display the progress as the Manifest Items comprising a patch are downloaded (only if ShowDownloadProgressForm is true).
DownloadPromptFormAssembly	N	Assembly containing the custom DownloadPromptFormClass (below)
DownloadPromptFormClass	N	Name of a custom form class extending AbstractDownloadConfirmPrompt that should be invoked to prompt your user for whether they want to download available updates at this time (only if

		PromptBeforeDownload is true).
EUService.ProxyAddress	Y	If present and EUService.UseProxy is True, specifies the address of an HTTP Proxy server that should be used by the client to reach the update host
EUService.ProxyPassword	Y	If present and EUService.UseProxy is True, specifies the Proxy server password
EUService.ProxyUserName	Y	If present and EUService.UseProxy is True, specifies the Proxy server username
EUService.Update	Y	Specifies the URL of the update host web service
EUService.UseProxy	Y	If present, causes the update client to connect through an HTTP Proxy server when connecting to the update host web service
InstallLogFile	Y	Passed in the logFile option to Install assemblies, if any exist in a Manifest being applied, invoked during the install (see .Net Framework documentation on installers).
LogToConsole	Y	Passed in the logToConsole option to Install assemblies, if any exist in a Manifest being applied, invoked during the install (see .Net Framework documentation on installers).
PromptBeforeApply	N	
PromptBeforeCheckForUpdate	N	
PromptBeforeDownload	N	
PromptBeforeReboot	N	
ShowApplyProgressForm	N	
ShowDownloadProgressForm	N	
TempDir	N	During an Update install, TempDir will be set to a temporary directory where downloaded files are saved prior to applying the update. Setting this property has no effect – it is effectively read-only.
Version		The current version of the application performing update checks. The version will be updated following successful application of an update Manifest to the final version indicated by the Manifest.

EverUdate's configuration supports token substitution. When you use a placeholder of the following format in one of your configuration values, EverUpdate will substitute the indicated value for the placeholder at runtime:

"{CFG_foo}", where foo is a value from app.config of the executing application

"{ENVPATH_foo}", where foo is one of the **Environment.SpecialFolder** enum values

"{ENVVAR_foo}", where foo is an environment variable

"{EVERUPDATE_foo}", where foo is another parameter defined in the current EverUpdate config space

App.Config

You must include a section in your application's .config file defining availability of Microsoft's Web Services Extensions 2.0 and disabling size limitations for downloaded files. These settings enable the SOAP support necessary for EverUpdate to download patches from the host web service.

```

<configuration>

  <configSections>
    <section name="microsoft.web.services2"
      type="Microsoft.Web.Services2.Configuration.WebServicesConfiguration,
      Microsoft.Web.Services2, Version=2.0.0.0, Culture=neutral,
      PublicKeyToken=31bf3856ad364e35" />
  </configSections>

  <appSettings>
  </appSettings>

  <microsoft.web.services2>
    <messaging>
      <maxRequestLength>-1</maxRequestLength>
    </messaging>
    <security>
      <timeToleranceInSeconds>86400</timeToleranceInSeconds>
    </security>
    <diagnostics />
  </microsoft.web.services2>

</configuration>

```

Logging Configuration

The EverUpdate client library assembly incorporates log4net to log debug and error information messages during execution. We recommend that you enable logging to log, at a minimum, error information. This will provide you with valuable troubleshooting information if the end user of your application reports problems getting updates.

You may wish to familiarize yourself with log4net in addition to reading this section. More information on the log4net framework is at

<http://logging.apache.org/log4net/release/manual/configuration.html>

You can enable logging in one of two ways:

Via app.config

To load the logging configuration from your application's .config file, you'll first need to register the log4net config section by adding an entry like this to the <configuration> section¹:

```

<configSections>
  <section name="log4net"
    type="log4net.Config.Log4NetConfigurationSectionHandler, log4net" />
</configSections>

```

you'll then include a log4net configuration:

```

<log4net>
  <!-- Define a file output appenders -->
  <appender name="LogFileAppender" type="log4net.Appender.FileAppender" >
    <!-- Example using relative path -->
    <file value="./EverUpdate.log" />
    <!-- Example using environment variables in params -->
    <!--file value="{TMP}/EverUpdate.log" /-->
  </appender>
</log4net>

