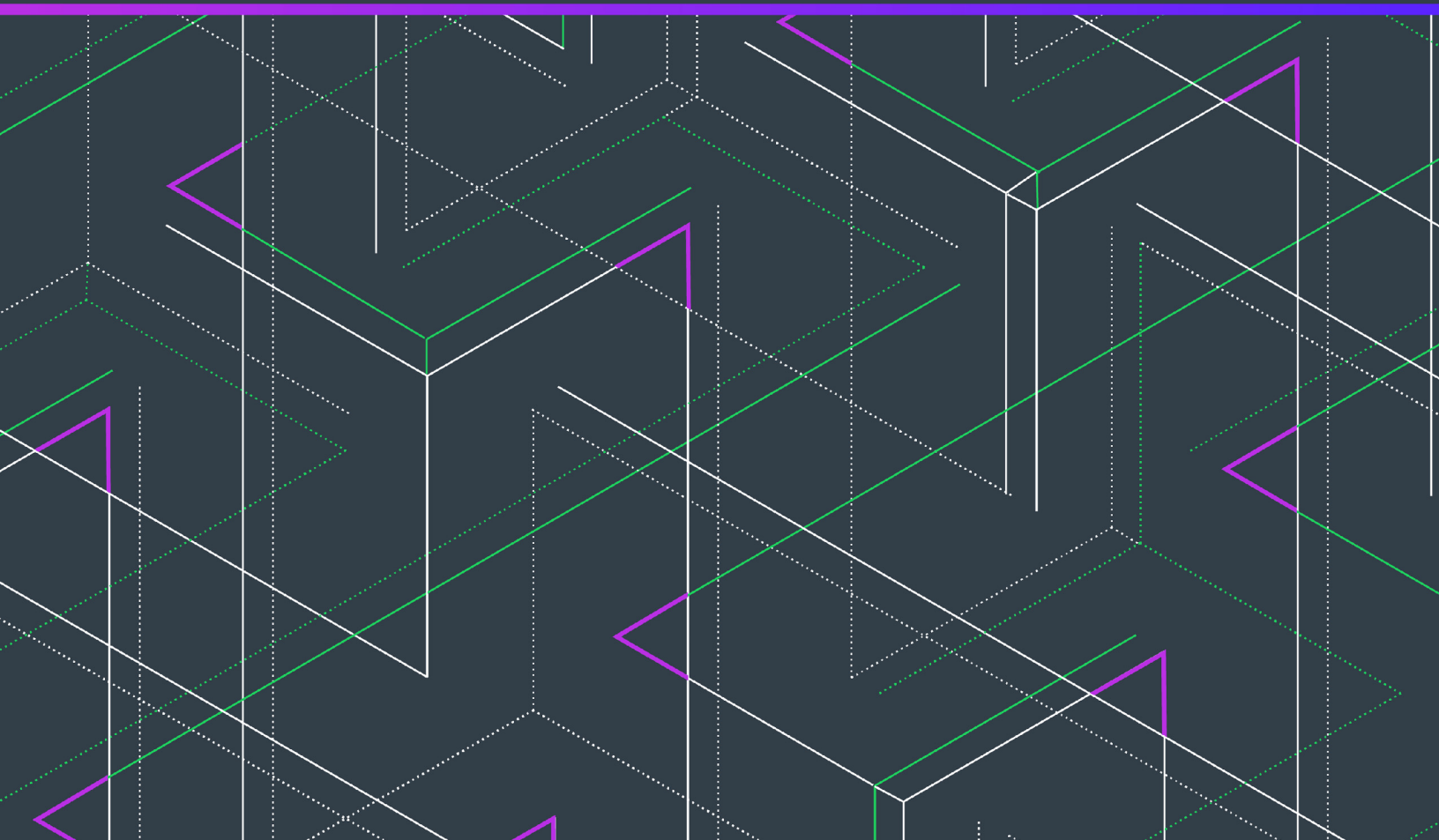
revenera

FlexNet Embedded Client 2020 R3

.NET XT SDK User Guide



Legal Information

Book Name:	FlexNet Embedded Client 2020 R3 .NET XT SDK User Guide
Part Number:	FNE-UAI-2020R3-NXTSDK-UG00
Product Release Date:	July 2020
Documentation Last Updated:	August 02, 2019

Copyright Notice

Intellectual Property

Restricted Rights Legend

Contents

1 About the FlexNet Embedded Client .NET XT Toolkit and this Guide 11

Toolkits Described in This Book 11

FlexNet Embedded Client .NET XT Toolkit 11

FlexNet Embedded Client .NET Core XT Toolkit 12

User Guide Overview 12

Product Support Resources 13

Contact Us 14

2 Quick Start with the .NET XT Toolkit 15

Toolkit Requirements 15

Downloading the Toolkit 15

Creating the Producer Identity 16

Creating the Identity Binary Files 16

Generating .NET-compatible Identity Data 18

Distributing Identity Data 18

Updating Identity Data 19

Building and Running the "BasicClient" Licensing Example 19

Building the "BasicClient" Example 20

Troubleshooting Compilation Errors for the "BasicClient" Example 20

Preparing to Run "BasicClient" 20

Generate License Rights 21

Install the FlexNet Embedded Client Libraries for FlexNet Embedded 21

Running "BasicClient" 22

Troubleshooting Build Errors for the "BasicClient" Example 22

Building and Running the "Notification" Example 23

Building the "Notification" Example 23

Troubleshooting Compilation Errors for the "Notification" Example 23

Preparing to Run "Notification" 24

Install the FlexNet Embedded Client Libraries for Updates and Insights 24

Add a Product and Its Update to the Publisher Site . . . 24
Running “Notification” . . . 25
Troubleshooting Build Errors for the “Notification” Example . . . 27
Next Steps . . . 27

3 Quick Start with the .NET Core XT Toolkit . . . 29
Toolkit Requirements . . . 29
Downloading the Toolkit . . . 29
Creating the Producer Identity . . . 30
Create the Identity Binary Files . . . 30
Generate .NET-compatible Identity Data . . . 32
Distribute Identity Data . . . 32
Update Identity Data . . . 32
Building and Running the “BasicClient” Licensing Example . . . 33
Phase 1: Provide the Prerequisites for “BasicClient” . . . 33
Client Identity File: Copied to Proper Location . . . 34
License Rights: Created and Copied to Proper Location . . . 34
.Net Core: Installed on the Target Machine . . . 35
Phase 2: Prepare to Build “BasicClient” Executable . . . 35
Step 1: Ensure Project File Points to Correct .NET Core Framework Version . . . 35
Step 2: Run “restore” to Obtain Latest Packages . . . 35
Step 3: Copy “FlxCore” to the Project Folder . . . 36
Phase 3: Build and Run the “BasicClient” Example . . . 36
Troubleshooting Compilation or Execution Errors for the BasicClient Example . . . 37
Next Steps . . . 38

4 Toolkit Overview . . . 39
Concepts: Licensing and Updates Functionality . . . 39
FlexNet Embedded Concepts . . . 40
Hostids . . . 40
Feature Definitions . . . 41
Back-office Servers and License Servers . . . 43
Trusted Storage . . . 44
Capability Requests and Responses . . . 44
Concepts of Updates and Insights . . . 45
Updates and Insights Client . . . 45
Notifications . . . 46
Producer Site and Portal . . . 46
Notification Server . . . 46
Toolkit Requirements . . . 46
FlexNet Embedded Client .NET XT Toolkit Contents . . . 46
FlexNet Embedded Client .NET Core XT Toolkit Contents . . . 48
About the Example Projects . . . 49
FlexNet Embedded Examples . . . 49

Updates and Insights Examples . . . 50
Building and Running the Examples in the .NET XT Toolkit . . . 50
Obtaining Producer Identity Data . . . 50
Building the Examples . . . 51
Running the Examples . . . 51
Displaying Usage Help for an Example . . . 51
Running the FlexNet Embedded Examples . . . 52
Running the Updates and Insights Example . . . 53
Building and Running Examples in the .NET Core XT Toolkit . . . 54
Obtaining Producer Identity Data . . . 54
Basic Process for Building and Running an Example in the .NET Core XT Toolkit . . . 54
Phase 1: Provide the Prerequisites . . . 54
Phase 2: Prepare to Build the Executable for an Example . . . 56
Phase 3: Build and Run the Example . . . 57
Displaying Usage Help for an Example . . . 58
Toolkit Files to Distribute with Your Product . . . 58
.NET XT Toolkit Files to Distribute . . . 58
.NET Core XT Toolkit Files to Distribute . . . 59

5 Overview of the .NET XT APIs . . . 61
FlexNet Embedded API Interfaces . . . 61
Updates and Insights API Interfaces . . . 62
FlexNet Common API Interfaces . . . 63
Conventions for Retrieving Exception Information . . . 63

6 Using the FlexNet Embedded APIs . . . 65
Common Steps to Prepare for Licensing . . . 66
Creating Your Producer Identity Files . . . 66
Creating Core Licensing Objects . . . 66
Specifying the Trusted Storage Location . . . 67
Specifying the Hostid Type to Use . . . 68
Final “Get Licensing” Argument . . . 69
Detecting a Containerized Environment . . . 69
Detecting a Cloned Environment . . . 70
Detecting Clock Windback . . . 70
Identifying the Device User . . . 71
Retrieving Feature Expiration and Grace Period Information . . . 71
Types of Expiration Information Available for Retrieval . . . 71
.NET Properties Used to Retrieve Expiration Information . . . 72
Including Vendor Dictionary Data . . . 72
Advanced Topic: Secure Anchoring . . . 73
Prerequisites . . . 73
Enabling Secure Anchoring . . . 73
Buffer Licenses . . . 73
Setting Up the License File . . . 74

Step 1: Create an Unsigned License File . . . 74
Step 2: Generate a Signed Binary License File . . . 74
Using the License on the Client . . . 74
Step 1: Create and Populate the License Sources . . . 75
Step 2: Acquire the License(s) . . . 75
Step 3: Read the License Details . . . 76
Licenses Obtained from the Back-Office Server . . . 76
FlexNet Operations as "Back-Office Server" . . . 77
Configuring the Back-office Server to Provide Access to Licenses . . . 77
Activation or Upgrade Steps . . . 77
Step 1: Create the License Source . . . 77
Step 2: Create the Capability Request . . . 78
Additional Capability-Request Options . . . 78
Step 3: Send the Request to the Back-Office Server . . . 80
Step 4: Process the Capability Response . . . 81
Licenses Obtained from a License Server . . . 82
Provision the License Server with Licenses for the Demonstration . . . 83
Register the Client with the Cloud Licensing Service . . . 83
Provide the URL for the License Server in the Command . . . 83
Modify the Example Code to Request "desired features" . . . 84
Additional Capability-Request Options . . . 84
Incremental Capability Requests . . . 84
Attribute to Check Out All Available Quantity for a Feature If Requested Count Cannot Be Satisfied . . . 86
Feature Selectors in a Capability Request . . . 88
Secondary Hostids . . . 89
Option to Force a Capability Response . . . 90
Borrow Interval and Granularity Overrides . . . 90
License Checkout from the License Server . . . 91
Capability Preview . . . 92
Types of Preview Counts . . . 92
Creating a Preview Capability Request . . . 93
Processing the Preview Capability Response . . . 94
Creating a Regular Capability Request Based on Preview Features . . . 96
Other Considerations . . . 96
Limited-duration Trials . . . 97
Trial Preparation . . . 97
Create the Binary Trial License Rights . . . 97
Getting and Using the Trial on the Client System . . . 98
Step 1: Create and Populate the License Sources . . . 98
Step 2: Get Trial Data from the Binary Trial File . . . 98
Secure Re-hosting . . . 99
Removing Capabilities from Host A . . . 100
Step 1: Start License-Enabled Code on Host A . . . 100
Step 2: Submit Capability Request from Host A to the Back-Office Server . . . 100
Step 3: Back-Office Server Processes Request and Sends "Reduced" Response Back to Host A . . . 101
Step 4: Process "Reduced" Capability Response on Host A . . . 101

Step 5: Submit Another Capability Request from Host A to the Back-Office Server . . . 102
Step 6: Back-Office Server Processes Capability Request from Host A . . . 102
Adding Capabilities to Host B . . . 103
Step 7: Start License-Enabled Code on Host B . . . 104
Step 8: Submit Capability Request from Host B to the Back-Office Server . . . 104
Step 9: Back-Office Server Processes Request and Sends Response Back to Host B . . . 104
Capturing Feature Usage on the Client . . . 104
Capability Requests and Usage Capture . . . 105
Operation Type . . . 105
Correlation ID . . . 106
Other Optional Identifiers . . . 106
Desired Features and Rights IDs . . . 106
Preparation in FlexNet Operations . . . 107
License Source Creation . . . 107
Client Registration with the Cloud Licensing Service . . . 108
Uncapped Usage Capture . . . 109
Capped Usage Capture . . . 110
Recall a “Used” Metered Feature . . . 111
Post-Usage-Capture: Managing Usage Data . . . 111
Additional Metered License Attributes . . . 111
Examining License Rights in a License Source . . . 112
Step 1: Create and Populate a Diagnostic License Source . . . 113
Step 2: Examine Features in the Feature Collection . . . 114
Advanced Topic: FlexNet Publisher Certificate Support . . . 117
Preparing Your Identity Data for Certificate Support . . . 117
Using the Lmflex Example . . . 118
Create the Certificate License Source . . . 118
Acquire Features from the Certificate License Source . . . 118
Differences in Certificate Licensing Behavior . . . 119
Advanced Topic: Multiple-Source Regenerative Licensing . . . 119
Use Cases for Multiple-Source Regenerative Licensing . . . 120
Providing Support for Multiple-Source Regenerative Licensing in the Client Code . . . 120
Creating the License Source for a Server Instance . . . 120
Identifying the Server Instance in the Capability Request . . . 121
Processing the Response from a Server Instance . . . 121
Considerations . . . 122

7 Using the Updates and Insights APIs . . . 123

Common Preparation Steps . . . 124
Obtaining Your Producer Identity Files . . . 124
Adding a Product and Its Update to Your Producer Site . . . 124
About the Manifest File for An Update Notification . . . 125
Creating Core Notification Objects and Registering the Client . . . 125
Setting Up the Updates and Insights Client Object . . . 125
Registering the Client Device with FlexNet Operations . . . 126

Setting Up the Product Package Object . . . 126
Obtaining Notifications and Downloading and Installing the Updates . . . 127
Obtaining Notifications . . . 128
Step 1: Send the Notification Request . . . 128
Step 2: Inspect the Collection . . . 129
Downloading and Installing an Update . . . 130
Download Payload and Install the Update . . . 130
(Optional) Use a Callback Function to Track the Update Progress . . . 131
One-time Event: Client Device Registration with FlexNet Operations . . . 133
The Registration Process . . . 134
Requesting the Client Registration . . . 134
About Client Communications for the Updates and Insights . . . 135

8 Utility Reference . . . 137
Tools Shared by FlexNet Embedded and Updates and Insights . . . 138
Publisher Identity Utility . . . 138
Purpose . . . 138
Usage . . . 139
Entering Your Identity Data . . . 140
Further Tasks and Considerations . . . 140
Print Binary Utility . . . 141
Viewing Contents . . . 141
Viewing Contents and Validating Signatures . . . 141
Displaying Binary-File Contents in Compiler-Readable Format . . . 141
Converting License Data to Base 64 Format in FlexNet Embedded . . . 142
Additional printbin Switches . . . 142
Tools Specific to FlexNet Embedded . . . 143
Identity Update Utility . . . 143
Usage . . . 143
Device Hostid Types Used to Restrict Hostid Detection . . . 144
Example Identity Update . . . 145
License Conversion Utility . . . 146
Trial File Utility . . . 146
Capability Server Utility . . . 147
Considerations for Using the Utility . . . 148
Usage . . . 148
Starting and Stopping the Capability Server Utility . . . 148
About License Templates . . . 149
Endpoint for Sending Capability Requests to the Utility . . . 151
Capability Request Utility . . . 151
Capability Response Utility . . . 154
Secure Profile Utility . . . 156
Viewing Available Security Profiles . . . 156
Enabling Secure Anchoring . . . 157
.NET XT Toolbox . . . 157

Preparing the .NET XT Toolbox . . . 157
Working with License Sources . . . 159
Server Communications . . . 161
License Acquisition . . . 163

A Manifest File Contents for a Product Update . . . 165

Manifest File Format . . . 165
Header Line . . . 166
File Entries . . . 166
Manifest File Setup Rules . . . 167
Manifest File Processing Rules . . . 167

Index . . . 169

1

About the FlexNet Embedded Client .NET XT Toolkit and this Guide

The *FlexNet Embedded Client C SDK User Guide* provides guidance on how to integrate the .NET XT FlexNet Embedded licensing and Updates and Insights functionality into your product code.

Toolkits Described in This Book

The guide covers how to get started with and use these two .NET XT toolkits:

- FlexNet Embedded Client .NET XT Toolkit
- FlexNet Embedded Client .NET Core XT Toolkit

FlexNet Embedded Client .NET XT Toolkit

The FlexNet Embedded Client .NET XT toolkit combines the .NET Framework functionality of FlexNet Embedded licensing and Updates and Insights services, offering implementers a source of APIs and tools to leverage one or both of these components in their applications. Briefly, the FlexNet Embedded Client .NET XT toolkit offers the following:

- FlexNet Embedded functionality that provides a secure licensing framework in which producers can control features to which end users are entitled on their client systems. The licensing functionality enables producers to offer different product configurations, enforce node-locked licensing, and enable hands-free activation, silent trials, and electronic (field) upgrades.
- Updates and Insights functionality that enables your product to receive notifications about product updates from FlexNet Operations and provides appropriate methods to download and install available updates.

Note • *In some cases, FlexNet Embedded licensing and the Updates and Insights operations share common functionality. However, where necessary, the guide distinguishes whether a given function is specific to licensing or to Updates and Insights.*

FlexNet Embedded Client .NET Core XT Toolkit

The FlexNet Embedded Client .NET Core XT toolkit offers the same FlexNet Embedded functionality found in the FlexNet Embedded Client .NET XT toolkit, but extends this functionality to platforms currently supported by both the FlexNet Embedded Client C XT toolkit and .NET Core, enabling you to build licensed .NET code that runs on Windows, Linux, or OS X. (See https://www.microsoft.com/net/core#windowsvs2017 for available .NET Core implementations.)

This toolkit does not support Updates and Insights functionality.

User Guide Overview

The purpose of this user guide is to provide an overview of the FlexNet Embedded Client .NET XT and .NET Core XT toolkits to help you start integrating licensing code in your product. The guide includes the following chapters:.

Table 1-1 • Overview of the *FlexNet Embedded Client .NET XT SDK User Guide*

Topic	Content
Quick Start with the .NET XT Toolkit	Provides an introductory walkthrough of preparing the FlexNet Embedded Client .NET XT toolkit and then building and running example code that performs basic licensing and Updates and Insights operations. The chapter is geared toward potential customers or current customers who want a simple “getting started” demonstration to see how the licensing and Updates and Insights functionality works.
Quick Start with the .NET Core XT Toolkit	Provides an introductory walkthrough of preparing the FlexNet Embedded Client .NET Core XT toolkit, and then building and running example code that performs basic licensing operations. The chapter is geared toward potential customers or current customers who want a simple “getting started” demonstration to see how the licensing functionality works.
Toolkit Overview	Provides an overview of the FlexNet Embedded Client .NET XT and .NET Core XT toolkits, describing: • Terminology used for FlexNet Embedded licensing and Updates and Insights notification services • Toolkit contents • Example projects included with the toolkit • Build instructions for the toolkit examples
Overview of the .NET XT APIs	Describes the groups of FlexNet Embedded, Updates and Insights, and common APIs included in the toolkit.

Table 1-1 • Overview of the *FlexNet Embedded Client .NET XT SDK User Guide*

Topic	Content
Using the FlexNet Embedded APIs	Describes the primary objects included in the FlexNet Embedded .NET XT programming model, and walks through sample implementations of various scenarios, including: • Using node-locked licenses on a client system • Using a back-office server (FlexNet Operations or the test back-office server utility `capservertutil`) to activate license rights on a FlexNet Embedded client device • Having a license server provision the client with licenses • Enabling limited-duration trial functionality on a client • Tracking feature usage
Using the Updates and Insights APIs	Walks through a sample Updates and Insights implementation that retrieves a notification for a product update and then downloads and installs the update.
Utility Reference	Explains how to use the toolkit utilities to test and prepare your licensing-enabled or updates-enabled client application for production.
Manifest File Contents for a Product Update	Provides a description of a manifest file used by Updates and Insights to download one or more files needed to install a product update. (Downloaded files marked for execution can then be run to complete the update.)