```

¹ Note that there should already be a "<section name='microsoft.web.webservices2'.../>" entry for the webservice configuration within the <configSections> tags

```

    <appendToFile value="true" />

    <!-- An alternate output encoding can be specified -->
    <!-- <encoding value="unicodeFFFF" /> -->
    <layout type="log4net.Layout.PatternLayout">
      <header value="[Header]#13;#10;"/>
      <footer value="[Footer]#13;#10;"/>
      <conversionPattern value="%d [%t] %-5p %c [%x] &lt;%X{auth}&gt; - %m%n"/>
    </layout>

  </appender>
  <!-- Setup the root category, add the appender and set the default level -->
  <root>
    <level value="DEBUG" />      <!--Set this to DEBUG, WARN, ERROR, etc. -->
    <appender-ref ref="LogFileAppender" />
  </root>
</log4net>

```

Finally, you'll add the following statement to the initialization code of your application to cause log4net to be configured based on the app.config file:

```
log4net.Config.DOMConfigurator.Configure();
```

Via a custom config file

To use a separate file to define your logging settings, simply create a file with the `<log4net>` section described above in it, save it to the directory your executable runs from, and add the following statement to your application's initialization code:

```
log4net.Config.DOMConfigurator.Configure("<your file name>");
```

The sample executables shipped with this SDK use this approach. Refer to the source code for those applications for an example.

Redistributing/Deploying Your Client Application

In order for applications you build that incorporate the EverUpdate client to function correctly, certain prerequisites must exist on the client machines to which your application is deployed. These include:

- The EverUpdate client components
- Microsoft's Web Services Enhancements (WSE) 2.0
- A custom EverUpdate Config file

EverUpdate Client Components

You first need to include the EverUpdate client components, which consist of numerous assemblies (.dll's), in your Setup program. The easy way to do this is to include the EverUpdateClient.msm merge module in the setup project for your application.

WSE 2.0

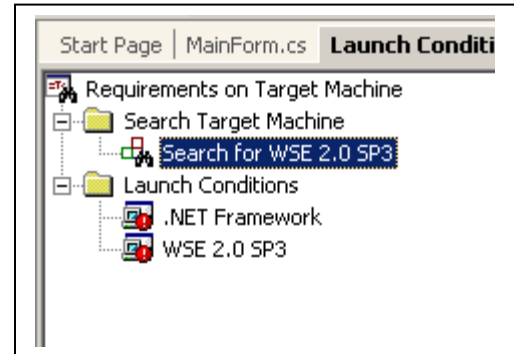
The EverUpdate client library relies on Web Services with SOAP to download patch Manifests and their contents. This implementation relies on Microsoft's Web Services Enhancements (WSE) 2.0 .Net framework add-on to provide security and binary download capabilities. WSE 2.0 support must therefore be installed on each client machine where the EverUpdate client is used.

Microsoft recommends that this support always be installed using the WSE 2.0 redistributable they provide, so you should ensure that your Setup project either includes a 'bootstrapper' that installs the Microsoft WSE 2.0 redistributable, or requires it as a prerequisite before your application can be installed.

Requiring WSE 2.0 as a prerequisite is typically achieved by adding a Launch Condition that requires the registry entry indicating that WSE 2.0 is installed. For example, perform the following step in your application's setup project:

Add a Windows Installer Search (as illustrated in this figure) to the "Search Target Machine" folder with the following properties:

```
(name)          : Search for WSE 2.0 SP3
ComponentId    : {AC245E8D-C75F-4B53-A0CF-A9E47837C90E}
Property       : WSE20SP3EXISTS
```



Add a Launch condition to the "Launch Conditions" folder with the following properties:

```
(name)          : Search for WSE 2.0 SP3
ComponentId    : {AC245E8D-C75F-4B53-A0CF-A9E47837C90E}
Condition      : WSE20SP3EXISTS
InstallUrl     :
                http://www.microsoft.com/downloads/details.aspx?FamilyID=80
                70e1de-22e1-4c78-ab9f-07a7fcf1b6aa&DisplayLang=en
Message       : {your application} requires Web Services Enhancements (WSE)
                2.0 for Microsoft .NET. Do you want to go to Microsoft's site to
                download this component now?
```

Include a Customized EverUpdate Config File

You will need to include an EverUpdate Config file that contains properties that have been customized to your application. As described above, these properties control the behavior of the update process, allowing you to tailor it to your specific requirements. Include a config.xml or config.bin file in the Application Folder under the File view of your setup project. We recommend that you use the .bin format in order to deter end users from trying to tweak the behavior or the updater (unless, of course, you want them to be able to do so).

Appendix A: UML

Figure 3

Primary classes involved in update process

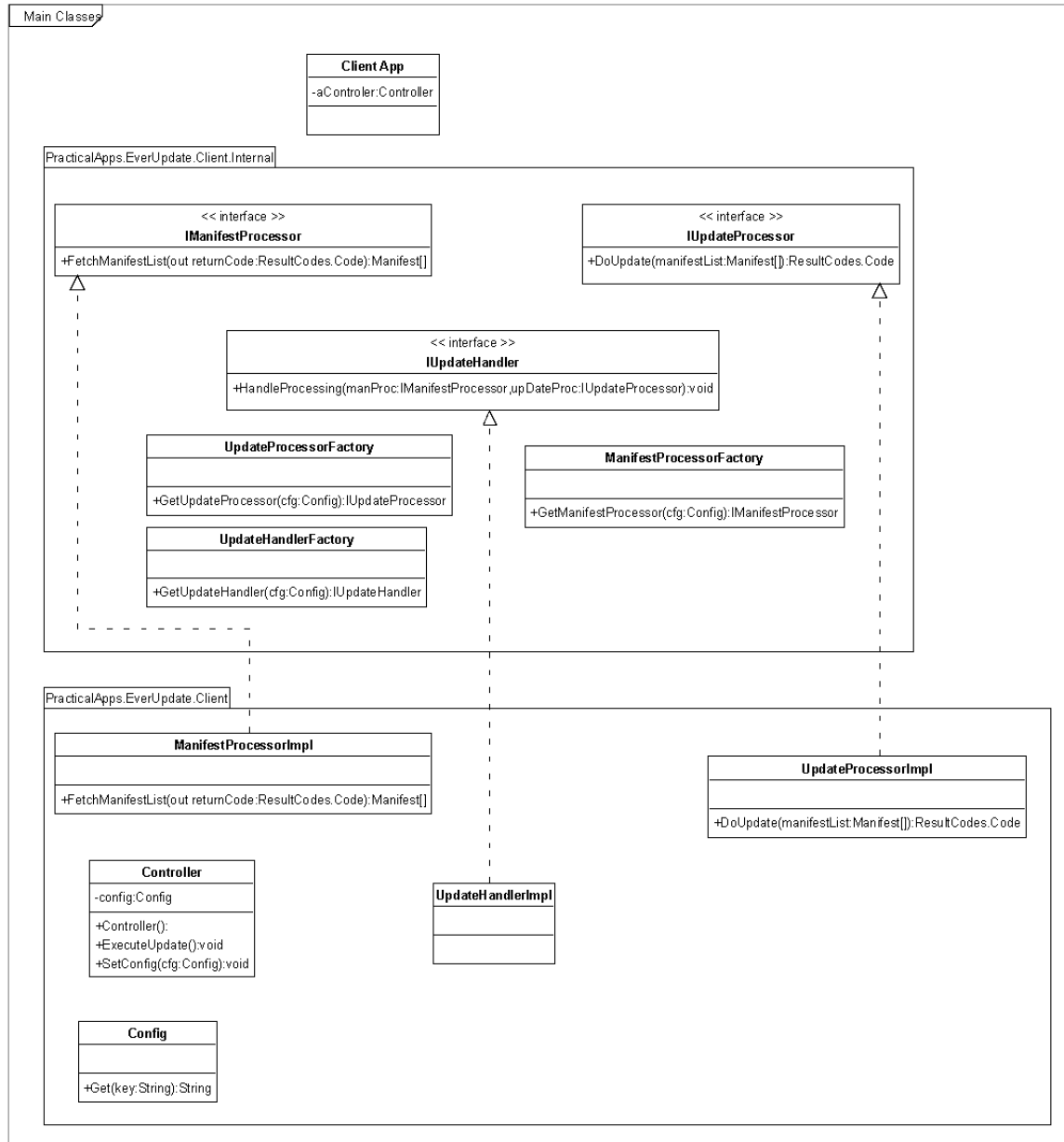


Figure 4

Generalized sequence diagram illustrating update process