Product Support Resources

The following resources are available to assist you with using this product:

- Revenera Community
- Revenera Learning Center
- Revenera Support

Revenera Community

On the Revenera Community site, you can quickly find answers to your questions by searching content from other customers, product experts, and thought leaders. You can also post questions on discussion forums for experts to answer. For each of Revenera's product solutions, you can access forums, blog posts, and knowledge base articles.

https://community.revenera.com

Revenera Learning Center

The Revenera Learning Center offers free, self-guided, online videos to help you quickly get the most out of your Revenera products. You can find a complete list of these training videos in the Learning Center.

https://learning.revenera.com

Revenera Support

For customers who have purchased a maintenance contract for their product(s), you can submit a support case or check the status of an existing case by making selections on the **Get Support** menu of the Revenera Community.

https://community.revenera.com

Contact Us

Revenera is headquartered in Itasca, Illinois, and has offices worldwide. To contact us or to learn more about our products, visit our website at:

http://www.revenera.com

You can also follow us on social media:

- Twitter
- Facebook
- LinkedIn
- YouTube
- Instagram

2

Quick Start with the .NET XT Toolkit

This chapter describes the basics needed to get started with the FlexNet Embedded Client .NET XT toolkit:

- Toolkit Requirements
- Downloading the Toolkit
- Creating the Producer Identity
- Building and Running the “BasicClient” Licensing Example
- Building and Running the “Notification” Example
- Next Steps

The remaining chapters provide more information about the toolkit and delve into actual use of its FlexNet Embedded and Updates and Insights functionality.

Toolkit Requirements

For information about supported toolkit platforms and prerequisites for developing, building, and deploying your product with the toolkit, see the current version of the *FlexNet Embedded Client Release Notes*.

Downloading the Toolkit

The email you received from Revenera provides instructions for downloading the FlexNet Embedded Client .NET XT toolkit from the Product and License Center. These instructions can vary, depending on whether you are downloading a purchased toolkit or one that you are evaluating.

Once you have downloaded the appropriate `.zip` file for the toolkit, decompress the archive somewhere on your system.

In this book, the root directory of the decompressed toolkit is referred to as *`install_dir`*.

Creating the Producer Identity

Each producer's FlexNet Embedded Client .NET XT toolkit is separate, in the sense that no organization's license-enabled code can use another organization's license rights.

Note • *The term "producer" is synonymous with "publisher", a term more commonly used for the .Net XT toolkit user in previous documentation releases. Certain components of the toolkit functionality still use "publisher".*

Each producer is identified by a unique producer name and producer keys. To enable your FlexNet Embedded Client .NET XT toolkit, you must generate *producer identity data* to be used by your back-end tools and by your client code. A set of producer identity data files contains a combination of cryptographic data and settings used to digitally sign your license rights and notification messages on the back-office server and to validate these license rights and notification messages sent to the client.

Whether you intend to use FlexNet Embedded functionality only, Updates and Insights functionality only, or both sets of functionality, you must compile your code with access to this identity data.

To generate your organization's identity files for testing purposes, you can use the Publisher Identity utility `pubidutil` utility in the FlexNet Embedded Client .NET XT toolkit. While this utility can be used to generate identity data for your production environment, typically you would use FlexNet Operations to generate and download these identity files for your production environment.

The following sections describe how to generate the identity files using `pubidutil` and how to distribute these files:

- Creating the Identity Binary Files
- Generating .NET-compatible Identity Data
- Distributing Identity Data

Creating the Identity Binary Files

Identity files are created as binary files. To use the `pubidutil` utility provided in the FlexNet Embedded Client .NET XT toolkit, you need the organization-specific production keys included in your email from Revenera.

To run the `pubidutil` tool in graphical mode, launch *`install_dir`*`\bin\tools\pubidutil.bat`.

In the **Publisher Identity Utility** window, enter:

- Names and locations where your server-identity and client-identity files will be created (or reopened, if the files already exist).
- An **Identity Name**—such as `demo-med-rsa`—for this collection of identity settings. This name must be unique on your FlexNet Operations site.
- The **Publisher Name**, which is a case-sensitive value such as `demo`, and **Publisher Keys**, which are the five hexadecimal numbers that you obtained based on your email message from Revenera.
- The desired digital signature type and strength. For this example, select `RSA` from the **Signature Type** options, and select `Medium (1024 bit)` and `SHA-256` from the **Signature and digest strength** options.

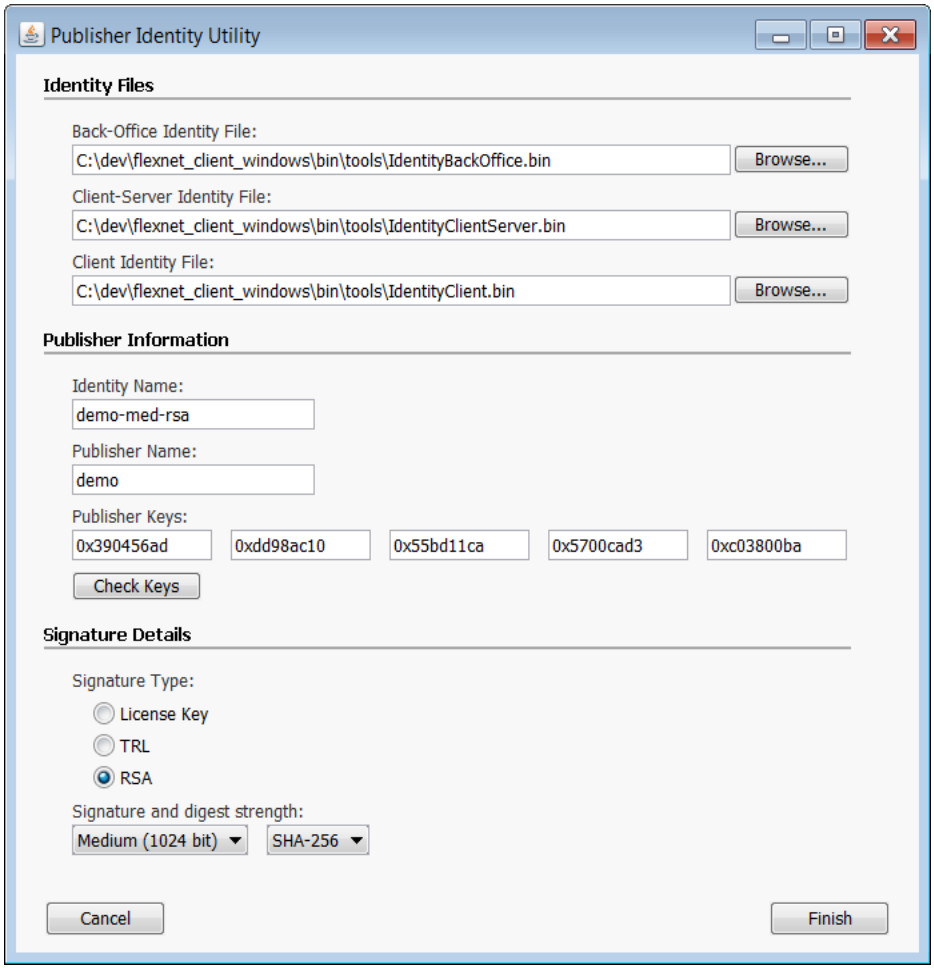


Figure 2-1: Entering Producer-specific Information in the Publisher Identity Utility

Click **Finish** when you have entered this information, and the Publisher Identity utility creates the identity files in the location you specified.

- The *back-office* identity data—by convention called `IdentityBackOffice.bin`—is used by the back-office server to digitally sign license rights and notification messages, and must be kept secure.

  For FlexNet Embedded, the back-office server is either FlexNet Operations or the test back-office server utility `capserverutil` (Capability Server utility). For Updates and Insights, the back-office server is Revenera-hosted FlexNet Operations.

- The *client* identity data—by convention called `IdentityClient.bin`—is included in your code in order to acquire license rights or procure notification messages at run time.

- The *client-server* identity data—`IdentityClientServer.bin`—is used when preparing a license server, which is a separately downloaded component (not addressed here) used with FlexNet Embedded licensing. Information about the functionality that the client can use to obtain licenses from a license server is described in the Using the FlexNet Embedded APIs.

You can also run the Publisher Identity utility in text mode by opening a command prompt window to the *`install_dir`*`\bin\tools` directory and running the `pubidutil` script with the `-console` switch, entering information in the prompts that follow.

For information about updating producer identity data, see Updating Identity Data.

Generating .NET-compatible Identity Data

The Publisher Identity utility generates your identity files in binary format. FlexNet Embedded Client .NET XT API methods that initialize client-side identity information in product code take an array of bytes as an argument. To generate a text representation of your client-identity data, you can use the `printbin` utility, also in the *`install_dir`*`\bin\tools` directory of your FlexNet Embedded Client .NET XT toolkit.

To generate a C# array of bytes for use in FlexNet Embedded Client .NET XT code, run the following command:

```
printbin -cs IdentityClient.bin -o IdentityClient.cs
```

The output should look similar to the following:

```
using System;

/* ...comment listing signature name, type, and keys... */

namespace IdentityData
{
    internal static class IdentityClient
    {
        internal static readonly byte[] IdentityData = new byte[] {
            0x68, 0x61, 0x70, 0x70, 0x79, 0x20, 0x6c, 0x69, 0x63, 0x65,
            0x6e, 0x73, 0x69, 0x6e, 0x67, 0x21, 0x20, 0x2d, 0x72, 0x6f,
    ...
            0x62, 0x65, 0x72, 0x74, 0x64, 0x00};
    }
}
```

Distributing Identity Data

To prepare the client-identity data for use in the FlexNet Embedded Client .NET XT code, copy the file `IdentityClient.cs` into a directory where it can be accessed by and compiled into your executable code. (To run the FlexNet Embedded Client .NET XT examples, copy the identity data to the *`install_dir`*`\examples\identity` directory.)

Caution • *For security reasons, it is strongly recommended that your client identity data be embedded in your code in this fashion, as opposed to loading the binary identity data from an external file at run time.*

Finally, upload the `IdentityBackOffice.bin` to one or both locations as needed:

- If you are enabling your code for FlexNet Embedded licensing, either use the Producer Portal to upload the file to FlexNet Operations; or, if testing your license-enabled code against the back-office server utility, `capserverutil`, copy `IdentityBackOffice.bin` to the *`install_dir`*`/bin/tools` directory so that it is accessible by the utility.
- If you are enabling your code for Updates and Insights functionality, use the Producer Portal to upload `IdentityBackOffice.bin` to FlexNet Operations. Once the identity is uploaded, it is available to the Updates and Insights notification server.

For information about using the FlexNet Operations Producer Portal to upload the back-office identity, refer to the *FlexNet Operations User Guide* that is available in the Producer Portal. For information about using `capserverutil`, see Capability Server Utility in the Utility Reference chapter.

Updating Identity Data

In some cases you might need to update existing identity data with new information. For example, if you purchase additional platforms from Revenera and therefore receive new publisher keys, you need to update your identity data so that it reflects the new platform information. New identity data is always generated based on existing identity data. In other words, the publisher keys contained within the identity files are replaced with the new values. Other elements, such as signing and encryption keys, remain unchanged.

Using FlexNet Operations

Typically, you would use FlexNet Operations to update identity data for production systems. For information about how to do this, refer to the FlexNet Operations documentation. After you update identity data, you need to export the new client identity and embed it in any new clients. Existing clients in the field do not need to be updated if only publisher keys have been updated.

Using pubidutil

If you want to generate new identity data for testing purposes, you can use the `pubidutil` utility (either in command-line or GUI mode). To ensure that `pubidutil` generates new identity data that is compatible with the previous set of identity files, you need to specify the existing back-office identity file when running `pubidutil`.

To update identity data using `pubidutil` in UI mode, follow the instructions in section Creating the Identity Binary Files. Make sure that you use all original settings (identity files, identity name, publisher name, and signature details), with the exception of the publisher keys which you need to replace with the new keys that you received from Revenera.

To update identity data using `pubidutil` in console mode, run the `pubidutil` script in the `bin/tools` subdirectory of the toolkit with the `-console` switch:

```
pubidutil -console
```

You will prompted for all of the required information at the command line, with the previous identity information provided as default values. Press Enter to accept the default values, except for the publisher keys. When prompted, enter the new publisher keys.

Next Steps

Follow the instructions in the sections Generating .NET-compatible Identity Data and Distributing Identity Data to update client and server components with the new identity data.

Building and Running the "BasicClient" Licensing Example

This section walks you through the process of building and running the **BasicClient** example found in the FlexNet Embedded Client .NET XT toolkit. The process is geared toward those producers who intend to use only the FlexNet Embedded functionality in the toolkit or who intend to use both Updates and Insights functionality and FlexNet Embedded, but simply want to observe how a product license works.

The **BasicClient** example is the simplest example of code that uses FlexNet Embedded APIs to enable licensing. This example acquires license rights from a local binary file and prints a message if the license acquisition succeeds. The following sections walk you through **BasicClient** build and execution process:

- Building the "BasicClient" Example
- Preparing to Run "BasicClient"
- Running "BasicClient"

Whether you are in a demo or production environment, the build step is required to run the example.

Building the "BasicClient" Example

To build the **BasicClient** example, use these steps:

1. Open the *install_dir*\examples\client_samples\BasicClient directory and open the Visual Studio project file BasicClient.csproj. (The source code for the example is in BasicClient.cs.)
2. In Visual Studio, pull down the **Build** menu and select **Build Solution**, saving the solution file BasicClient.sln when prompted.

When the build finishes, the BasicClient executable is in a Debug subdirectory relative to the project file.

Note • *The projects are set up to build the "Any CPU" configuration. It may be necessary to select a different configuration, depending on your development and target platforms. For example, if building a 32-bit toolkit example on a 64-bit system, you should select the "x86" configuration.*

In the Preparing to Run "BasicClient" section, you will get ready to run the **BasicClient** example by generating the license rights to be acquired by the example and by installing the FlexNet Embedded client libraries required by FlexNet Embedded.

Troubleshooting Compilation Errors for the "BasicClient" Example

If your **BasicClient** project fails to compile with the error `License-enabled code requires client identity data, which you create with pubidutil and printbin -cs or with FlexNet Operations`, make your sure you have done the following:

- Copied the C#-compatible identity file that you created with pubidutil and printbin into examples\identity.
- Named the identity file IdentityClient.cs.

Preparing to Run "BasicClient"

Before running the BasicClient executable, do the following:

- Generate License Rights
- Install the FlexNet Embedded Client Libraries for FlexNet Embedded

Generate License Rights

License rights used by license-enabled code are specific to each producer, which means that a producer's license-enabled code can work with only that producer's licenses. One way to store license rights on a client is in a digitally signed binary file, which you can create based on an unsigned text representation of the license rights.

Create a text file called `license.txt` with the following contents:

```
INCREMENT survey demo 1.0 1-jan-2025 uncounted HOSTID=ID_STRING=1234567890
INCREMENT highres demo 1.0 1-jan-2025 uncounted HOSTID=ID_STRING=1234567890
```

Each line in the text representation is made up of the following components:

- The names after the `INCREMENT` keyword (`survey` and `highres`) are your *feature names*; at run time your license-enabled code attempts to acquire license rights for those features, and reacts accordingly based on whether valid rights are available.
- Each feature is tied to a particular *producer name*: `demo` for the demo toolkit, and your producer name for a production toolkit.
- Each feature is versioned; the license-enabled code requests a particular version, and if the version in the license rights is greater than or equal to the requested version, the request succeeds.
- Each feature has an expiration date; if the license rights have expired, the attempt to acquire the license will fail.
- A feature is tied to a particular client using a `HOSTID` value. In the default toolkit examples, clients are assumed to have a hard-coded string identifier "0123456789", while in practice your code specifies the desired type of client identifier (such as an Ethernet address) to examine at run time in order to compare it with the identifier in the license rights. For more information, see Hostids.

For more details about feature-definition syntax, see Feature Definitions.

To digitally sign the license rights so that only your license-enabled code can acquire the licenses, use the `licensefileutil` utility. From the *`install_dir`*`\bin\tools` directory, run the following command in a console window:

```
licensefileutil -id IdentityBackOffice.bin license.txt license.bin
```

The output, `license.bin`, is a binary representation of your license rights that can be acquired by license-enabled code at runtime. You will later copy this file to a location where your FlexNet Embedded Client .NET XT code can read it.

Install the FlexNet Embedded Client Libraries for FlexNet Embedded

FlexNet Embedded makes use of certain FlexNet Embedded client libraries for some of its licensing functionality. These libraries, located in the `lib` subdirectory of your FlexNet Embedded Client .NET XT toolkit, include the following:

- The `FlxCore.dll` (on 64-bit systems, `FlxCore64.dll`) native library
- .NET assemblies `FlxClientCommon.dll` and `FlxLicensingClient.dll`

The provided Visual Studio project for the examples automatically copies the libraries to the directory where the executable resides.

In practice, your installation program should copy these DLLs to an appropriate directory on a target system.

Important • *Always use the version of the "FlxCore" library that matches the version of FlexNet Embedded used in your implementation.*

Running "BasicClient"

At this point, you can put the pieces together and test the **BasicClient** example.

Copy the binary license rights file `license.bin` you generated in the section Generate License Rights into the current directory.

Next, launch the `BasicClient` executable (from the `examples\client_samples\BasicClient\Debug` directory), specifying the location of `license.bin` as a command-line argument:

```
BasicClient.exe license.bin
```

If the attempt to acquire the licenses is successful, you will see confirmation that the `survey` and `highres` features were successfully acquired.

```
INFO: Using default license file .\license.bin
INFO: Reading data from .\license.bin
INFO: License acquisition for feature 'survey' version '1.0' successful
INFO: License acquisition for feature 'highres' version '1.0' successful
```

Troubleshooting Build Errors for the "BasicClient" Example

If your `BasicClient` executable fails to launch or is unable to acquire any licenses, try these troubleshooting methods:

- If an error beginning `Failed to load FlxCore library` is displayed, or the application exits with a `FileNotFoundException` or `DllNotFoundException` message that refers to `FlxLicensingClient`, `FlxClientCommon` or `flxcore`, verify that the libraries `FlxCore.dll, FlxLicensingClient.dll` and `FlxClientCommon.dll` have been properly installed, as described in Install the FlexNet Embedded Client Libraries for FlexNet Embedded.

  In addition, verify that the project has been built for the correct architecture: if building an example from the 32-bit toolkit on a 64-bit system, change the build configuration from "Any CPU" to "x86".

- If an error message `ERROR: Unable to find file license.bin` is displayed, verify that the `BasicClient` executable was able to find the binary license rights file `license.bin`. You can specify the path to the buffer license file as a command-line argument to `BasicClient`.

- If an error message `Data version is not supported` is displayed, verify that the `BasicClient` executable is loading a signed binary license file, and not (for example) a text file or other file type.

- Verify that both the client identity (`IdentityClient.cs`) used by the executable and the back-office identity (`IdentityBackOffice.bin`) have been generated from the same information in the `pubidutil` utility. Both identities must use the same producer name, producer keys, and signature algorithm and strength. If a discrepancy issue exists, the example prints exception information (`ERROR: Signature didn't pass validation`) that the identities do not match.

- If launching **BasicClient** displays an error message such as `BasicClient.exe is not a valid Win32 application`, verify that the project was built with the correct configuration. For example, this error will occur when building the x64 configuration but executing the code on a 32-bit system.

- The **View** example that ships with the toolkit will display a simple diagnostic report of the license rights contained in a binary license file. If the `BasicClient` executable is unable to acquire licenses, pointing the **View** example to the binary license file may indicate issues with the original license's syntax or attributes.

Building and Running the "Notification" Example

This section walks you through the process of building and running the basic **Notification** example found in the FlexNet Embedded Client .NET XT toolkit. The process is geared toward those producers who intend to use only the Updates and Insights functionality in the toolkit or who intend to use both Updates and Insights functionality and FlexNet Embedded, but simply want to observe a sample product-notification process (in this case, for product updates).

- Building the "Notification" Example
- Preparing to Run "Notification"
- Running "Notification"

The results of running this example can help you verify whether your required Updates and Insights components are in place.

Building the "Notification" Example

Whether you are in a demo or production environment, the build step is required to run the example.

To build the **Notification** project do the following:

1. Open the project file *`install_dir`*`\examples\uai_client_samples\Notification\Notification.csproj` in Visual Studio.
2. In Visual Studio, pull down the **Build** menu and select **Build Solution**, saving the solution file `notification.sln` when prompted.

When the build finishes, the `Notification.exe` executable is available in the `Debug` subdirectory relative to the project file.

Troubleshooting Compilation Errors for the "Notification" Example

If your **Notification** project fails to compile with the error `License-enabled code requires client identity data, which you create with pubidutil and printbin -cs or with FlexNet Operations`, ensure that you have done the following:

- Copied the C#-compatible identity file that you created with `pubidutil` and `printbin` into `examples\identity`.
- Named the identity file `IdentityClient.cs`.

Preparing to Run "Notification"

Before running the `Notification` executable, do the following:

- Install the FlexNet Embedded Client Libraries for Updates and Insights
- Add a Product and Its Update to the Publisher Site

Install the FlexNet Embedded Client Libraries for Updates and Insights

Updates and Insights functionality makes use of certain FlexNet Embedded client libraries for some of its licensing functionality. These libraries, located in the `lib` subdirectory of your FlexNet Embedded Client .NET XT toolkit, include the following:

- The `FlxCore.dll` native library (`FlxCore64.dll` for 64-bit systems)
- .NET assemblies `FlxClientCommon.dll` and `FlxUAIClient.dll`

The provided Visual Studio project for the examples automatically copies these libraries to the directory where the executable resides.

In practice, your installation program should copy these DLLs to an appropriate directory on a target system.

Important • *Always use the version of the "FlxCore" library that matches the version of the FlexNet Embedded Client toolkit used in your implementation.*

Add a Product and Its Update to the Publisher Site

Before you can successfully run the **Notification** example, a sample product package must be defined and its update published in FlexNet Operations for access by the notification server. For instructions, refer to the *FlexNet Operations User Guide* that is available in the Producer Portal.

Then, before executing the **Notification** example, you can access the Package Products or the Updates page in FlexNet Operations to obtain the following information for the product package and update:

- The package ID for the product for which you will request update notifications. This information is required for running the example.
- The Microsoft Language Locale Identifier (LCID) for the language of the product package if the default value (1033) used by **Notification** example is not applicable. See Running "Notification" for details.

Tip • *For best results, follow the exercises in the sections "Getting Started with Entitlement Management" and "Getting Started with Updates and Insights" in the FlexNet Operations User Guide. There you can create a product, an entitlement, download packages, and an update that you can use to test the Updates and Insights API implementations.*

Running "Notification"

At this point, test the **Notification** example.

From the directory where the executable resides (*install_dir*\examples\uai_client_samples\Notification\Debug), run a command similar to this:

```
notification -server https://siteID-ns-uat.flexnetoperations.com -register ACT01-PhotoPrint
    -download -packageid PhotoPrintInstaller -productlang 1033
```

The following describes the arguments used in the command:

- `-server` *url* identifies the location of the notification server to which the Updates and Insights client is connecting to obtain available notifications. This URL needs to be made available to each of the clients.

  Note that the URL above points to a User Acceptance Test (UAT) environment indicated by the `-uat` following the *siteID*. For production environments, the `-uat` is omitted.

- `-register` *activationid* uses a valid activation ID (called *rights ID* in FlexNet Embedded) to register the hostid of the client device with Revenera-hosted FlexNet Operations. Registration is a one-time event for the device. If the device has been previously registered either through Updates and Insights or through FlexNet Embedded client functionality, you do not need to re-register it. The `-register` and `-packageid` arguments can be specified in the same command first to register the device and then retrieve notifications.

  The activation ID specified for this argument must be associated with an entitlement in FlexNet Operations that is mapped to a valid customer account to which you are registering this device. The entitlement itself must contain an unmetered, uncounted license for the purpose of simply identifying the client device.

  In a production environment, the producer must supply the end user with the appropriate activation ID with which to register the device. This information is typically conveyed through an email message.

  For more information, see One-time Event: Client Device Registration with FlexNet Operations in the *Using the* Updates and Insights *APIs* chapter.

- `-download` downloads the payload associated with each update notification. If the payload is a single file, that file is downloaded and, if marked for execution, is run to complete the update installation. If the payload is a manifest file, the file is downloaded and, in turn, each of the items in the manifest is downloaded. Those items marked for execution are then run to complete the product update installation.

  Currently, the notification server supports only a manifest file as the content type for an update (that is, its payload).

- `-packageid` identifies the product package for which you are requesting notifications. If this argument is not specified, the value defaults to `Test1`.

- `-productplat` identifies the platform on which the product package runs. Valid values include `WIN32` and `WIN64`. If this argument is not specified, the operating system of the client device is detected and sent in the request.

- `-productlang` specifies the Microsoft Language Locale Identifier (LCID) in decimal format for the language of the current product package. If this argument is not specified, the LCID defaults to `1033` (for U.S English).

- `-commproduct` creates a communications object at the product-package object level, enabling custom options to be set for communications with the notification server.

  If this argument is not used, communications are handled by a default communications object created in internally for the product package object.

- `-commdownload` creates a communications object for a given item marked as "update" in the notification collection, enabling you to set custom options for each for each item marked as "update" in the notification collection to download the payload for that item.

  If this argument is not used, communications are handled by default communications objects created internally for the "update" notification items.

- `-maxrate` sets the maximum rate for downloads in bytes per second across all communications objects created for those items marked as "update" in the notification collection. This argument automatically implies the `-commdownload` argument.

Confirmation Output

If the attempt to run the **Notification** example is successful, confirmation output similar to the following is displayed.(In this case, if the `-download` argument was used in the command.)

Note that the `'To' Product Package ID` value is the ID of the product package ID to which the existing product package is updating.

```
Registering client with activation id: ACT01-PhotoPrint.
Polling notification server for client registration result.
Client registration complete.

----------------------------------------------------------------------
  Product package id: PhotoPrintInstaller
  Product language:   1033
  Product platform:   WIN32
----------------------------------------------------------------------

Fetching notification collection for product package id: PhotoPrintInstaller.
Polling notification server for notification collection.
Polling notification server for notification collection.
Notification collection received.

1 notification items returned for requested product package id.

Notification item 1 of 1 attributes:
  Type:                          Update
  Notification ID:               PhotoPrint-Update-V12.1
  Product Package ID:            PhotoPrintInstaller
  'To' Product Package ID:       PhotoPrintInstaller-V12.1
  Notification Name:             PhotoPrint-Update
  Title:                         English
  Content Type:                  MANIFEST
  Description:                   English language
  Details:                       English language
  Download Type:                 PRESENT_FIRST
  Download URL:                  https://mytenant-ns-uat.flexnetoperations.com/manifests/c0b6059b-
                                 11ef-4339-8146-5b88b8d78928
  Download Size:                 1024
  Availability Date:             04/30/2018
  Expiration Date:               04/30/2019
  Elevation Required:            False

Downloading notification #1
```

```
Download started
Download ended
Execution started
Execution ended
Download successful to: C:\Users\JohnFrum\Downloads\c0b6059b-11ef-4339-8146-5b88b8d78928
SUCCESS: Notification application complete
```

Troubleshooting Build Errors for the "Notification" Example

If your `Notification` executable fails to launch or is unable to complete the notification and update process, consider the following:

- If an error beginning with `Failed to load FlxCore library` is displayed, or the application exits with a `FileNotFoundException` or `DllNotFoundException` message that refers to *FlxCore*, verify that the `FlxCore.dll` library has been properly installed, as described in Install the FlexNet Embedded Client Libraries for Updates and Insights.
- If an error containing this information is displayed, make sure you have provided the correct URL or port number for the notification server:

  ```
  ERROR: FlxDotNetClient.ServerResponseException encountered:
  Server response error.
  The remote name could not be resolved: 'some.bad.url'
  ```

- Checking for proxy servers and fire walls might resolve certain communications errors as well.
- Verify that the client identity (`IdentityClient.cs`) and the back-office identity (`IdentityBackOffice.bin`) have been generated from the same information in the `pubidutil` utility. Both identities must use the same producer name, producer keys, and signature algorithm and strength. If a discrepancy issue exists, the example prints exception information that the identities do not match.
- If launching the **Notification** example displays an error message such as `Notification.exe is not a valid Win32 application`, verify that the project was built with the correct configuration. For example, this error can occur when building the x64 configuration but executing the code on a 32-bit system.

Next Steps

This chapter has demonstrated simple scenarios for verifying that your toolkit is working properly. See the remaining chapters for information about the following:

- The FlexNet Embedded Client .NET XT toolkit directory structure, examples, API groups, and terminology
- Using FlexNet Embedded functionality to create and manage additional sources of license rights: demo licenses, replacement licenses (served to trusted storage or buffers), and certificate licenses
- Dynamically generating license rights with a back-office serve.
- Using the Updates and Insights functionality to obtain product updates from the notification server and download and install them
- Using utilities provided with the toolkit.

3

Quick Start with the .NET Core XT Toolkit

This chapter describes the basics needed to get started with the FlexNet Embedded Client .NET Core XT toolkit:

- Toolkit Requirements
- Downloading the Toolkit
- Creating the Producer Identity
- Building and Running the “BasicClient” Licensing Example
- Next Steps

The remaining chapters provide more information about the toolkit and delve into actual use of its FlexNet Embedded functionality.

Toolkit Requirements

For information about supported toolkit platforms and prerequisites for developing, building, and deploying your product with the toolkit, see the current version of the *FlexNet Embedded Client Release Notes*.

Downloading the Toolkit

The email you received from Revenera provides instructions for downloading the FlexNet Embedded Client .NET Core XT toolkit from the Product and License Center. These instructions can vary, depending on whether you are downloading a purchased toolkit or one that you are evaluating.

You have the option to download either `flexnet_client-xt-dotnet_core-`*`version`*`.zip` or `flexnet_client-xt-dotnet_core-`**`version.`**`tgz`, depending on the archive method most convenient for you. Once the toolkit archive is downloaded, you decompress it somewhere on your system.

In this book, the root directory of the decompressed toolkit is referred to as *`install_dir`*.

Creating the Producer Identity

As a producer, your downloaded FlexNet Embedded Client .NET Core XT toolkit is unique to your organization, in the sense that no organization's license-enabled code can use another organization's license rights.

Each producer is identified by a unique producer name and unique producer keys. To enable your .NET XT Core toolkit, you must generate *producer identity data* to be used by your back-end tools and by your license-enabled code. A set of producer identity data files contains a combination of cryptographic data and settings used to digitally sign your license rights on the back-office server and to validate these license rights messages sent to the client.

To generate your organization's identity files, use the `pubidutil` utility available in the toolkit. This utility can be used to generate identity information in a demo or production environment.

Note • *You typically use "pubidutil" to generate demo identity information but use FlexNet Operations to generate your producer identity data for production.*

The following sections describe how to create your producer identity:

- Create the Identity Binary Files
- Generate .NET-compatible Identity Data
- Distribute Identity Data

Create the Identity Binary Files

To create your producer-identity binary files, you need the demo keys or production keys specific to your organization, as provided in your email from Revenera. You also need access to the `pubidutil` utility (also called the *Publisher Identity utility*) provided in the FlexNet Embedded Client .NET Core XT toolkit.

Task ***To create the identity binary files for your organization***

1. Launch `pubidutil` by double-clicking *`install_dir`*`\bin\tools\pubidutil.bat` in a Windows environment (or *`install_dir`*`/bin/tools/pubidutil` in other environments) or by running the utility from a command window.

   The **Publisher Identity Utility** window opens.

2. Enter the following:

   - Names and locations where your server identity and client identity files will be created (or reopened, if the files already exist).
   - An **Identity Name**, which is a unique name—such as `demo-med-rsa`—for this collection of identity settings.
   - The **Publisher Name**—a case-sensitive value such as `demo`—and **Publisher Keys**—five hexadecimal numbers—all of which are provided in your email message from Revenera.
   - The desired digital signature type and strength. For this example, select `RSA` from the **Signature Type** options, and select `Medium (1024 bit)` and `SHA-256` from the **Signature and digest strength** options.

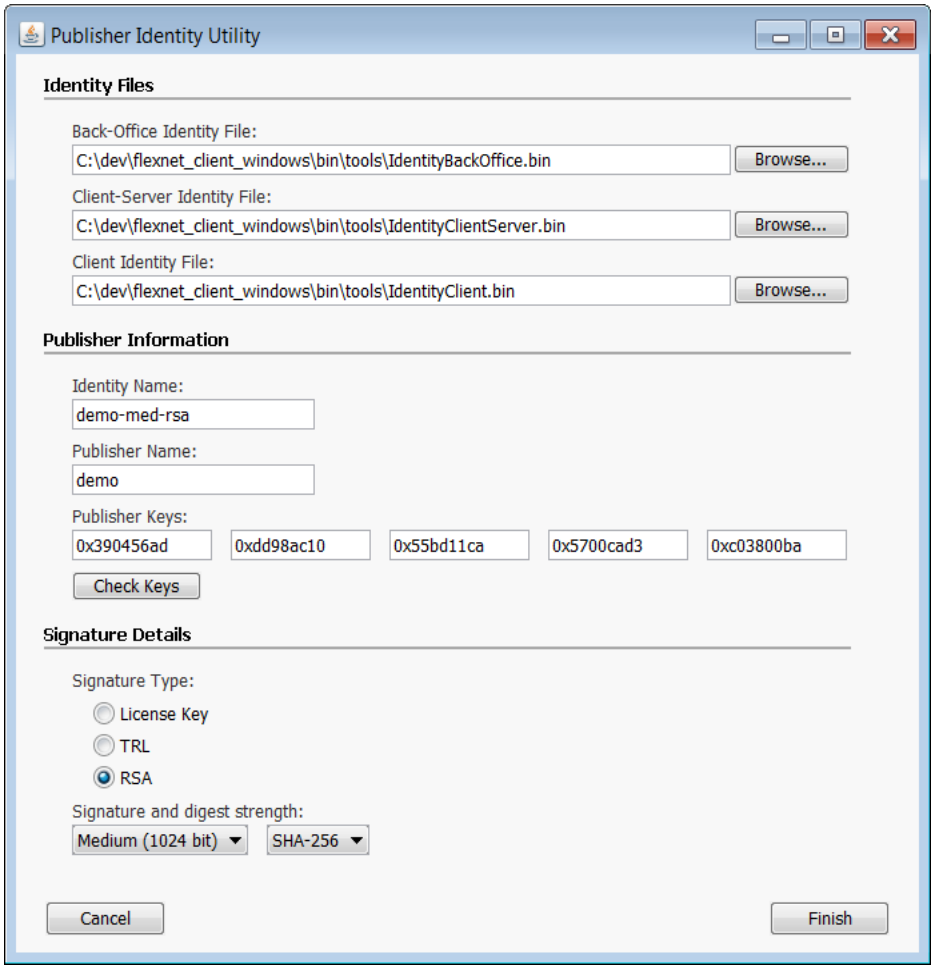


Figure 3-1: Entering Producer-specific Information for the Publisher Identity Utility

3. Click **Finish** when you have entered this information. The `pubidutil` utility creates the following identity files in the location you specified.

   - The *back-office* identity data—by convention called `IdentityBackOffice.bin`—is used by the back-office server to digitally sign license rights and must be kept secure. The back-office server is either FlexNet Operations or the test back-office server utility `capserverutil` (Capability Server utility).
   - The *client* identity data—by convention called `IdentityClient.bin`—is included in your license-enabled code in order to acquire license rights at run time.
   - The client–server identity data—`IdentityClientServer.bin`—is used when preparing a FlexNet Embedded license server, as described in the *FlexNet Embedded License Server Producer Guide*.

You can also run the Publisher Identity utility in text mode by opening a command prompt window to the *`install_dir`*`\bin\tools` directory and running the `pubidutil` script with the `-console` switch, entering information in the prompts that follow.

For information about updating producer identity data, see Update Identity Data.

Generate .NET-compatible Identity Data

The `pubidutil` utility generates your identity files in binary format. However, the toolkit API methods that initialize client-side identity information in your license-enabled assembly code take a C# byte array as an argument. To generate this byte array, use the `printbin` utility, found in the *`install_dir`*`\bin\tools` directory of your FlexNet Embedded Client .NET Core XT toolkit.

Run the following command:

```
printbin -cs IdentityClient.bin -o IdentityClient.cs
```

The output should look similar to the following:

```
using System;

/* ...comment listing signature name, type, and keys... */

namespace IdentityData
{
    internal static class IdentityClient
    {
        internal static readonly byte[] IdentityData = new byte[] {
            0x68, 0x61, 0x70, 0x70, 0x79, 0x20, 0x6c, 0x69, 0x63, 0x65,
            0x6e, 0x73, 0x69, 0x6e, 0x67, 0x21, 0x20, 0x2d, 0x72, 0x6f,
    ...
            0x62, 0x65, 0x72, 0x74, 0x64, 0x00};
    }
}
```

This array of bytes representing your client identity will be compiled into .NET assembly code. See Distribute Identity Data for information about where to put this file for inclusion in the build.

Caution • *For security reasons, it is strongly recommended that your client-identity data be embedded in your code in this fashion, as opposed to loading the binary identity data from an external file at run time.*

Distribute Identity Data

To prepare the client-identity data for use in .NET code, copy the file `IdentityClient.cs` into a directory where it can be accessed by and compiled into your executable code. (To run the FlexNet Embedded Client .NET Core XT examples, copy the identity data to the *`install_dir`*`\examples\identity` directory.)

Finally, provide the `IdentityBackOffice.bin` to the licensing back-office server—that is, FlexNet Operations or the test back-office server utility `capserverutil` (Capability Server utility).

Update Identity Data

In some cases you might need to update existing identity data with new information. For example, if you purchase additional platforms from Revenera and therefore receive new publisher keys, you need to update your identity data so that it reflects the new platform information. New identity data is always generated based on existing identity data. In other words, the publisher keys contained within the identity files are replaced with the new values. Other elements, such as signing and encryption keys, remain unchanged.

Using FlexNet Operations

Typically, you would use FlexNet Operations to update identity data for production systems. For information about how to do this, refer to the FlexNet Operations documentation. After you update identity data, you need to export the new client identity and embed it in any new clients. Existing clients in the field do not need to be updated if only publisher keys have been updated.

Using pubidutil

If you want to generate new identity data for testing purposes, you can use the `pubidutil` utility (either in command-line or GUI mode). To ensure that `pubidutil` generates new identity data that is compatible with the previous set of identity files, you need to specify the existing back-office identity file when running `pubidutil`.

To update identity data using `pubidutil` in UI mode, follow the instructions in section Create the Identity Binary Files. Make sure that you use all original settings (identity files, identity name, publisher name, and signature details), with the exception of the publisher keys which you need to replace with the new keys that you received from Revenera.

To update identity data using `pubidutil` in console mode, run the `pubidutil` script in the `bin/tools` subdirectory of the toolkit with the `-console` switch:

```
pubidutil -console
```

You will prompted for all of the required information at the command line, with the previous identity information provided as default values. Press Enter to accept the default values, except for the publisher keys. When prompted, enter the new publisher keys.

Next Steps

Follow the instructions in the sections Generate .NET-compatible Identity Data and Distribute Identity Data to update client and server components with the new identity data.

Building and Running the "BasicClient" Licensing Example

This section walks you through the process of building and running the **BasicClient** example found in the FlexNet Embedded Client .NET Core XT toolkit. The process is geared toward those producers who want to observe how a product license works.

The **BasicClient** example is the simplest example of code that uses FlexNet Embedded APIs to enable licensing. This example acquires license rights from a local binary file and prints a message if the license acquisition succeeds. The following sections walk you through **BasicClient** build and execution process:

- Phase 1: Provide the Prerequisites for "BasicClient"
- Phase 2: Prepare to Build "BasicClient" Executable
- Phase 3: Build and Run the "BasicClient" Example

Phase 1: Provide the Prerequisites for "BasicClient"

Before building and running the `BasicClient` example, ensure that the prerequisites are in place:

- Client Identity File: Copied to Proper Location
- License Rights: Created and Copied to Proper Location
- .Net Core: Installed on the Target Machine

Client Identity File: Copied to Proper Location

Verify that the `IdentityClient.cs` file (built using steps in Creating the Producer Identity) resides in the *`install_dir`*`\examples\identity` directory.

License Rights: Created and Copied to Proper Location

License rights used by license-enabled code are specific to each producer, which means that a producer's license-enabled code can work with only that producer's licenses. One way to store license rights on a client is in a digitally signed binary file, which you can create based on an unsigned text representation of the license rights. The following procedure describes how to create a binary license file for the **BasicClient** example.