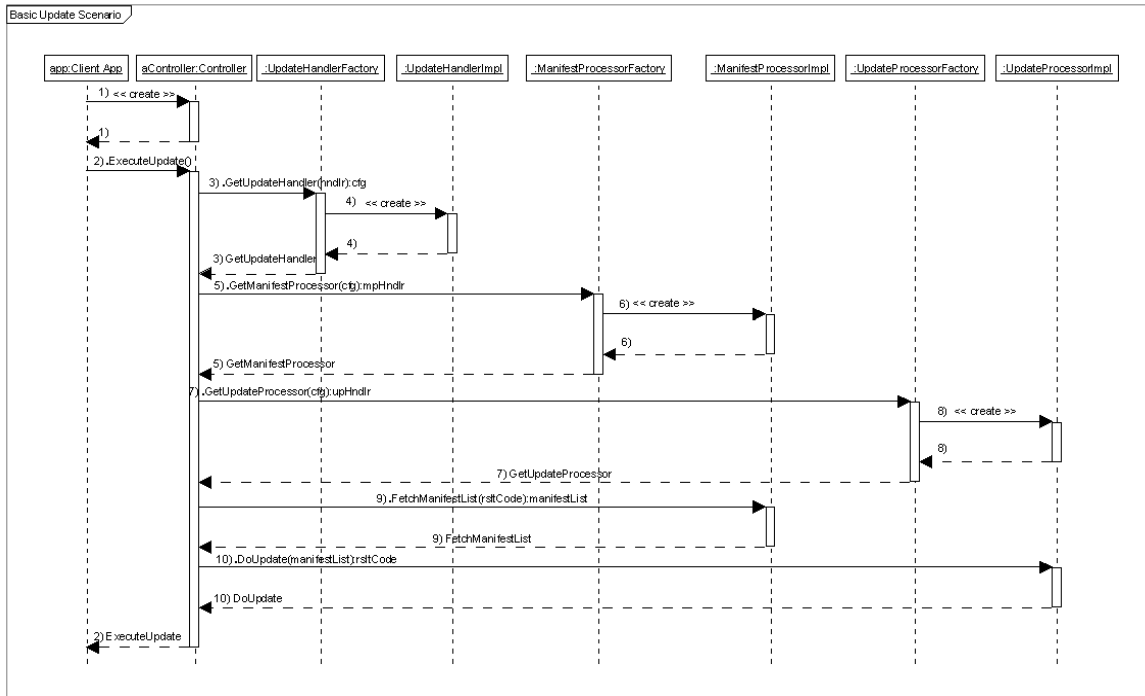


Figure 5

Classes involved in custom forms and prompts

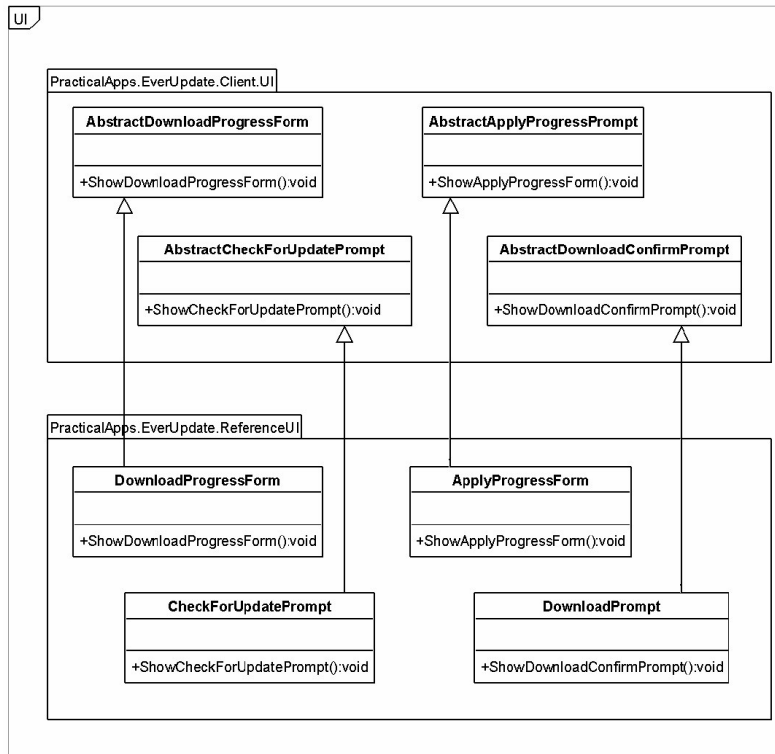


Figure 6

Example of how progress form implementation is obtained