Task ***To create the binary license file for the BasicClient example***

1. Create a text file called `license.txt` with the following contents:

   ```
   INCREMENT survey demo 1.0 1-jan-2025 uncounted HOSTID=ID_STRING=1234567890
   INCREMENT highres demo 1.0 1-jan-2025 uncounted HOSTID=ID_STRING=1234567890
   ```

   Each line in the text representation of license rights is made up of the following components. (For details about feature definitions, see Feature Definitions.)

   - The names after the `INCREMENT` keyword (`survey` and `highres`) are your *feature names*; at run time your license-enabled code attempts to acquire license rights for those features, and reacts accordingly based on whether valid rights are available.
   - Each feature is tied to a particular *producer name*: `demo` for the demo toolkit, and your producer name for a production toolkit.
   - Each feature is versioned; the license-enabled code requests a particular version, and if the version in the license rights is greater than or equal to the requested version, the request succeeds.
   - Each feature has an expiration date; if the license rights have expired, the attempt to acquire the license will fail.
   - A feature is tied to a particular client using a `HOSTID` value. In the default toolkit examples, clients are assumed to have a hard-coded string identifier "0123456789", while in practice your code specifies the desired type of client identifier (such as an Ethernet address) to examine at run time in order to compare it with the identifier in the license rights. For more information, see Hostids.

2. Use the `licensefileutil` utility to digitally sign the license rights so that only your license-enabled code can acquire the licenses. From the *`install_dir`*`\bin\tools` directory, run the following command in a console window:

   ```
   licensefileutil -id IdentityBackOffice.bin license.txt license.bin
   ```

   The output, `license.bin`, is a binary representation of your license rights that can be acquired by license-enabled code at run time. This file must be placed in a location where your license-enabled .NET code can read it.

3. So that `license.bin` resides in a place where the .NET code for **BasicClient** can read it, copy the file to the *`install_dir`*`\examples\client_examples\BasicClient` directory (where the **BasicClient** project file is located.)

.Net Core: Installed on the Target Machine

Microsoft .NET Core needs to be installed on the machine on which you are building and executing **BasicClient**. Refer to the current *FlexNet Embedded Client Release Notes* for the list of .NET Core framework versions that FlexNet Embedded supports.

If you need to download and install .NET Core, you can use the following link:

https://www.microsoft.com/net/core#windowsvs2017

Phase 2: Prepare to Build "BasicClient" Executable

Use the following steps to prepare to build the **BasicClient** executable:

- Step 1: Ensure Project File Points to Correct .NET Core Framework Version
- Step 2: Run "restore" to Obtain Latest Packages
- Step 3: Copy "FlxCore" to the Project Folder

Step 1: Ensure Project File Points to Correct .NET Core Framework Version

By default, the project file for the `BasicClient` points to a specific .NET Core framework target to build and run `BasicClient`. You need to review this file and make any necessary edits to ensure that it specifies the .NET Core framework version installed on your machine, as described in .Net Core: Installed on the Target Machine.

Task ***To ensure that the project file points to the correct .NET Core version***

1. Navigate to the *`install_dir`*`\examples\client_samples\BasicClient` directory, and open the project file `BasicClient.csproj` in a text editor.
2. If necessary, edit the contents to ensure that the project points to the correct .NET Core framework version installed on your machine.

   You have the option to point to multiple target frameworks. If you do so, all specified targets are compiled when you build the project. However, you will need to designate the specific target under which to execute the assembly (described later in Phase 3: Build and Run the "BasicClient" Example).

Step 2: Run "restore" to Obtain Latest Packages

A "dotnet restore" process is necessary to ensure that your machine has the latest packages required to build and run the `BasicClient` example. In addition to installing required .NET Core packages, this process also installs the required FlexNet Embedded component `flexera.flxlicensingclient.core` if it is missing.

Task ***To run the "dotnet restore"***

At a command prompt, navigate to the *install_dir*\examples\client_samples\BasicClient directory (that is, the location of the project file), and run the following:

```
dotnet restore
```

This process installs the necessary packages on your machine. Typically, this is in the current user's .nuget directory.

Step 3: Copy "FlxCore" to the Project Folder

FlexNet Embedded makes use of FlxCore native component for some of its licensing functionality. You must extract the appropriate FlxCore component in your toolkit and copy it to a location accessible by the BasicClient example. The **FlxCore** native components are stored under the FlxCore folder within the Flexera.FlxLicensingClient.core.2017.11.0.nupkg archive, which is located in the *install_dir*\lib directory.

Task ***To locate and copy the native FlxCore component***

1. Within the Flexera.FlxLicensingClient.core.2017.11.0.nupkg archive, navigate to the FlxCore directory to locate the component appropriate for the operating system on which your building and executing the example.

   The following are the available FlxCore native components:

   - libFlxCore.so.*version* (for i86 Linux)
   - libFlxCore64.so.*version* (for x64 Linux)
   - FlxCore.dll (for i86 Windows)
   - FlxCore64.dll (for x64 Windows)
   - libFlxCore.*version*.dylib (for OS X 10)

   Important • *Always use the version of the "FlxCore" library that matches the version of FlexNet Embedded used in your implementation.*

2. Extract the FlxCore component and copy it to the location of the project file. For **BasicClient** this is the *install_dir*\examples\client_samples\BasicClient directory.

Phase 3: Build and Run the "BasicClient" Example

The following procedure describes how to build and run the **BasicClient** example.

Note • *While the procedure described here uses a command line, you can also build the example using Microsoft Visual Studio. See the current FlexNet Embedded Client Release Notes for the Visual Studio versions that support this process.*

Task ***To build and run the* BasicClient *example***

At a command prompt, do one of the following, based on your .NET Core target specification in the project (see Step 1: Ensure Project File Points to Correct .NET Core Framework Version):

- If you pointed to a single .NET Core framework target in the project, run the following command from the *install_dir*\examples\client_samples\BasicClient directory to compile and run the example run using that framework version:

  ```
  dotnet run
  ```

- If you specified multiple .NET Core framework targets, use the `-f` argument to specify the framework version under which to execute `BasicClient`. (Note that all framework targets that you specify in the project are compiled.) For example, to run the example under the `netcoreapp2.0` target (that is, .NET Core 2.0), you would enter the following:

  ```
  dotnet run -f netcoreapp2.0
  ```

When the example runs, it acquires license rights from the license binary file you created and prints a message if the license acquisition succeeds.

Troubleshooting Compilation or Execution Errors for the BasicClient Example

The following sections provide some troubleshooting suggestions for errors that occur during the build or execution of the **BasicClient**.

Compilation Errors

Your **BasicClient** project might fail to compile with the following error:

```
License-enabled code requires client identity data, which you create with pubidutil and printbin -cs
or with FlexNet Operations
```

To troubleshoot this problem, make your sure you have done the following:

- Copied the C#-compatible identity file that you created with `pubidutil` and `printbin` into `examples\identity`.
- Named the identity file `IdentityClient.cs`.

Execution Errors

If your `BasicClient` executable fails to launch or is unable to acquire any licenses, try these troubleshooting methods:

- If an error beginning `Failed to load FlxCore library` is displayed, verify that the `FlxCore` native component has been properly installed, as described in Phase 2: Prepare to Build "BasicClient" Executable.
- If an error message `ERROR: Unable to find file license.bin` is displayed, verify that the `BasicClient` executable was able to find the binary license rights file `license.bin`. You can specify the path to the buffer license file as a command-line argument to `BasicClient`.
- If an error message `Data version is not supported` is displayed, verify that the `BasicClient` executable is loading a signed binary license file, and not (for example) a text file or other file type.

- Verify that the client identity—that is, the array of bytes in `IdentityClient.cs` created from `IdentityClient.bin` using `printbin`—and the back-office identity, `IdentityBackOffice.bin`, used when digitally signing license rights were generated from the same information in the `pubidutil` utility. Both identities must be generated from the same producer name, producer keys, and signature algorithm and strength. If this is the issue, the example prints exception information (`ERROR: Signature didn't pass validation`) that the identities do not match.
- The **View** example that ships with the toolkit will display a simple diagnostic report of the license rights contained in a binary license file. If the `BasicClient` executable is unable to acquire licenses, pointing the **View** example to the binary license file may indicate issues with the original license's syntax or attributes.

Next Steps

This chapter has demonstrated simple scenarios for verifying that your toolkit is working properly. See the remaining chapters for information about the following:

- The FlexNet Embedded Client .NET Core XT toolkit directory structure, examples, API groups, and terminology.
- Using FlexNet Embedded functionality to create and manage additional sources of license rights: trial licenses, replacement licenses (served to trusted storage or buffers), certificate licenses, and other licenses.
- Dynamically generating license rights with a back-office server.
- Using utilities provided with the toolkit.

4

Toolkit Overview

Both the FlexNet Embedded Client .NET XT toolkit (which uses the .NET Framework) and the FlexNet Embedded Client .NET Core XT toolkit (which uses .NET Core) provide a collection of libraries, example source code, and utilities used by a producer to enable FlexNet Embedded licensing functionality in your product code. The FlexNet Embedded Client .NET XT additionally supports Updates and Insights functionality to provide notification services.

This chapter provides basic information to help you get started with either toolkit:

- Concepts: Licensing and Updates Functionality
- Toolkit Requirements
- FlexNet Embedded Client .NET XT Toolkit Contents
- FlexNet Embedded Client .NET Core XT Toolkit Contents
- About the Example Projects
- Building and Running the Examples in the .NET XT Toolkit
- Building and Running Examples in the .NET Core XT Toolkit
- Toolkit Files to Distribute with Your Product

Concepts: Licensing and Updates Functionality

This section explains basic terminology and concepts used in this guide to describe FlexNet Embedded licensing and Updates and Insights functionality:

- FlexNet Embedded Concepts
- Concepts of Updates and Insights

The following are some of the concepts and terminology used throughout this documentation.

FlexNet Embedded Concepts

The following are some of the FlexNet Embedded concepts and terminology used throughout this documentation:

- Hostids
- Feature Definitions
- Back-office Servers and License Servers
- Trusted Storage
- Capability Requests and Responses

Hostids

FlexNet Embedded uses system identifiers, called hostids, for identification and license locking. A feature is tied to a particular client using a hostid value which is specified in the license file. When the client requests a license for a feature from the license server, the client includes a hostid in the request to which the license server binds the licenses sent in the response.

This section focusses on client hostids for the .NET XT SDK. For information about hostids for other SDKs, refer to the user guide for the respective client kit. For information about server hostids, refer to the *FlexNet Embedded License Server Administration Guide*.

Hostid Types

Supported hostid types include:

- String hostid, typically used in testing.
- Ethernet (MAC) address
- IPv4 address
- IPv6 address
- Aladdin dongle (`flexid9`)
- Wibu-Systems dongle (`flexid10`)
- UUID of a supported virtual machine
- Container ID of a supported containerization technology.

Hostid Keywords in Feature Definitions

Each feature definition specifies the hostid to which the license is node-locked. The hostid should be expressed as `HOSTID=`*`type=zzz`*, where *`type`* is a keyword indicating the hostid type and *`zzz`* is a string of UTF-8 characters. (An exception is the special expression **`HOSTID=ANY`**, which matches any client system, and which is commonly used with served licenses.) For more detailed information about feature definitions, see the following section, Feature Definitions.

Most of the hostid comparisons are not case sensitive, with the exception of STRING (for example, **`HOSTID=ID_STRING=AAAAA`** is different from **`HOSTID=ID_STRING=aaaaa`**).

The following table lists the keywords to use for different types of hostids in the text license file and when creating license rights using the `licensefileutil`, `capresponseutil`, and other testing and development tools:

Table 4-1 • Supported Hostid Types and their Keywords

Hostid Type	Keyword	Example	Case Sensitive
String	`ID_STRING`	`HOSTID=ID_STRING=12345ABcde`	Yes
Ethernet (MAC) address	This type of hostid does not use a keyword between `HOSTID=` and the hostid value.	`HOSTID=0037c0b82e82`	No
Internet (IPv4) address	`INTERNET`	`HOSTID=INTERNET=11.22.33.44`	No
Internet (IPv6) address	`INTERNET6`	`HOSTID=INTERNET6=2001:0db8:0000:0000:ff8f:effa:13da:0001`	No
Aladdin dongle	`FLEXID9`	`HOSTID=FLEXID=9-566d9316`	No
Wibu-Systems dongle	`FLEXID10`	`HOSTID=FLEXID=10-0bebc202`	No
UUID of a supported virtual machine	`VM_UUID`	`HOSTID=VM_UUID=AAAAAAAA-BBBB-CCCC-DDDDEEEEEEEEEEEE`	No
Container ID of a supported containerization technology	`CONTAINER_ID`	`HOSTID=CONTAINER_ID=adbdac367028`	No

Feature Definitions

At run time, license-enabled code attempts to acquire one or more licenses, the presence or characteristics of which enable a certain capability, capacity, or configuration. Licenses are acquired from license sources, which in turn contain license rights. License rights, in turn, are made up of one or more features, each with a feature definition.

The following provides details about features:

- Feature Definition Syntax
- Required Feature Fields
- Optional Feature Keywords
- More About Feature Processing

Feature Definition Syntax

Each feature definition in an unsigned set of license rights uses the following format:

`INCREMENT` *`name producername version exp count`* `[`*`optional keywords`*`] HOSTID=`*`type=zzz`*

Toolkit utilities convert the unsigned text license representation into a digitally signed binary representation that can be consumed by license-enabled code.

Required Feature Fields

The following shows an example feature definition:

```
INCREMENT lights demo 1.0 1-jan-2025 uncounted HOSTID=ID_STRING=Bldg123 \
   START=1-jan-2013 VENDOR_STRING="Hello, World!"
```

The first six fields (the `INCREMENT` keyword through the *`count`* field) in an unsigned feature definition must occur in this order; the `HOSTID` and optional keywords can occur in any order.

The following describes the required fields:

- Feature names used in FlexNet Embedded Client C XT are case sensitive. For example, a feature named `F1` is different from one named `f1`.

  Caution • *A licensing keyword—START, NOTICE, and so forth—cannot be used as a feature name. Moreover, there are some reserved words that cannot be used as feature names, including CAPACITY, PACKAGE, SUPERSEDE, and UPGRADE.*

- The "demo" version of the FlexNet Embedded Client .NET XT or .NET Core XT toolkit uses the **`demo`** producer name (used for evaluation purposes only). The producer name is case-sensitive.

- Feature versions must use the format *a.b*, where *a* and *b* are numbers. The integer part *a* can be a value from 0 through 32767, and the fractional part *b* can be a value from 0 through 65535. Version comparisons are performed field by field: 2.0 is a greater version than 1.5; 1.10 is a greater version than 1.1; 1.01 and 1.1 are considered equal versions.

- Expiration dates in a feature definition must be expressed using the format *`dd-mmm-yyyy`*, where *`mmm`* is the first three letters of the English month name (`jan`, `feb`, `mar`, and so forth). To indicate a license that does not expire, the expiration date can be expressed as **`permanent`** or as a date with year zero, such as **`1-jan-0`**.

- The count value **`uncounted`** (or value **`0`**) is used for uncounted node-locked licenses. Counted licenses, whether to be used on a client or served by a license server, use a positive integer count. (The count value **`2147483647`** is also treated as uncounted.)

- The hostid to which the license is node-locked should be expressed as `HOSTID=`*`type=zzz`*, where *`type`* is a keyword indicating the hostid type and *`zzz`* is a string of UTF-8 characters. See Hostids for a list of hostid keywords. You can express multiple hostids using the space-separated hostid format **`HOSTID="ID_STRING=A1 ID_STRING=B2"`**. However, a hostid itself cannot contain a space character. See Specifying the Hostid Type to Use in the *Using* FlexNet Embedded *APIs* chapter for details.

Optional Feature Keywords

The following optional keywords can be used in a feature definition, in the form *`KEYWORD=value`*. Keyword values containing spaces must be surrounded with quotation marks, as in **`NOTICE="For the use of Example Customer"`**.

Except for the date-related `ISSUED` and `START` keywords, the keyword values can be arbitrary UTF-8 text. (If you include UTF-8 characters in license keyword values and use the FlexNet Embedded Client C XT utilities such as `licensefileutil` to generate binary license rights, your unsigned text license file must be saved in UTF-8 format.)

- `ISSUED`: Date the license was issued, in *`dd-mmm-yyyy`* format.

- `ISSUER`: Organization that issued the license.

- NOTICE: Commonly used to store intellectual property notices.
- SN: Used to store a serial number value for the license.
- START: Date the feature becomes active, in *dd-mmm-yyyy* format.
- VENDOR_STRING: Arbitrary producer-defined license data, such as feature selectors (described in Feature Selectors in a Capability Request in the *Using* FlexNet Embedded *APIs* chapter for details):

  ```
  VENDOR_STRING="%%KEY:VALUE[, KEY:VALUE, ...]%%"
  ```

  For example, the following shows two feature selectors defined as the VENDOR_STRING value:

  ```
  VENDOR_STRING="%%DEPT:ACCT, ROLE:AUDIT%%"
  ```

More About Feature Processing

The toolkit provides utilities—including `licensefileutil` and the test back-office server utility `capserverutil`—for converting unsigned feature definitions into the digitally signed binary format used by license-enabled code. The licenses are digitally signed using the digital signature algorithm and key size you specified when generating your identity data.

During execution, your license-enabled code contains functions that attempt to acquire a license. The FlexNet Embedded libraries validate such conditions as the expiration date not having arrived and the feature's hostid matching the current client system's hostid. If the license acquisition succeeds, your code would then enable the corresponding capability.

In your license-enabled code, you can additionally read the values of any license fields (such as VENDOR_STRING) and use them for any desired purpose.

Back-office Servers and License Servers

Two different categories of "server" used with FlexNet Embedded functionality are *back-office servers* and *license servers*.

The back-office server used for licensing purposes is integrated with the producer's back office. FlexNet Embedded uses FlexNet Operations (a separately purchased product) as the back-office server. In a typical licensing scenario, FlexNet Operations receives capability requests from client systems and sends back capability responses that install or update license rights in the client's trusted storage. As a back-office server, FlexNet Operations implements business logic that determines what license rights a particular client is entitled to receive.

A license server, on the other hand, is a system used to manage a counted pool of licenses for a single customer network. This type of license server can either reside at an enterprise customer site (called a FlexNet Embedded *local license server*) or be a CLS (Cloud Licensing Service) instance, and will typically be integrated with a provisioning or configuration system to manage planned or dynamic deployment scenarios on the customer's network.

A common situation where the two types of servers interact is when the license server at a customer site receives a pool of licenses from the producer's back-office server. The license server sends a capability request to the back-office server, which in turn responds with a capability response that places license rights in the server's trusted storage. The license server can then serve the pool of licenses to client systems.

The license servers for FlexNet Embedded include the FlexNet Embedded local license server, described in the *FlexNet Embedded License Server Producer Guide*, and the CLS (Cloud Licensing Service) license server, described in section "Getting Started with Cloud Licensing Service" in the *FlexNet Operations User Guide* and in the *FlexNet Embedded License Server Producer Guide*.

Note • *For simplicity, this user guide uses the term "FlexNet Embedded license server" or simply "license server" to refer to both the local and the cloud license-server types. For areas of FlexNet Embedded functionality that support one or the other license-server type, the documentation notes this as such.*

Trusted Storage

In addition to storing license rights in binary license files, some types of license rights are stored in *trusted storage*. Trusted storage is a secure location bound to a particular client system. The FlexNet Embedded libraries implement file-based and memory-based trusted storage options. For more information about configuring trusted storage in your code, see Creating Core Licensing Objects.

In addition to its contents being encrypted, a security feature of trusted storage is *anchoring*, which provides trust that trusted storage has not been deleted or rolled back to an earlier state on the same system. This prevents a system from restarting a trial license if trusted storage is deleted, for example. To achieve this, anchoring stores a small amount of data in a location that is difficult to observe or access.

If anchoring reports an inconsistency when trusted storage is accessed, the FlexNet Embedded run-time reports a "break", which indicates a breach of trust, and you can decide the action to take in such cases.

Trusted storage also provides the means for obtaining metered licenses, as described in the **UsageCaptureClient** example.

Similar to starting the process of creating binary license rights with a text license file, the `trialfileutil` utility enables you to generate binary trial license rights based on an unsigned text license file along with additional trial-related attributes.

Characteristics of trial license rights include:

- The duration of the trial period (10 days from the time the application is first launched, for example)
- The features included in the trial (using the same syntax described earlier in this chapter)
- Various identifiers for the trial

The **Trials** example in the toolkit shows how to process binary trial data, and then how to acquire license rights from the trial license source.

Capability Requests and Responses

When a client communicates with a back-office server or a license server to install or update dynamic license rights, the communications involve *capability requests* and *capability responses*.

Standard Request-Response Process

A capability request is generated by the client (either a client directly requesting features that it will acquire, or a license server requesting a pool of features from the back office to serve to client). The request data contains some combination of a host identifier, one or more rights identifiers, and any other producer-defined data to pass to the back-office server or the license server.

The back-office server or license server then processes the capability request, reading the various identifiers and custom data. If the server determines that the requested licenses are available to the client, it generates a capability response. The response data contains current license rights available for the client. The capability response is then conveyed back to the client, which processes the response, after which the license rights are stored in trusted storage and can be acquired by

your license-enabled product code. Just as licenses are digitally signed using the digital signature algorithm and signature strength you selected when generating identity data, capability responses are digitally signed to prevent tampering and detect corruption.

Any previous license rights in trusted storage are overwritten with the data from the new capability response. For this reason, trusted storage rights are sometimes called *regenerative* license rights or *replacement* license rights.

Communications with the Back Office or License Server

With FlexNet Embedded functionality, there is no requirement that the client system communicate directly with a back-office server or license server. As an alternative to direct communication, the capability request can be generated on the client and then exported as a binary file to be conveyed to the server. Similarly, the server's capability response can be generated as an external file, which will then be conveyed to the client for processing.

Concepts of Updates and Insights

The following sections describe common concepts and terminology pertaining to the Updates and Insights functionality:

- Updates and Insights Client
- Notifications
- Producer Site and Portal
- Notification Server

Updates and Insights Client

The Updates and Insights client is functionality that you integrate in your application code to enable your application for notification operations. Using a set of published interfaces, the client requests product notifications from the Updates and Insights notification server (see Notification Server) and, once the client receives the available notifications from the server, can respond to them. For example, if your application logic decides to install an update specified in a notification, the Updates and Insights client provides an appropriate method to download the update installer.

If your application (or your end user) chooses to launch an update, the Updates and Insights client provides the flexibility to use the toolkit's communications (cURL) implementation or your own appropriate method to download the file or files needed to install the update. If your customer loses the connection during the download, the client can prompt to reconnect and resume the download where it left off. The client can then execute any files needed to complete the installation.

- Downloads from HTTP, HTTPS, or FTP
- HTTP or HTTPS protocol for communication
- Proxy servers (the client is not affected by firewalls that allow Web-browsing—that is, allow Internet HTTPS requests)

The Updates and Insights APIs are flexible enough to allow your application to respond to the retrieved notification information silently or with your own visual interface that lets your end users see the messages or updates and act on them.

Notifications

The term *notifications* (also called *notification items*) collectively refers to product update notifications and other messages that you can deliver to your product end-users using Updates and Insights client code in your application.

Currently, FlexNet Operations supports only update notifications to product end-users (also called Updates and Insights *clients*). Future releases will extend this support to include functionality that collects data from client environments.

Producer Site and Portal

Your product updates are published on your FlexNet Operations producer site hosted in the Revenera Cloud. The site is accessed through the Producer Portal and contains information about your software applications and versions, customer entitlements to these applications, and the available updates to the applications.

When you want to publish a notification about a product update, the Producer Portal steps you though a series of forms—which prompt you for information about the update, the location of the files pertaining to the update, and the products or versions that will receive the update. Once an update is published, its notification is available to those Updates and Insights clients that request notifications for the products and versions affected by the update.

For more information about the publishing product updates, access the *FlexNet Operations User Guide* that is available in the Producer Portal.

Notification Server

The Updates and Insights notification server is a Revenera-hosted application that handles all requests from your Updates and Insights clients to retrieve notification information about your products. When an Updates and Insights client requests notifications for a given product package, the server collects the available notifications from FlexNet Operations and delivers them to the requesting client. Additionally, the notification server caches this collected notification information to minimize database queries and improve scalability.

The notification server also maintains a history of all client device activity pertaining to notifications, such as the download and installation progress of product updates and the success and failure statuses of these updates. You can view this activity from the Devices view in the Producer Portal. Future releases will provide facilities to extract this history and use it for reporting purposes.

Toolkit Requirements

For information about supported toolkit platforms and prerequisites for developing, building, and deploying your product with the toolkit, see the current version of the *FlexNet Embedded Client Release Notes*.

FlexNet Embedded Client .NET XT Toolkit Contents

The `.zip` file for the FlexNet Embedded Client .NET XT toolkit that you downloaded should be extracted onto your development environment system. Throughout this documentation, the directory into which you extracted the toolkit is referred to as *`install_dir`*.

The toolkit contains the following directories:

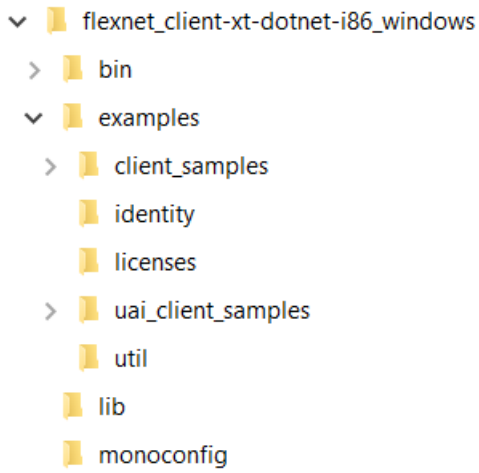


Figure 4-1: Directories in the FlexNet Embedded Client .NET XT Toolkit

The contents of the toolkit are organized in the following directory structure:

Table 4-2 • FlexNet Embedded Client .NET XT Toolkit Directories

Directory	Contents
bin	The FlexNet Embedded Client .NET XT development and testing utilities. The `bin` directory contains the following subdirectories: • `tools`: Contains containing command-line development and testing utilities • `demo\toolbox`: Contains the .NET Toolbox, a graphical utility for exploring various licensing scenarios and processes, including license examination, server communications, and feature acquisition. This directory also contains the `tra-gen.exe` file used when enabling Tamper Resistant Application (TRA) technology in your application code. See *FlexNet Embedded TRA Getting Started Guide for .NET XT* .
examples	The files needed to build and run the examples in the FlexNet Embedded Client .NET XT toolkit to illustrate various FlexNet Embedded and Updates and Insights scenarios and capabilities. The `examples` directory contains the following subdirectories: • `client_samples`: C# source code and build files for example applications that demonstrate licensing scenarios using FlexNet Embedded functionality. • `identity`: C#-compatible client identity information used by the toolkit samples. • `licenses`: Example unsigned license files used by FlexNet Embedded examples. • `uai_client_samples`: C# source code for projects that demonstrates notification operations using Updates and Insights functionality. • `util`: Common files used by the example source code.
lib	The FlexNet Embedded client .NET assembly and native libraries used in license-enabled and updates-enabled code.

Table 4-2 • FlexNet Embedded Client .NET XT Toolkit Directories

Directory	Contents
monoconfig	Files needed to configure your application for the Mono platform.

FlexNet Embedded Client .NET Core XT Toolkit Contents

The `.zip` or `.tgz` file for the FlexNet Embedded Client .NET Core XT toolkit that you downloaded should be extracted onto your development environment system. Throughout this documentation, the directory into which you extracted the toolkit is referred to as *`install_dir`*.

The toolkit contains the following directories:

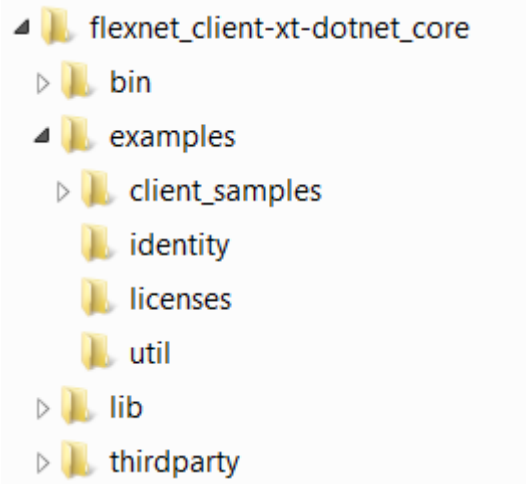


Figure 4-2: Directories in the FlexNet Embedded Client .NET Core XT Toolkit

The contents of the toolkit are organized in the following directory structure:

Table 4-3 • FlexNet Embedded Client .NET Core XT Toolkit Directories

Directory	Contents
bin	The FlexNet .NET development and testing utilities (found in the `bin` directory). The various command-line tools create binary producer identity information, generate different types of license rights, and so forth.

Table 4-3 • FlexNet Embedded Client .NET Core XT Toolkit Directories

Directory	Contents
examples	The files needed to build and run the examples in the FlexNet Embedded Client .NET Core XT toolkit to illustrate various FlexNet Embedded scenarios and capabilities. The `examples` directory contains the following subdirectories: • `client_samples`: C# source code and build files for example applications that demonstrate licensing scenarios using FlexNet Embedded functionality. • `identity`: C#-compatible client identity information used by the toolkit samples. • `licenses`: Example unsigned license files used by FlexNet Embedded examples. • `util`: Common files used by the example source code.
lib	The `Flexera.FlxLicensingClient.core.version.nupkg` archive, containing the .NET Core assembly and native libraries used in license-enabled code.
thirdparty	The root certificate your code might need to access SSL communications with FlexNet Operations. Should the certificate be needed, your code can install according to operating system requirements.

About the Example Projects

A *producer name* (“demo” in the evaluation toolkit) and *producer keys* are provided to build and run the example executables. Keep in mind that once you start creating your own license-enabled code, you will need your specific producer name and keys from Revenera.

FlexNet Embedded Examples

The **BasicClient** example, previously described in the Quick Start with the .NET XT Toolkit and the Quick Start with the .NET Core XT Toolkit chapters, is a minimum-dependency example that you can use to verify that the basic system functions as expected.

The other FlexNet Embedded examples listed here illustrate the different ways an application can acquire license rights, such as from a binary license file, a binary trial definition, or a license certificate. The examples also demonstrate how to use trusted storage to obtain licenses from a back-office server, such as FlexNet Operations, or a license server using a capability request. (The source code for these FlexNet Embedded examples is found in the *`install_dir`*`\examples\client_samples` directory.)

The following are the available FlexNet Embedded examples:

- The **Client** example illustrates how license-enabled code creates various license sources and then attempts to acquire features from the license sources. When license acquisition succeeds, the code illustrates how to obtain details of the license, such as its version, expiration date, and other attributes.
- The **CapabilityRequest** example illustrates how FlexNet Embedded code generates a capability request to send to a back office or a license server, sends the request, and then processes the server’s response into trusted storage. The example then illustrates how the client acquires the licenses from trusted storage. It also demonstrates how the client

can obtain a preview of available features on a license server without updating client trusted storage or changing the license server state.

- The **Trials** example illustrates how license-enabled code processes binary trial data that it stores in trusted trials storage. Once stored, the trial license rights can be acquired by the license-enabled code for a specified duration.
- The **UsageCaptureClient** example uses capability requests to send feature-usage data from a client to the FlexNet Embedded local license server or a CLS (Cloud Licensing Service) license server in a metered license model. The example supports scenarios for handling uncapped and capped feature usage.
- The **View** example illustrates how to use the diagnostic functionality in the FlexNet Embedded API to examine license rights contained in a binary license file and in trusted storage.

Tip • *The pre-built .NET Toolbox provides functionality equivalent to many of the example projects, which is useful for exploring licensing processes and capabilities. For details, see .NET XT Toolbox. (The .NET Toolbox is available in the FlexNet Embedded Client .NET XT toolkit only, not the .NET Core XT toolkit.)*

Updates and Insights Examples

The **Notification** example, previously described in the Quick Start with the .NET XT Toolkit chapter, is a minimum-dependency example that you can use to verify that a simple notification process functions as expected. The example retrieves product messages and update notifications from the Updates and Insights notification server and then downloads and executes an update on the client. You can find this example in the *install_dir*\examples\uai_client_samples.

Building and Running the Examples in the .NET XT Toolkit

The following sections describe the basics for building the examples projects included in the FlexNet Embedded Client .NET XT toolkit:

- Obtaining Producer Identity Data
- Building the Examples
- Running the Examples

Obtaining Producer Identity Data

Each producer is identified by a unique producer name and producer keys. To enable your FlexNet Embedded Client C XT toolkit, you must generate *producer identity data* to be used by your back-end tools and by your client code. For more information about generating your identity information and distributing appropriately, see Creating the Producer Identity in the *Quick Start with the .NET XT Toolkit* chapter.

You can also find instructions for creating producer identity data in the Publisher Identity Utility section of the *Utility Reference* chapter.

***Note** • The "demo" vendor keys expire after the evaluation period has elapsed. Contact Revenera sales if your evaluation keys have expired.*

Ensure that the client-identity information is installed in a location accessible by your application code during compilation. (For building the toolkit examples, copy the `IdentityClient.cs` file to the *`install_dir`*`\examples\identity` directory.) Then use the following procedures to build the example projects.

Building the Examples

To build the examples in a Windows version of the FlexNet Embedded Client .NET XT toolkit, repeat these steps for each example:

1. Access the appropriate *`install_dir`*`\examples\[uai_]client_samples\`*project* directory, where *project* is the name of the folder containing the example's project (for example, the `Notification` folder contains the project for the **Notification** example).
2. Open the project file (such as `Notification.vcproj`) in Visual Studio.
3. In Visual Studio, pull down the **Build** menu and select **Build Solution** (saving the solution file, such as `Notification.sln`, when prompted).

When the build finishes, the executable will be available in the `Debug` subdirectory relative to the project file. See Quick Start with the .NET XT Toolkit for additional build settings and troubleshooting information.

Running the Examples

The toolkit examples, which are console executables, are described in the following sections:

- Displaying Usage Help for an Example
- Running the FlexNet Embedded Examples
- Running the Updates and Insights Example

Displaying Usage Help for an Example

To display help information for any example, launch the executable with the -h or -help switch, as shown here for the client executable:

```
Client.exe -help
```

A usage message similar the following displays:

```
USAGE:


Client binary_license_file
         Attempts to acquire various features from binary license file,
         trusted storage, and trial license sources.
```

Running the FlexNet Embedded Examples

The FlexNet Embedded examples (the code for which is found in the client_samples directory) illustrate how to acquire license rights from various sources and therefore run with their own specific command-line arguments. To demonstrate running a basic FlexNet Embedded example, this section will execute one of the simplest examples, the **Client** example.

This section covers the following:

- Generating Example License Rights
- Running the Example "Client" Project

Generating Example License Rights

The various examples in the `client_samples` directory illustrate how to acquire license rights from various sources. One of the simplest examples, the **Client** example, attempts to acquire license rights from a binary license file, trusted storage, or trial storage, as available. To create a binary license file that can be used by the **Client** example, you can use the `licensefileutil` utility to convert a text license into binary format.

The **Client** example attempts to acquire several features. To create an unsigned license file that can be acquired by the **Client** example, create a text file called `demo.lic` with the following contents (or copy the provided example file `examples\licenses\demo.lic`):

```
INCREMENT survey demo 1.0 31-dec-2020 uncounted HOSTID=ID_STRING=1234567890
INCREMENT highres demo 2.0 permanent uncounted HOSTID=ID_STRING=1234567890
INCREMENT download demo 2.0 permanent uncounted HOSTID=ID_STRING=1234567890
INCREMENT upload demo 2.0 permanent uncounted HOSTID=ANY START=1-jun-2009
INCREMENT special demo 1.0 permanent uncounted HOSTID=ID_STRING=1234567890 START=1-jun-2009
INCREMENT updates demo 1.0 permanent uncounted HOSTID=ID_STRING=1234567890 START=1-jun-2009
INCREMENT sdchannel demo 1.0 permanent 100 HOSTID=ID_STRING=1234567890 START=1-jun-2009 \
   VENDOR_STRING="Standard Definition Channels" 20092016 \
   VENDOR_STRING="High Definition Channels"
```

(This example license uses some optional fields such as `VENDOR_STRING` not specifically used by the **Client** example. See Feature Definitions for an explanation of the text license format.)

To convert this file into binary format with digitally signed licenses, copy the file into the *`install_dir`*`/tools/bin` directory and run the command:

```
licensefileutil -id IdentityBackOffice.bin demo.lic demo.bin
```

In the following section, you will build the **Client** example and acquire licenses from the binary file `demo.bin`. For other samples, you can use other utilities for license conversion, such as `trialfileutil` for the **Trials** example. For more information, see Utility Reference.

Running the Example "Client" Project

As demonstration on how to run the FlexNet Embedded examples, run the **Client** example, using these steps:

1. Copy the binary license file `demo.bin` that you created in the previous step into the same directory as the `client` executable.
2. Ensure that the appropriate FlexNet Embedded client libraries for FlexNet Embedded have been properly installed. For information about installing these libraries, see Install the FlexNet Embedded Client Libraries for FlexNet Embedded.

3. Launch the `Client` executable (from the `examples\client_samples\Client\Debug` directory), specifying the location of `demo.bin` as a command-line argument:

   ```
   Client demo.bin
   ```

   When you run this command, the output similar to the following is displayed:

   ```
   INFO: Reading data from demo.bin
   INFO: Number of features loaded from demo.bin: 8
   INFO: Number of features loaded from TrustedStorage: 0
   INFO: Number of features loaded from Trials: 0
   INFO: Acquired: name=survey, version=1.0, count=1, expiration=12/31/2020 11:59:59 PM
   INFO: Acquired: name=highres, version=2.0, count=1, expiration=permanent
   ERROR: FlxDotNetClient.FeatureNotFoundException encountered:
   Acquiring lowres license : Requested feature was not found.: [1,5,5,0[7000000B,0,7022F]]
   INFO: Acquired: name=download, version=2.0, count=1, expiration=permanent
       [...similar messages...]
   INFO: Acquired: name=sdchannel, version=1.0, count=100, expiration=permanent
   INFO: Acquired: name=hdchannel, version=1.0, count=10, expiration=permanent
   ```

The messages indicate whether a license for a given feature in the binary license file could be acquired. (In this case, an expected exception is displayed because the license rights did not include a “lowres” feature license.)

The Using the FlexNet Embedded APIs chapter shows you how to experiment with different representations of license rights and processes using the other examples and additional toolkit utilities.

Running the Updates and Insights Example

To run the Updates and Insights examples (the code for which is found in `uai_client_samples`), use these steps:

1. Ensure that a sample product package is defined and its update published in FlexNet Operations. For more information, see Add a Product and Its Update to the Publisher Site in the *Quick Start* chapter for more information.

2. Ensure that the appropriate FlexNet Embedded client libraries for Updates and Insights have been properly installed for the example. For information, see Install the FlexNet Embedded Client Libraries for Updates and Insights.

3. Launch the example executable from *`install_dir`*`\examples\uai_client_samples\Notification\Debug`, specifying the required command-line arguments. Issue a command similar to this one:

   ```
   notification -server https://siteID-ns-uat.flexnetoperations.com -register ACT01-PhotoPrint
       -download -packageid PhotoPrintInstaller -productlang 1033
   ```

   Note that the URL above points to a User Acceptance Test (UAT) environment indicated by the `-uat` following the *siteID*. For production environments, the `-uat` is omitted.

For a description of the arguments used to run the **Notification** example and sample output, see Running “Notification” in the *Quick Start* chapter.

The Using the Updates and Insights APIs chapter walks you through the **Notification** example code that retrieves notifications and, for update notifications, downloads and installs the update.

Building and Running Examples in the .NET Core XT Toolkit

The following sections describe the basics for building and running the FlexNet Embedded examples included in the FlexNet Embedded Client .NET XT toolkit:

- Obtaining Producer Identity Data
- Basic Process for Building and Running an Example in the .NET Core XT Toolkit

Obtaining Producer Identity Data

Each producer is identified by a unique producer name and producer keys. To enable your FlexNet Embedded Client .NET Core XT toolkit, you must generate *producer identity data* to be used by your back-end tools and by your client code. For more information about generating your identity information and distributing appropriately, see Creating the Producer Identity in the Quick Start with the .NET Core XT Toolkit chapter.

You can also find instructions for creating producer identity data in the Publisher Identity Utility section of the Utility Reference chapter.

***Note** • The "demo" vendor keys expire after the evaluation period has elapsed. Contact Revenera sales if your evaluation keys have expired.*

Ensure that the client-identity information is installed in a location accessible by your application code during compilation. (For building the toolkit examples, copy the `IdentityClient.cs` file to the *`install_dir`*`\examples\identity` directory.) Then use the following procedures to build the example projects.

Basic Process for Building and Running an Example in the .NET Core XT Toolkit

The following describes the process for building and running a FlexNet Embedded example in the FlexNet Embedded Client .NET Core XT toolkit. You must repeat this process for each example.

- Phase 1: Provide the Prerequisites
- Phase 2: Prepare to Build the Executable for an Example
- Phase 3: Build and Run the Example
- Displaying Usage Help for an Example

Phase 1: Provide the Prerequisites

Before building and running a FlexNet Embedded example, ensure that the prerequisites are in place:

- Client Identity File: Copied to Proper Location
- License Rights: Created and Copied to Proper Location

- .Net Core: Installed on the Target Machine

Client Identity File: Copied to Proper Location

Verify that the `IdentityClient.cs` file (built using steps in Creating the Producer Identity) resides in the *`install_dir`*`\examples\identity` directory.

License Rights: Created and Copied to Proper Location

One way to store license rights on a client is in a digitally signed binary file, which you can create based on an unsigned text representation of the license rights. Both the **BasicClient** and the **Client** examples use a binary license file. The following procedure describes how to create the appropriate binary license file for each of these examples.

Task ***To create the binary license file for examples***

1. Create the appropriate text file:

   For the **BasicClient** example, create a text file called `license.txt` with the following contents:

   ```
   INCREMENT survey demo 1.0 1-jan-2025 uncounted HOSTID=ID_STRING=1234567890
   INCREMENT highres demo 1.0 1-jan-2025 uncounted HOSTID=ID_STRING=1234567890
   ```

   For the **Client** example, use a copy of `demo.lic`, found in the *`install_dir`*`\examples\licenses` directory. The file contains the following license rights:

   ```
   INCREMENT survey demo 1.0 31-dec-2020 uncounted HOSTID=ID_STRING=1234567890
   INCREMENT highres demo 2.0 permanent uncounted HOSTID=ID_STRING=1234567890
   INCREMENT download demo 2.0 permanent uncounted HOSTID=ID_STRING=1234567890
   INCREMENT upload demo 2.0 permanent uncounted HOSTID=ANY START=1-jun-2009
   INCREMENT special demo 1.0 permanent uncounted HOSTID=ID_STRING=1234567890 START=1-jun-2009
   INCREMENT updates demo 1.0 permanent uncounted HOSTID=ID_STRING=1234567890 START=1-jun-2009
   INCREMENT sdchannel demo 1.0 permanent 100 HOSTID=ID_STRING=1234567890 START=1-jun-2009 \
      VENDOR_STRING="Standard Definition Channels" 20092016 \
      VENDOR_STRING="High Definition Channels"
   ```

2. Use the `licensefileutil` utility to digitally sign the license rights so that only your license-enabled code can acquire the licenses. From the *`install_dir`*`\bin\tools` directory, run the following command in a console window (shown here for `demo.lic`):

   ```
   licensefileutil -id IdentityBackOffice.bin demo.lic demo.bin
   ```

   The output (for example, `demo.bin` or `license.bin`) is a binary representation of your license rights that can be acquired by license-enabled code at run time.

3. So that the binary license file resides in a place where the .NET code for the example can read it, copy the file to the *`install_dir`*`\examples\client_examples\`*`exampleName`* directory, which contains project file for the given example (identified by *exampleName*).

.Net Core: Installed on the Target Machine

Microsoft .NET Core needs to be installed on the machine on which you are building and executing a FlexNet Embedded examples. Refer to the current *FlexNet Embedded Client Release Notes* for the list of .NET Core framework versions that FlexNet Embedded supports.

If you need to download and install .NET Core, you can use the following link:

https://www.microsoft.com/net/core#windowsvs2017

Phase 2: Prepare to Build the Executable for an Example

Use the following steps to prepare to build the executable for a given example:

- Step 1: Ensure Project File Points to Correct .NET Core Framework Version
- Step 2: Run “restore” to Obtain Latest Packages
- Step 3: Copy “FlxCore” to the Project Folder

Step 1: Ensure Project File Points to Correct .NET Core Framework Version

By default, the project file for a given example points to a specific .NET Core framework target to build and run the FlexNet Embedded example. You need to review this file and make any necessary edits to ensure that it specifies the .NET Core framework version installed on your machine, as described in .Net Core: Installed on the Target Machine.

Task ***To ensure that the project file points to the correct .NET Core version***

1. Navigate to the *`install_dir`*`\examples\client_samples\`*`exampleName`* directory, and open the example’s project (`.csproj`) file in a text editor.
2. If necessary, edit the contents to ensure that the project points to the correct .NET Core framework version installed on your machine.

   You have the option to point to multiple target frameworks. If you do so, all specified targets are compiled when you build the project. However, you will need to designate the specific target under which to execute the assembly (described later in Phase 3: Build and Run the Example).

Step 2: Run “restore” to Obtain Latest Packages

A “dotnet restore” process is necessary to ensure that your machine has the latest packages required to build and run the FlexNet Embedded example. In addition to installing required .NET Core packages, this process also installs the required FlexNet Embedded component `flexera.flxlicensingclient.core` if it is missing.

Task ***To run the “dotnet restore”***

At a command prompt, navigate to the *`install_dir`*`\examples\client_samples\`*`exampleName`* directory (that is, the location of the example’s project file), and run the following:

```
dotnet restore
```

This process installs the necessary packages on your machine. Typically, this is in the current user's `.nuget` directory.

Step 3: Copy "FlxCore" to the Project Folder

FlexNet Embedded makes use of `FlxCore` native component for some of its licensing functionality. You must extract the appropriate `FlxCore` component in your toolkit and copy it to a location accessible by the FlexNet Embedded example. The **FlxCore** native components are stored under the `FlxCore` folder within the `Flexera.FlxLicensingClient.core.`*`version`*`.nupkg` archive, which is located in the *`install_dir`*`\lib` directory.

Task ***To locate and copy the native FlxCore component***

1. Within the `Flexera.FlxLicensingClient.core.`*`version`*`.nupkg` archive, navigate to the `FlxCore` directory to locate the component appropriate for the operating system on which your building and executing the example. For the list of available `FlxCore` native components, see .NET Core XT Toolkit Files to Distribute.

   Important • *Always use the version of the "FlxCore" library that matches the version of FlexNet Embedded used in your implementation.*

2. Extract the `FlxCore` component and copy it to the location of the project file for the example (*`install_dir`*`\examples\client_samples\`*`exampleName`* directory.

Phase 3: Build and Run the Example

The following procedure describes how to build and run the FlexNet Embedded example.

Note • *While the procedure described here uses a command line, you can also build the example using Microsoft Visual Studio. See the current FlexNet Embedded Client Release Notes for the Visual Studio versions that support this process.*

Task ***To build and run the example***

At a command prompt, do one of the following, based on your .NET Core target specification in the project (see Step 1: Ensure Project File Points to Correct .NET Core Framework Version):

- If you pointed to a single .NET Core framework target in the project, run the following command from the *`install_dir`*`\examples\client_samples\`*`exampleName`* directory to compile and run the example run using that framework version:

  ```
  dotnet run
  ```

- If you specified multiple .NET Core framework targets, use the `-f` argument to specify the framework version under which to execute the example. (Note that all framework targets that you specify in the project are compiled.) For example, to run the example under the `netcoreapp2.0` target (that is, .NET Core 2.0), you would enter the following:

  ```
  dotnet run -f netcoreapp2.0
  ```

When the example runs, it provides output to inform you if the example succeeds.

Displaying Usage Help for an Example

To display help information for any example, launch the executable with the `-h` or `-help` switch, as shown here for the client executable:

```
Client.exe -help
```

A usage message similar the following displays:

```
USAGE:


Client binary_license_file
         Attempts to acquire various features from binary license file,
         trusted storage, and trial license sources.
```

Toolkit Files to Distribute with Your Product

The following sections list the files you must distribute with your product enabled with FlexNet Embedded or Updates and Insights functionality (or both types of functionality).

- .NET XT Toolkit Files to Distribute
- .NET Core XT Toolkit Files to Distribute

.NET XT Toolkit Files to Distribute

When you distribute your product that has been leveraged with FlexNet Embedded licensing or Updates and Insights functionality or both, you need to distribute certain FlexNet Embedded Client .NET XT toolkit files along with your product. The table shown at the end of this section lists these files.

The toolkit files listed for **Common** in the table should always be shipped with your product, while the files listed for FlexNet Embedded and Updates and Insights in the table are shipped only if you have included that functionality in your product:

- If your product is enabled for FlexNet Embedded functionality, distribute the files listed for **Common** and FlexNet Embedded.
- If your product is enabled for FlexNet Updates and Insights functionality, distribute the files listed for **Common** and Updates and Insights.
- If you product is enabled for both FlexNet Embedded and Updates and Insights functionality, distribute the files listed for **Common**, FlexNet Embedded, and Updates and Insights.

All files listed in this table are found in the `lib` folder of your toolkit installation.

Important • *You have redistribution rights to the files listed in this table.*

Table 4-4 • Toolkit Deliverables to Customers

Client Functionality	Files
Common	• `FlxClientCommon.dll` • `FlxCore.dll` (or `FlxCore64.dll`)
FlexNet Embedded	`FlxLicensingClient.dll`
Updates and Insights	`FlxUAIClient.dll`

.NET Core XT Toolkit Files to Distribute

When you distribute your product that has been leveraged with licensing functionality from the FlexNet Embedded Client .NET Core XT toolkit, you need to distribute the following toolkit files along with your product:

- `Flexera.FlxLicensingClient.core.`*`version`*`.nupkg`
- The appropriate `FlxCore` native component, extracted from `Flexera.FlxLicensingClient.core.2017.11.0.nupkg` and placed in a location where the product can access it. You can find this component in the appropriate "operating system" folder for the client, located in the `FlxCore` directory within the .nupkg archive:
  - `libFlxCore.so.`*`version`* (for i86 Linux)
  - `libFlxCore64.so.`*`version`* (for x64 Linux)
  - `FlxCore.dll` (for i86 Windows)
  - `FlxCore64.dll` (for x64 Windows)
  - `libFlxCore.`*`version`*`.dylib` (for OS X 10)

Updates and Insights functionality is not supported in the .NET Core XT toolkit.

5

Overview of the .NET XT APIs

This chapter provides an overview of the .NET functionality included in the FlexNet Embedded Client .NET XT toolkit. The functionality is organized into the following interface groups. (Note that the FlexNet Embedded Client .NET Core XT toolkit supports only FlexNet Embedded functionality.)

- FlexNet Embedded API Interfaces—Used to perform licensing-related operations.
- Updates and Insights API Interfaces—Used to receive and act on product notifications.
- FlexNet Common API Interfaces—Used by both FlexNet Embedded and Updates and Insights functionality to handle errors and communications with servers.

See also the Conventions for Retrieving Exception Information for information on exception handling.

For more information about individual APIs in these groups, consult the API reference.

Note about References to Servers

In this chapter, note the following about references to servers:

- For FlexNet Embedded and Updates and Insights, “back-office server” refers to Revenera-hosted FlexNet Operations.
- References to “license server” apply to FlexNet Embedded only and refer to either the FlexNet Embedded local license server or a Cloud Licensing Service (CLS) instance.

FlexNet Embedded API Interfaces

The following are the primary interfaces defined by the FlexNet Embedded API:

- **ILicensing:** The `ILicensing` interface is used to get the various objects that handle licensing operations.
- **ILicenseManager:** The `ILicenseManager` interface provides methods for handling the most significant licensing operations: acquiring and releasing licenses, adding license sources, creating capability requests and processing capability responses, enabling virtual-machine detection, and more.
- **IPrivateDataSource:** This interface manages small amounts of producer-defined private data, which can be used by license-enabled code to assist with custom licensing scenarios.

- **IAdministration:** This interface enables code to perform administrative operations, such as deleting various types of storage.

Furthermore, the following interfaces are used for license manipulation and identification:

- **ILicense:** Once a license is acquired, the corresponding ILicense object can be queried for license details, such as its expiration date and optional license keywords (`VENDOR_STRING`, `NOTICE`, etc.).
- **IFeature:** This interface provides a way to examine the possible capabilities that can be acquired, by inspecting feature details before acquisition.
- **ICapabilityRequestOptions:** This interface enables license-enabled code to generate a binary capability request that can be communicated to a back-office server or a license server; the server then processes the request and generates a capability response. Depending on the type of server that will receive the request, the request includes some combination of data such as one or more rights IDs (for a back-office server) or sets of requested features (for a license server), and custom key–value dictionary pairs. The request can also be configured to request a preview of available features on the license server.
- **ICapabilityResponse:** This interface provides functionality for reading details of a capability response generated by a back-office server (such as FlexNet Operations or similar utility such as `capresponseutil`) or a license server. The server can send one or more response status (`IResponseStatus`) objects inside the response data to return additional information.
- **IInformationMessageOptions:** This interface represents options for messages that license-enabled code sends to a license server in certain failover or network-licensing scenarios to indicate license usage.

For more information about individual methods and interfaces in the FlexNet Embedded .NET XT API, consult the API reference.

Updates and Insights API Interfaces

The following are the primary Updates and Insights API interfaces:

- **IUAIClient:** The interface used to access the Updates and Insights client object (`UAIClient`), the top-level object used for all notification operations. This object maintains references to all product packages associated with it and manages registration of the client device with FlexNet Operations. The object is created through the `UAIClientFactory.GetUAIClient` method.
- **IUAIClientOptions:** An optional interface to store `IUAIClient` creation options.
- **IProductPackage:** The interface used to access the object for a given product package associated with the Updates and Insights client object. Through the product package object, all notifications for the given product package are retrieved from the notification server. (Only one notification collection per product package is allowed at any one time.)
- **INotification:** APIs that parse items in the notification collection for a given product package and determine notification item type (for example, an update or message).
- **INotificationUpdate:** APIs that download the payload for an update and install the update.

FlexNet Common API Interfaces

The following lists the groups of .NET XT APIs shared by both FlexNet Embedded and Updates and Insights functionality. The APIs in these groups are used for both error handling and communications with license and back-office servers.

Note • *For FlexNet Embedded, the reference to "back-office server" refers to FlexNet Operations. For Updates and Insights, "back-office server" refers to the notification server, a component of the Revenera-hosted FlexNet Operations. References to "license server" apply to FlexNet Embedded only.*

- **IComm:** APIs that enable basic HTTP, HTTPS, SSL, and proxy-server communications between the FlexNet Embedded client and the back-office server, notification server, or license server. Normally, the FlexNet Embedded client uses a default communications object. However, should communication settings need to be customized, these APIs provide the means to set up a new communication object with the desired settings.
- **IStatusInformation:** An interface that provides access to status information such status code, type, and description.
- **IHostIDCollection** and **IHostInformation:** An interface provides access to hostid management on the client.

Additionally, the **Common Client Exceptions** module provides the error codes shared by both FlexNet Embedded and Updates and Insights functionality. These codes identify errors that might be generated when either type of functionality is executed on the client, including communications with the back-office server, license server, or notification server.

Conventions for Retrieving Exception Information

The API methods in both .NET XT toolkits can throw exceptions derived from the `FlxException` class. For example, the `Acquire` licensing method that attempts to acquire a feature can throw a `FeatureNotFoundException`, a `FeatureExpiredException`, or a `FeatureHostIdMismatchException`, among others. Consult the API reference for further information about specific methods and their possible exceptions.

To retrieve information about an exception, your code can access these conventional properties in `FlxException`:

- The `Message` property retrieves an error or warning string corresponding to a particular exception.
- The `Code` property retrieves the error code associated with an exception.
- The `SystemCode` property retrieves the operating-system code associated with an exception.

Additionally, your code can use the `ToString` method to return a string representation of the exception information.

The following implementation is an example of how to get a property (in this case, the `SystemCode` property) for an exception:

```
public int SystemCode { get; set; }


catch (FlxException exc)
{
    int systemCode = exc.SystemCode;
}
```

You can use the `ToString` method to obtain all information for an exception, as shown in this example implementation:

```
Console.Writeline (exc.ToString());
```

6

Using the FlexNet Embedded APIs

You create license-enabled code that runs on a client machine using the FlexNet Embedded APIs included in the FlexNet Embedded Client .NET XT or the FlexNet Embedded Client .NET Core XT toolkit. These APIs are a family of .NET interfaces and methods for processing license rights, acquiring licenses, querying license data, and processing communications with the back-office server or a license server.

This chapter describes the general flow of FlexNet Embedded methods used when implementing various client scenarios, by referring to the source code for the sample projects. The source code files discussed in this chapter are located in the specific example project directory under *install_dir*\examples\client_samples.

Where appropriate, the walkthroughs illustrate usage of the corresponding toolkit utilities. For more information about the utilities provided with these FlexNet Embedded Client toolkits, see the chapter Utility Reference.

The scenarios described here include the following:

- Buffer Licenses
- Licenses Obtained from the Back-Office Server
- Licenses Obtained from a License Server
- Limited-duration Trials
- Secure Re-hosting
- Capturing Feature Usage on the Client
- Examining License Rights in a License Source
- Advanced Topic: FlexNet Publisher Certificate Support
- Advanced Topic: Multiple-Source Regenerative Licensing

Common Steps to Prepare for Licensing

The following describes steps to prepare your application code for any licensing scenario:

- Creating Your Producer Identity Files
- Creating Core Licensing Objects
- Detecting a Cloned Environment
- Detecting Clock Windback
- Identifying the Device User
- Retrieving Feature Expiration and Grace Period Information
- Including Vendor Dictionary Data
- Advanced Topic: Secure Anchoring

Creating Your Producer Identity Files

The following implementation walkthroughs assume you have already created your producer back-office identity, client–server identity, and client-identity files—by default called `IdentityBackOffice.bin`, `IdentityClientServer.bin`, and `IdentityClient.bin`—using a back-office server such as FlexNet Operations or the Publisher Identity utility `pubidutil`, as previously described in Creating the Producer Identity.

In addition, the example projects assume you have compiler-readable (C# byte array) identity information available in the files `IdentityClient.cs`, in *`install_dir`*`\examples\identity`. You use the `printbin` utility with the `-cs` switch to create such header files from your binary client-identity file `IdentityClient.bin`.

Special Consideration

You can configure the client identity binary to include hostid filtering and caching parameters for use during hostid detection on the client device. For more information, see Identity Update Utility in the *Utility Reference* chapter.

Creating Core Licensing Objects

In your license-enabled code, the first thing to do is to create your core `ILicensing` and `ILicenseManager` objects. The `ILicensing` object must be initialized with your producer client identity created with `pubidutil` (for details, see Publisher Identity Utility), along with your desired trusted storage implementation and optional hostid override. A sample implementation is the following:

```
private static ILicensing licensing = null;

// Get user's personal Documents folder
string strPath =
    Environment.GetFolderPath(Environment.SpecialFolder.Personal) + Path.DirectorySeparatorChar;

// Initialize ILicensing interface with identity data using file-based trusted storage
// and hard-coded string hostid "1234567890"
using (licensing = LicensingFactory.GetLicensing(
                        IdentityClient.IdentityData,
                        strPath,
```

```
                        "1234567890"))
{
   // Given the ILicensing object returned by GetLicensing, use LicenseManager property to
   // get the ILicenseManager object that provides most licensing operations
   licensing.LicenseManager.operationName(...);
}
```

(In the FlexNet Embedded functionality examples, most licensing operations are wrapped in a `try-catch` block. These blocks have been omitted from many of the code excerpts presented in this chapter.)

You initialize the `ILicensing` object by calling `LicensingFactory.GetLicensing`. The first argument to `GetLicensing` is your client-identity information used for validating licenses and capability response envelopes. The data for the binary client identity—the `IdentityClient.IdentityData` expression passed as the first argument to `LicensingFactory.GetLicensing`—used in this code sample can be found in `IdentityClient.cs` and was generated with the settings you specified when running `pubidutil`. This client-identity data contains the public key information used to authenticate licenses or capability response envelopes digitally signed by the back-office server, the license server, `licensefileutil`, and so forth.

Important • *For security reasons, your producer client identity should be stored as a buffer in the license-enabled code, and not as an external file. The "printbin" toolkit utility can convert a binary producer identity file (on a development system) into a format that can be used in .NET XT code.*

Specifying the Trusted Storage Location

The second argument to `GetLicensing` is a location where trusted storage license rights should be stored on a target system. To store trusted storage in files, specify in this argument a string that resolves to a writeable directory on the target system. (Your installation program or instructions should adjust directory permissions, as appropriate.) Many examples use `Environment.GetFolderPath` to get the location of the current user's `Documents` (or `My Documents`) folder.

Passing `null` in this argument—as is done in the **BasicClient** example—causes FlexNet Embedded functionality to use an in-memory implementation. This implementation does not use files for trusted storage, but instead stores license rights in memory, which means the information will be lost when the code exits. This is useful in situations where licenses are transient to a degree where writing to disk is unnecessary, such as using a license server with frequent renewals; or where the application requests its licenses at startup and keeps them in memory. In-memory storage is inappropriate in such cases as limited-duration trials, where license information should persist between application launches.

The trusted storage directory to use depends on your desired license models. In a multi-user environment, it may be desirable to specify a common, per-machine directory, so that license rights are shared by all users. However, this may require that your product installation procedure modify the trusted storage directory's permissions. For per-user license models, a per-user trusted storage location is generally appropriate.

After calling `LicenseFactory.GetLicensing` to initialize your `ILicensing` object, use its `LicenseManager` property to get your `ILicenseManager` object. This interface provides the methods for performing most licensing operations, such as acquiring and releasing licenses, creating and processing FlexNet Embedded messages, and so forth.

Specifying the Hostid Type to Use

An optional third argument to `GetLicensing` is a hostid override value. Normally, your code will specify a type of hostid used by the license-enabled code for the sake of node locking and for other client-system identification related to licensing. For testing, however, you can specify a string to use as a hard-coded hostid value. The example code in the FlexNet Embedded Client .NET XT or .NET Core XT toolkit uses the hard-coded string hostid value "1234567890", which corresponds to `HOSTID=ID_STRING=1234567890` in license rights.

The following describes more information about specifying the hostid:

- Setting a Default Hostid
- Processing the Hostid

For information about the hostid value in the license syntax and hostid case-sensititivy, see Hostids.

Setting a Default Hostid

At run time, FlexNet Embedded functionality can use any available hostid for the sake of node locking and other system identification. When not specifying a string hostid override, use the `SetHostId` method of the `ILicenseManager` interface.

The method signature is the following:

```
void SetHostId(HostIdEnum type, String id);
```

where:

- *type* is the hostid type to use. Supported types are:
  - `HostIdEnum.FLX_HOSTID_TYPE_ETHERNET` for an Ethernet address
  - `HostIdEnum.FLX_HOSTID_TYPE_INTERNET` for an IPv4 address
  - `HostIdEnum.FLX_HOSTID_TYPE_INTERNET6` for an IPv6 address
  - `HostIdEnum.FLX_HOSTID_TYPE_FLEXID9` for an Aladdin dongle
  - `HostIdEnum.FLX_HOSTID_TYPE_FLEXID10` for a Wibu-Systems dongle
  - `HostIdEnum.FLX_HOSTID_TYPE_VM_UUID` for a supported virtual machine's UUID value
  - `HostIdEnum.FLX_HOSTID_TYPE_CONTAINER_ID` for a container ID of a supported containerization technology
- *id* is the string representation of the hostid value.

Note • *Consult your back-office server documentation to see which hostid types it supports.*

To find the hostid types and values available on a particular client system at run time, use the `HostIds` property of the `ILicenseManager` interface. Its signature is:

```
Dictionary<HostIdEnum, List<String>> HostIds( )
```

In practice, client code will typically read the available hostid types and values using the `HostIds` property, and then call `SetHostId` with the desired type and value. (The `SetHostId` method throws an exception if the specified hostid is not present on the system. Modifying the `HostIds` dictionary has no effect on the set of available hostid values.)

For example, the following code will specify to use the first Ethernet address returned:

```
// get all available hostids
```

```
Dictionary<HostIdEnum, List<String>> hostIDs = licensing.LicenseManager.HostIds;

if (hostIDs.ContainsKey(HostIdEnum.FLX_HOSTID_TYPE_ETHERNET))
{
    // select only the Ethernet addresses
    List<String> ethernetIDs = hostIDs[HostIdEnum.FLX_HOSTID_TYPE_ETHERNET];
    // use the first Ethernet address (index 0) in the list
    licensing.LicenseManager.SetHostId(HostIdEnum.FLX_HOSTID_TYPE_ETHERNET, ethernetIDs[0]);
}
```

To use this code in the examples, modify the `GetLicensing` call not to use the string hostid override by removing the final "1234567890" argument, and place this code in the `using` block under `GetLicensing`. If your code does not set a hostid or use the string hostid override, by default the first Ethernet address is used.

Note that you can limit the hostid types retrieved on your system by injecting hostid-type filters in the client identity binary. See Identity Update Utility in the *Utility Reference* chapter for details.

Processing the Hostid

Setting the hostid type and value changes the hostid sent in capability requests and information messages, but does not affect whether licenses can be acquired. For example, consider a particular host that has Ethernet addresses E1, E2, and E3, on which the code has set the hostid value to E2. Capability requests originating from this host will use hostid E2, but the host can acquire (for example) features from a buffer license generated for E1, E2, or E3, or any other valid hostid of the system. Also note that, if a system has already created trusted storage by successfully processing a capability response, FlexNet Embedded will use the existing hostid from trusted storage in future capability requests.

Final "Get Licensing" Argument

The final argument to `GetLicensing` is an optional string name for the licensing object, used in situations where the `ILicensing` object using in-memory trusted storage goes out of scope and `GetLicensing` is called multiple times. For more information, consult the API reference.

Detecting a Containerized Environment

Containerization enables applications to run in an isolated environment. The API `IsContainerized` (part of the `iLicenseManager` interface) detects whether the client application is running in a container (returning a boolean value). Once this has been determined, your code can take appropriate action, such as to deny its operation.

If the application is running in a container, the client code can call the `HostIds` property to return and read the available hostid types. In containerized environments, calling `HostIds` should return a hostid CONTAINER_ID, which can be used for node-locking.

Note that the CONTAINER_ID hostid is not universally unique. However, due to it being short-lived (the CONTAINER_ID is only available while the container is running), it can be considered to be sufficient for concurrent and metered (usage–based) license models.

The **view** example included with the toolkit illustrates the use of container detection.

Output from **view** example:

- `Client is contained in a docker host`
- `Client is not contained in a docker host`

Detecting a Cloned Environment

A FlexNet Embedded client can obtain its licenses through capability exchanges with the back-office server (FlexNet Operations) or with a license server (FlexNet Embedded local license server or CLS license server), as described in Licenses Obtained from the Back-Office Server and Licenses Obtained from a License Server. When these types of exchanges are used, the server has a means to detect when a client might be running in a cloned environment and can provide this information to the back office, where you can then generate reports listing these potential clones. Note that this clone-detection feature simply reports plausible clones; it does not take any action on clone activity. Details about the feature are found in the "Advanced License Server Features" chapter in the *FlexNet Embedded License Server Producer Guide*.

In addition to being sent to the back office, the "clone suspect" status is also returned in the capability response to the client. As another means of gathering clone information, you can incorporate functionality in your code to get the `CloneSuspect` status directly from the response and use it for your own purposes.

The following shows an example implementation of this method:

```
ICapabilityResponse response = licensing.LicenseManager.GetResponseDetails(binCapResponse);
if (response.CloneSuspect)
{
    // provide message indicating clone suspect
}
```

Detecting Clock Windback

Clock-windback detection is a security feature that detects an attempt to set the client machine's system clock back in order to extend expiring license rights. The implementation of clock-windback detection involves comparing the current system time with a timestamp stored in anchor storage. A stored timestamp that is later than the current time—by a value that is greater than a given tolerance—is interpreted as an attempt to set the system clock back. If clock-windback detection is enabled, methods for acquiring a license or processing trial license rights will report a windback state if one is detected, and an implementer can explicitly test for the windback state at any time. Clock-windback detection requires use of trusted storage, but will work with buffer license sources (in addition to working with trial and trusted storage license sources).

By default, clock-windback detection is disabled, and an implementer enables it using the method `EnableClockWindbackDetection` of your `ILicenseManager` object. (You can disable clock-windback detection using `DisableClockWindbackDetection`.) In the method signature—

```
EnableClockWindbackDetection(uint tolerance, uint frequency);
```

—the arguments are for the windback tolerance and frequency.

The windback *tolerance* setting limits the number of seconds' difference allowed without triggering a clock-windback state. The windback *frequency* setting is a number of seconds between updates to the stored timestamp. Normally, the timestamp will be updated for any time-sensitive event (such as acquiring a license or processing trial rights or a new capability response), but setting a larger interval may be desirable when working with systems on which frequent writes to the trusted storage anchor are unacceptable or unnecessary. Setting the frequency to zero indicates that the anchor will be updated every time clock-windback detection occurs, whether explicitly or implicitly.

For example, suppose an expiring license is written to a system's trusted storage, at which time a timestamp is stored on the client system. Whenever the license is used, the system clock is compared to the stored timestamp; if the system time is equal to or later than the stored time, the time is considered valid and the stored timestamp is updated. When the license has expired, the stored timestamp is updated to a time beyond the license's expiration date. If the clock is subsequently wound back to a point before the license expiration, the current system time is found to be earlier than the stored

timestamp, and the system is found to be in a wound-back state. In such a case, the license-enabled code might indicate that the clock should be set to the correct time, after which the client system is no longer found to be in the wound-back state.

To explicitly detect if the client is in a clock-windback state, use the `ClockWindbackDetected` property. Calling the method `EnableClockWindbackDetection` a second or later time updates the windback-detection parameters.

The **View** example included with the toolkit illustrates the use of clock-windback detection.

Identifying the Device User

The capability request can include a requestor ID value to associate a user with the FlexNet Embedded client device issuing the request. This information is then used by FlexNet Operations to provide user association with the device. Depending on the producer-specific policies configured in FlexNet Operations, a requestor ID can be a mandatory field in a capability request. In such a case, when a capability request does not include valid requestor ID information, the capability response from the back office can contain the error status `FLX_MS_CODE_REQUESTOR_ID_INVALID`.

Set the `RequestorId` property in the `ICapabilityRequestOptions` interface to identify the device user in the capability request. Refer to the FlexNet Operations documentation for information about configuring support for, maintaining, and enforcing this user information in the back office.

Retrieving Feature Expiration and Grace Period Information

Product features obtained from the back-office server or a license server can have an expiration date defined in the entitlement in the back office. When features reach their entitlement expiration date, they can no longer be acquired by the product code, causing possible disruption in the use of the product until the customer renews the features in the back office. However, the entitlement can also define a grace period that goes into effect when the entitlement expiration date is reached, enabling your customers to continue operating under their normal business workflow, but also allowing them sufficient time to renew product licenses.

FlexNet Embedded provides .NET methods that retrieve feature expiration information from the capability response. You can use these methods to implement logic that informs customers to renew soon-to-expire features.

Types of Expiration Information Available for Retrieval

This section describes the three types of expiration dates that can be retrieved through FlexNet Embedded .NET methods. A simplistic way to understand these three dates is as follows:

(feature) expiration date <= entitlement expiration date <= final expiration date

The following explains the expiration dates in more detail:

- **Expiration date**—The date at which a feature is no longer available for acquisition on the FlexNet Embedded client. (That is, the client checks this date to determine whether a specific feature in a license source can be used to satisfy one of the license acquisition requests on the client.)
  - If the feature is served by a license server, the expiration date is calculated by taking the borrow interval into account. For more details, see How the Borrow Interval Is Determined.
  - If the feature is obtained directly from back-office server, the expiration date is the same as the final expiration date (see the next bullet).

- **Final expiration date**—The final date, as defined in the back office, when a feature is no longer available for serving by the license server or for acquisition from the back-office server (and consequently no longer available to satisfy license acquisition requests on the client). This date reflects the entitlement expiration date *plus the defined grace period*. If no grace period is defined, the final expiration date is the same as the entitlement expiration date (see the next bullet).
- **Entitlement expiration date**—The original expiration date in the entitlement; no grace period is included in this date. Subsequently, if a grace period is defined in the back office, the entitlement expiration date is earlier than the final expiration date. If no grace period is defined, the entitlement expiration and final expiration dates are the same.

.NET Properties Used to Retrieve Expiration Information

The following .NET properties in FlexNet Embedded retrieve the expiration information:

- **Expiration (in IExtendedLicensingAttributes interface)**—For features obtained directly from the back-office server, retrieves the final expiration date; for features served by a license server, retrieves the calculated borrow expiration date (or the final expiration date if the borrow expiration is later than or equal to the final expiration).
- **FinalExpiration (in IFeature interface)**—Obtains the final expiration date for a feature.
- **EntitlementExpiration (in IFeature interface)**—Obtains the entitlement expiration date for a feature.
- **IsInGracePeriod (in IFeature interface)**—Determines whether a feature is currently in a grace period (that is, the current date is *after* the entitlement expiration date but *before* the final expiration date).

Including Vendor Dictionary Data

The *vendor dictionary* provides an interface for an implementer to send custom data in a capability request (in addition to the FlexNet Embedded–specific data) to the back office or license server and vice-versa. Basically, the vendor dictionary provides a means to send information back and forth between the client and server for any producer-defined purposes, as needed; FlexNet Embedded does not interpret this data.

Vendor dictionary data is stored as key–value pairs. The key name is always a string, while a value can be a string or a 32-bit integer value. Keys are unique in a dictionary and hence allow direct access to the value associated with them. (Note that the maximum size for a vendor dictionary sent to the license server is 7168 bytes in base64.)

In a client implementation, call `AddVendorDictionaryItem` to add a single string or integer vendor dictionary item to a request. After the server's response has been received, inspect the `VendorDictionary` property to retrieve vendor dictionary items from the response.

For illustration, the **CapabilityRequest.cs** example sets two vendor-dictionary items—one a string, the other an integer:

```
// Optionally add capability request vendor dictionary items.
options.AddVendorDictionaryItem(dictionaryKey1, "Some string value");
options.AddVendorDictionaryItem(dictionaryKey2, 123);
```

Advanced Topic: Secure Anchoring

FlexNet Embedded offers advanced functionality called *secure anchoring* that provides a greater level of anchor security than the standard FlexNet Embedded anchoring techniques normally used for trusted storage on machines that run your license-enabled applications (see Trusted Storage). While default anchoring stores the anchoring information in the anchor file whose location you specify in `LicensingFactory.GetLicensing`, secure anchoring uses additional techniques to store anchor information on the target system.

Prerequisites

The operating systems supported by FlexNet Embedded generally provide all of the resources required to enable secure anchoring. However, if you choose to enable secure anchoring, you should be aware that its methods might result in issues, such as insufficient-rights issues, on some end-user systems. If a required resource is not available, FlexNet Embedded seamlessly defaults to a reduced set of anchoring techniques for that system.

Enabling Secure Anchoring

Enabling secure anchoring functionality requires you to perform an extra step after generating client-identity information. Specifically, after you have obtained the file containing the client-identity binary data (for example, `IdentityClient.bin`), as described in Creating the Producer Identity, and before you run the Print Binary Utility to put the identity data into a code-compatible format, you must run the Secure Profile Utility `secureprofileutil`. This utility uses a specified security profile to embed secure-anchoring configuration information into the identity data. (Security profiles, which are pre-defined by FlexNet Embedded, configure specific levels of anchor security. Currently, FlexNet Embedded offers only one security profile, called `xt-medium`.) A typical command is:

```
secureprofileutil -profile xt-medium IdentityClient.bin IdentityClientSecure.bin
```

You would then use `printbin` to create your C#-compatible identity file, and recompile your license-enabled application using this new identity data. (This process must be repeated if you ever update your identity data.)

When you enable secure anchoring, no code changes are necessary in your application, apart from updating the identity data as previously described. However, you must specify a trusted storage location in `LicensingFactory.GetLicensing`, as described in Specifying the Trusted Storage Location section. If no storage path is specified—thus indicating that in-memory trusted storage is to be used—secure anchoring will not be enabled.

When you enable secure anchoring, the additional techniques will require changes to the testing process. For example, when working with limited-duration trials, as described in Licenses Obtained from a License Server, to re-test a trial with a particular trial ID, it will be necessary to specify a load-always trial using the `-always` switch to `trialfileutil`, as an existing anchor typically prevents a trial from being re-processed. Before releasing the product, you would then specify a load-once trial with the default `-once` switch to `trialfileutil`.

Note that anchors created with secure-anchoring functionality are independent of anchors using standard anchoring functionality. This means, for example, that an existing product that uses standard anchors will not be affected by a newer product version that uses secure-anchoring functionality.

Buffer Licenses

The following scenario describes how the FlexNet Embedded functionality can be used when the producer wants to provide node-locked licenses on the client system.

Setting Up the License File

Any scenario involving manually creating buffer license rights involves the following steps:

- Step 1: Create an Unsigned License File
- Step 2: Generate a Signed Binary License File

Step 1: Create an Unsigned License File

Begin by manually creating a license file in any plain-text editor (such as vi or Windows Notepad), entering one or more feature definitions, as described in Feature Definitions, and saving the text file with the `.lic` file name extension, as in `license.lic`.

```
INCREMENT survey demo 1.25 1-jan-2025 uncounted HOSTID=ID_STRING=1234567890
INCREMENT highres demo 1.25 1-jan-2025 uncounted HOSTID=ID_STRING=1234567890
```

The names `survey` and `highres` are feature names. At run time, the license-enabled code will attempt to acquire features with these names. Feature names are case sensitive, whereas most hostid values specified in the `HOSTID` keyword are not case sensitive. For more information, see Hostids.

The name that follows the feature name is your producer name; “demo” is the name used by the evaluation toolkit. For a production toolkit, replace “demo” in the license rights with your producer name obtained from Revenera, which you used in `pubidutil` to generate your producer identity files.

Step 2: Generate a Signed Binary License File

The text-based license file must be converted to binary form so that it can be used on the client system in the license-enabled code. This conversion can be performed manually using the License Conversion Utility `licensefileutil`. In a full production environment, this functionality would likely be integrated into the regular back-office server processing.

To generate the signed binary license file, run the following command from the *`install_dir`*`/bin/tools` directory:

```
licensefileutil -id IdentityBackOffice.bin license.lic license.bin
```

The command assumes your producer back-office identity file `IdentityBackOffice.bin` (created with `pubidutil`) is in the same directory.

See the section on the License Conversion Utility for more details on how to run this utility.

Using the License on the Client

The following functionality should be added into the license-enabled code running on the client machine. The purpose of this logic is to enable the license-enabled code to identify the licensed features for the client system. Refer to the source code in the file `BasicClient.cs`, located in the `examples\client_samples\BasicClient` directory.

Running the **BasicClient** example with the `-h` or `-help` switch displays usage information.

Assuming you have already created the core objects as described in Creating Core Licensing Objects, perform the following steps:

- Step 1: Create and Populate the License Sources
- Step 2: Acquire the License(s)

- Step 3: Read the License Details

Step 1: Create and Populate the License Sources

For this sample code, the license data is read directly from a specified input file into an input buffer (using the helper method `Util.ReadData`), which is then loaded as a buffer license source.

```
Util.DisplayInfoMessage(string.Format("Reading data from {0}", inputFile));
byte[] buffer = Util.ReadData(inputFile);

if (buffer == null)
{
    // issue encountered accessing input file
    return;
}

// Add buffer license source
licensing.LicenseManager.AddBufferLicenseSource(buffer, "BasicClientLicenseSource");
```

The client source provided with the toolkit additionally adds trusted storage and trial license sources to the license source collection.

Tip • *See the API reference for the multiple signatures of methods that add licenses sources. For example, "AddBufferLicenseSource" can accept an array of bytes, a string file path, or a collection of items.*

Step 2: Acquire the License(s)

The license-enabled code can now attempt to acquire the license for a particular capability (in this case, a single license for a feature called "survey", version 1.0, as in the example signed license rights you created earlier) from the license source collection.

```
try
{
   // acquire the license
   ILicense acquiredLicense = licensing.LicenseManager.Acquire("survey", "1.0");

   try
   {
      string acqdFeature = acquiredLicense.Name;
      string acqdVersion = acquiredLicense.version;

      Util.DisplayInfoMessage(
         string.Format("License acquisition for feature '{0}' version '{1}' successful",
            acqdFeature, acqdVersion));

      //// application logic here
   }
   finally
   {
      // return license. note: it is also possible to use a "using" statement to
      // implicitly return the license
   acquiredLicense.ReturnLicense( );
```

```
}
```

The attempt to acquire a license looks through the sources in the license source collection in order, accepting the first valid license it encounters. Note that the feature version string specified in the acquisition attempt is the minimum acceptable version: a request for version 1.0 will succeed if version 1.0, 1.25, or 2.0 (for example) is available, and will fail if only version 0.5 is available. (Recall that feature names are case sensitive, and that feature versions are expected to be in *a.b* format.) A request for version 0.0 of a feature indicates that any version of the feature is acceptable.

The `acquire` method returns an `ILicense` object, populated with information about the license, if the license was successfully acquired. If acquisition failed, the exception returned contains information about the reason. For example, `FeatureHostIdMismatchException` indicates that the `HOSTID` value in the license does not match the current client machine's hostid, and `FeatureNotStartedException` indicates a license start date (`START` keyword value) that occurs in the future.

A license-acquisition attempt can request a specific count of licenses, which succeeds only if a sufficient count exists in the target's aggregate pool of licenses. The feature counts can be pooled from multiple license sources, if necessary. A feature definition that uses the count value "uncounted" will satisfy a request for any number of copies. A client can call `AvailableAcquisitionCount` to determine an available acquisition count for a feature in a given license source collection.

Tip • *To help diagnose difficulties acquiring license rights, FlexNet Embedded provides a diagnostic API that verifies whether buffer-based or trusted storage license rights are valid. For more information and an example, see Examining License Rights in a License Source. In addition, the FlexNet Embedded API can be used to query license rights without attempting to acquire the license rights. For example, "GetFeatureCollection" and related methods will list features in a license source without validating signatures or expiration dates; and "ValidStatusForAcquisition" will indicate whether a feature could be acquired without acquiring it. For more information, consult the API reference.*

Step 3: Read the License Details

It is expected that the license-enabled code will sometimes require access to additional information specified within optional license keywords such as `VENDOR_STRING`. This is performed by reading properties of the `ILicense` interface obtained with the `Acquire` method:

```
Console.Out.WriteLine(
   string.Format("License acquisition for feature '{0}' version '{1}' successful.",
      license.Name, license.Version));
```

Licenses Obtained from the Back-Office Server

This scenario involves license-enabled code generating a capability request and sending it to the back-office server. The server then processes the request and generates a capability response, which is then conveyed back to the client system. Once the response has been processed, the license rights in the response are available for acquisition on the client system. The **CapabilityRequest** example is used to demonstrate this process. The example is run against FlexNet Operations or, for a simple test, against `capserverutil`, the toolkit's test back-office server utility.

This section describes the following steps:

- Configuring the Back-office Server to Provide Access to Licenses
- Activation or Upgrade Steps

FlexNet Operations as "Back-Office Server"

This book assumes that FlexNet Operations is the back-office server (or simply "back office", as it is sometimes called in this book) and that all back-office functionality described is that of FlexNet Operations.

Configuring the Back-office Server to Provide Access to Licenses

To run the **CapabilityRequest** example, you need to configure FlexNet Operations with the proper entitlement information—enabling the example either to activate a rights ID or to obtain all available rights mapped to the client device. See your FlexNet Operations documentation for instructions to set up the entitlement. You might need to adjust the example code accordingly to reflect the appropriate client hostid and rights ID.

Alternatively, for a simple test back-office server implementation, you can use the Capability Server utility, included in the toolkit. This utility comes with some sample rights that work easily with the **CapabilityRequest** example to activate a specific rights ID or to obtain all available rights mapped to the client hostid ("1234567890" in the example). For details, see Capability Server Utility in the *Utility Reference* chapter.

The following steps assume that the back-office server—FlexNet Operations or the Capability Server utility—has been configured appropriately and is running.

Activation or Upgrade Steps

The following steps are to be performed in the license-enabled code to obtain a license with an updated set of capabilities (a new client configuration, for example). These steps typically would be initiated by the user or automatically as a scheduled task from code running on the client system. See the FlexNet Embedded source code in the file *install_dir*`\examples\client_samples\CapabilityRequest\CapabilityRequest.cs` for an example implementation.

Running the `CapabilityRequest` executable with the `-h` or `-help` switch displays usage information. (Specific commands are described later in this section.)

Perform the following steps:

- Step 1: Create the License Source
- Step 2: Create the Capability Request (with optional attributes described in Additional Capability-Request Options)
- Step 3: Send the Request to the Back-Office Server
- Step 4: Process the Capability Response

Step 1: Create the License Source

In license-enabled code that has created the core objects, this step is identical to the corresponding step in Buffer Licenses, Step 1: Create and Populate the License Sources, except that a trusted storage license source must be created.

```
// Add trusted storage license source
licensing.LicenseManager.AddTrustedStorageLicenseSource( );
```

Note that some operations, such as processing a capability response, automatically create a trusted storage license source.

If trusted storage has been previously used to store a capability response, then the trusted storage license source will contain license rights from this response, and these rights are immediately available for acquisition. If trusted storage has not been previously used, the trusted storage license source will have no license rights. In both cases, to get the current set of client-system capabilities from the back-office server, the license-enabled code must generate a capability request and process the capability response as described below.

Step 2: Create the Capability Request

The next step is to generate the capability requests, providing the producer identity, requested license rights, and other attributes.

```
ICapabilityRequestOptions options = licensing.LicenseManager.CreateCapabilityRequestOptions( );

// Set capability request options here, such as rights IDs or desired features

// Generate the request
ICapabilityRequestData capabilityRequestData =
   licensing.LicenseManager.CreateCapabilityRequest(options);
```

FlexNet Operations uses an activation ID to identify the license rights that the client is requesting. However, in the capability request, this activation ID is sent as a *rights ID*. The `AddRightsId` method adds a rights ID and a count value (for the number of rights ID copies) to the request. You can call `AddRightsId` multiple times to add multiple rights IDs.

For a description of other options you can include in the capability request, see the next section, Additional Capability-Request Options.

Additional Capability-Request Options

The following options are available when requesting rights IDs from the back-office server:

- Attribute to Obtain All Available Copies for a Rights ID If Requested Count Cannot Be Satisfied
- Host Names (Aliases) and Types
- Option to Force a Capability Response

Attribute to Obtain All Available Copies for a Rights ID If Requested Count Cannot Be Satisfied

By default, the back-office server grants a given rights ID only if the number of copies requested for that ID is available in the back office. As an alternative to this default behavior, the FlexNet Embedded client can mark a rights ID in a capability request as “partial”, indicating that the back-office server should go ahead and send however many copies are available for that rights ID should the available copy count in the back office fall short of the requested count.

The following describes more about rights IDs marked with the “partial” attribute:

- Marking a Rights ID as “partial”
- How the Request is Processed
- Considerations

Marking a Rights ID as "partial"

To mark one or more rights IDs as "partial" in the capability request, provide this basic flow in the code:

1. Create an `IRightsIdOptions` object using the `licensing.LicenseManager.CreateRightsIdOptions` method, and set the `PartialFulfillment` property to `true`.
2. For each rights ID you want to mark as "partial", use the `AddRightsId` method overload, which takes a reference to the `IRightsIdOptions` object.

Example Implementation

The following shows a sample implementation that requests two rights IDs—one marked as "partial" (`li1`) and one (`li2`) not marked with this attribute:

```
ICapabilityRequestOptions crOptions =
      licensing.LicenseManager.CreateCapabilityRequestOptions();
IRightsIdOptions rightsOptions = licensing.LicenseManager.CreateRightsIdOptions();
rightsOptions.PartialFulfillment = true;
crOptions.AddRightsId(new RightsIdData("li1", 9, rightsOptions));
// Old of adding rights ids without options is not impacted.
crOptions.AddRightsId("li2", 2);
```

How the Request is Processed

When the back-office server processes a capability request that contains a rights ID marked as "partial", the server attempts to satisfy the number of copies requested for that ID. If the back office does not have a sufficient number of copies to satisfy the requested number, it sends whatever remaining copies are available for that rights ID in the capability response. The following examples demonstrate what happens when given rights IDs are marked and not marked as "partial".

Example 1

The FlexNet Embedded client sends a capability request for 5 copies of the rights ID `li1` and 15 copies of `li2`. The license server currently has 5 copies of `li1` but only 10 copies of `li2`. The following happens:

- If neither rights ID is marked as "partial", the back-office server sends the 5 copies of `li1` only. No copies of the rights ID `li2` are included in the capability response because the back-office server cannot satisfy all 15 copies requested.
- If both rights IDs are marked as "partial", the back-office server sends the 5 copies of `li1` and the available 10 copies of `li2` in the capability response.

Example 2

The FlexNet Embedded client sends a capability request for 5 copies of the rights ID `li1` and 15 copies of `li2`. The back-office server currently has only 4 copies of `li1` and 10 copies of `li2`. The following happens:

- If neither rights ID is marked as "partial", the back-office server sends no rights IDs in the capability response since it cannot satisfy the requested number of copies for either rights ID.
- If the rights ID `li1` is marked as "partial" but `li2` is not, the back-office server sends the remaining available 4 copies of `li1` in the capability response but includes no copies of `li2`.
- If both rights ID are marked as "partial", the back-office server sends the remaining 4 copies of `li1` and the remaining 10 copies of `li2` in the capability response.

Considerations

The availability of rights IDs in a given entitlement on the back-office server depends on the collective activities of all FlexNet Embedded clients in an organization that share that entitlement. Therefore, if a client resends a capability request for a rights IDs marked as "partial", the resulting capability response can include a copy count different from the number of copies returned for that rights ID when the request was sent previously.

Host Names (Aliases) and Types

In addition, a capability request can include a *host name*—that is, a human-readable "alias", in contrast to the hostid—and a *host type* if the back office requires these attributes to determine the client's license rights. Check with the FlexNet Operations administrator to determine whether these attributes are required.

To set these values, an implementation uses the `HostName` and `HostType` properties in the `ILicenseManager` object.

Option to Force a Capability Response

The typical behavior of the back office is to send a capability response only if the client's license rights have changed since the last capability exchange. If no changes have occurred, a 0-size response is returned to the client.

However, you can use the `ForceResponse` method in the `ICapabilityRequestOptions` object to set a "force response" flag in the capability request. When this flag is sent in the capability request, the back office will always return a capability response even if the client's license rights have not changed.

Use this flag with caution, mostly for exception cases such as these:

- To restore current license rights should client trusted storage be deleted due to corruption
- To process a capability response that previously failed
- To resolve synchronization timestamp issues

Step 3: Send the Request to the Back-Office Server

The **CapabilityRequest** example uses the `SendCapabilityRequest` helper method to send the request to the test back-office server. The request is sent over `HTTP POST`, and the response is obtained as part of the corresponding `HTTP POST` response.

Note • *FlexNet Embedded provides support for HTTP, proxy-server, and SSL communications with a back-office server. See the API reference for a description of the methods used for the different types of communications.*

For the **CapabilityRequest** example, issue this command to send the capability request to FlexNet Operations and to process the response. If necessary, adjust the default port (8888) to match your FlexNet Operations installation.

```
capabilityrequest -server http://hostname:8888/flexnet/deviceservices
```

If you are using the Capability Server utility (the test back-office server) to run the **CapabilityRequest** example, issue this command:

```
capabilityrequest -server http://localhost:8080/request
```

The example's `SendCapabilityRequest` method uses the `SendBinaryMessage` method from the `IComm` class to send the binary capability request and receive the response.

Note • *Any errors encountered by "IComm" methods will throw exceptions derived from "CommException". See the API reference for additional configuration options and error details. If desired, for troubleshooting, you can enable network tracing functionality for your .NET executable by providing an application configuration file. Consult the Microsoft .NET documentation for details.*

It is not a requirement to use the `IComm` methods to send messages to a back-office server or license server. An implementation can use any transport mechanism available on the client machine that is used to generate requests and receive responses. The producer supports a given implementation as part of a collection of "remote update/verification" transactions.

FlexNet Embedded functionality also supports the use of intermediate files for the capability request-and-response exchange instead of direct communications. The **CapabilityRequest** sample project supports offline transactions, and Capability Request Utility and Capability Response Utility utilities can also be used for such purposes.

More About the Capability Response

In whatever manner the capability request is conveyed to the back-office server, the server generates a capability response, which contains the new or updated license rights to be stored in trusted storage for acquisition. It is this capability response that is processed in the next step.

However, FlexNet Embedded does not require that the back office produce a capability response if license rights have not changed on the client system since the last capability request, unless the request has the "force response" flag set. For more information, about this "force response" flag, see Option to Force a Capability Response.

Step 4: Process the Capability Response

In order to use license rights from a capability response, the license-enabled code processes it into a trusted storage license source. When processing a capability response, the FlexNet Embedded libraries automatically validate the response's digital signature generated by the back-office server, the response hostid, and other information.

The `ProcessCapabilityResponse` helper method accepts the byte array containing the capability response, either received directly from a server or read in from a binary file, and creates the trusted storage license source.

```
ICapabilityResponse response =
       licensing.LicenseManager.AddTrustedStorageLicenseSource(binCapResponse);
```

Retrieving Response Contents

If the response contains any vendor-dictionary items sent by the server, the code can inspect the `VendorDictionary` property to get the dictionary.

A capability response can also contain one or more *response status items*. A response status item consists of a status code, status item detail, and a status category, and a server can return multiple status items to indicate different statuses if (for example) multiple rights IDs were included in a capability request sent to FlexNet Operations or multiple features were requested from a license server. To get the status items contained in the response, examine the `Status` property; followed by looping over the list of items, reading properties of `IResponseStatus` to get data from each individual status item.

Additionally, the capability response can include a flag indicating that the client needs to send a confirmation request back to the back-office server. (The back office might need such a request to verify that the client has successfully processed a license-return response. The request confirms that the client's license count is indeed reduced by the returned amount.) Client code can retrieve this flag using `ConfirmationRequestNeeded` in the `ICapabilityResponse` object. Note that the client has the option to send the confirmation request or ignore it. However, in some cases, not sending a confirmation request can have undesirable results. For example, if the back-office server is set up to require confirmation requests for returned licenses, the customer is not credited for the returned licenses until the back office receives the confirmation request.

Processing into Trusted Storage

When a capability response is processed into the trusted storage license source, it replaces any previously available license rights in this license source. This can make some of the licenses that were previously acquired from this source invalid. For this reason, it is strongly recommended to return licenses before processing a new response, and then to re-acquire the licenses after the new response is processed.

When a capability response is successfully processed, the new data is automatically saved to the trusted storage location and is available to the license-enabled code even after a client-system restart.

Once license rights are in trusted storage, license-enabled code can acquire the license rights and read the license details using the code described in Using the License on the Client, as long as the license collection includes a trusted storage license source.

Licenses Obtained from a License Server

You can use the **CapabilityRequest** example to demonstrate how the FlexNet Embedded client obtains licenses from a license server instead of directly from the back-office server. The client uses mostly the same APIs to create and send capability requests and process responses, whether it is obtaining licenses from the back-office server or a license server. The most obvious difference is that the client requests licenses from the license server by specifying "desired features", not a rights IDs as it does when requesting licenses from the back office.

A license server can be one of these:

- A FlexNet Embedded local license server
- A Cloud Licensing Service (CLS) instance, also called the *CLS license server*

The following sections describe the modifications needed to run the example against a license server. The sections also highlight options available specifically for capability requests sent to a license server and describe how the license server grants the requested licenses.

- Provision the License Server with Licenses for the Demonstration
- Register the Client with the Cloud Licensing Service
- Provide the URL for the License Server in the Command
- Modify the Example Code to Request "desired features"
- Additional Capability-Request Options
- License Checkout from the License Server
- Capability Preview

For all other instructions about preparing your licensing environment, setting up and sending a capability request in general, and processing the capability response, see Configuring the Back-office Server to Provide Access to Licenses and Activation or Upgrade Steps in *Licenses Obtained from the Back-Office Server* section.

Provision the License Server with Licenses for the Demonstration

You must use FlexNet Operations as the back-office server needed to provision the license server with the licenses needed for serving the FlexNet Embedded client in the **CapabilityRequest** example. See your FlexNet Operations documentation for instructions about setting up an entitlement that allows you to provision the license server.

Alternatively, for a simple test back-office server implementation, you can use the Capability Server utility, included in the toolkit, to provide licenses for the license server. While this utility allows you to create license rights, it comes with sample license rights that easily mesh with the **CapabilityRequest** example. For details about configuring, starting, and using this utility, see Capability Server Utility in the *Utility Reference* chapter.

The license server must be running and provisioned with licenses before executing the **CapabilityRequest** example. Refer to the *FlexNet Embedded License Server Administration Guide* for instructions about starting the license sever and requesting license activation.

Additionally, before running the **CapabilityRequest** example, you might need to modify the code to reflect the correct client hostid and desired features (as described in the next section).

Register the Client with the Cloud Licensing Service

The configuration of a CLS license server can require that the FlexNet Embedded client device register with the Cloud Licensing Service as an extra security measure before allowing the client to request features or report usage. However, by default, this requirement is disabled. If using a CLS license server, consult the FlexNet Operations administrator to determine whether registration is required. If it is, refer to Client Registration with the Cloud Licensing Service in the *Capturing Feature Usage on the Client* section for information about setting up a separate capability request that initiates this registration.

Provide the URL for the License Server in the Command

Specify the appropriate URL for the license server when running the **CapabilityRequest** example:

- For the FlexNet Embedded local license server, run the following, where *hostname* is the name of the machine running the license server:

  ```
  capabilityrequest -server http://hostname:7070/request
  ```

- For the CLS license server, run the following, where *siteID* is the producer's specific site ID supplied by Revenera and *instId* is the server's instance ID in the Cloud Licensing Service:

  ```
  capabilityrequest -server https://siteID-uat.compliance.flexnetoperations.com/instances/instId/
      request
  ```

  Note that the URL above points to a User Acceptance Test (UAT) environment indicated by the `-uat` part following the *siteID*. For production environments, the `-uat` is omitted.

Modify the Example Code to Request “desired features”

Modify the example code to request *desired features* from the license server (instead of the *rights ID* used to obtain licenses from the back office), and rebuild the code:

```
options.AddDesiredFeature(new FeatureData(“f1”, “1.0”, 1));
options.AddDesiredFeature(new FeatureData(“f2”, “1.0”, 1));
```

Additional Capability-Request Options

The following options are available when requesting desired features from a license server:

- Incremental Capability Requests
- Attribute to Check Out All Available Quantity for a Feature If Requested Count Cannot Be Satisfied
- Feature Selectors in a Capability Request
- Secondary Hostids
- Option to Force a Capability Response
- Borrow Interval and Granularity Overrides

Incremental Capability Requests

As described in Processing into Trusted Storage, each time the FlexNet Embedded client processes a capability response containing its “desired” features from the license server, the client’s currently existing licenses are removed and the new licenses added. If the client wants to maintain its existing licenses when it requests new features, two options are available. Either the client can explicitly include the existing licenses as desired features in the capability request along with any new desired features; or it can use the `Incremental` property in the `ICapabilityRequestOptions` interface to mark the request as “incremental” (and avoid having to specify the existing features).

The following topics describe how incremental capability requests work:

- Marking a Request as “incremental”
- How the Request is Processed
- Incremental Request Examples
- Considerations and Limitations

Marking a Request as “incremental”

The following shows a sample implementation for marking a capability request as “incremental”:

```
ICapabilityRequestOptions options =
        licensing.LicenseManager.CreateCapabilityRequestOptions();
options.Incremental = true;
```

How the Request is Processed

When the license server processes a capability request marked as “incremental”, it automatically attempts to renew all licenses currently served to the client and include the renewed licenses in the capability response along with any new features requested. If the license server determines that an existing feature on the client cannot be renewed (for example, it has expired or is no longer available), that feature is not included in the response. Ultimately then, the capability response includes all available non-expired existing features, along with all available desired features. If no desired features are specified in the incremental request, the server sends only the available non-expired existing features.

The capability request can also explicitly include an existing feature as a desired feature with a positive or negative count to add to or decrement the count renewed for that feature.

Incremental Request Examples

The next sections provide examples of the type of capability responses the FlexNet Embedded client can receive from incremental capability requests.

Example 1: Renew Existing Features and Add New Features

If a client that has been previously served only one license—1 count of the `survey` feature—sends a capability request for 1 desired count of the `highres` feature, the following happens:

- If the capability request is not marked as “incremental”, the license server generates a capability response for the 1 desired count of the `highres` feature, if it is available. (If `highres` is not available, the response is sent with no feature included, and the client ends up with no licenses.)
- If the request is marked as “incremental”, the response from the license server contains both 1 count of `survey` (if the feature is renewable) and 1 desired count of `highres` (if it is available). If the 1 count of `survey` is not renewable but the 1 desired count of `highres` is available, the response contains the 1 desired count of `highres` only. Conversely, if the 1 count of `survey` is renewable but the 1 desired count of `highres` is unavailable, the response contains 1 count of `survey` only.

Example 2: Renew Existing Features and Add Counts to Selected Renewed Features

A capability request marked as “incremental” allows the client to increment the count of an existing feature. For example, if a client that has been previously served 1 count of `survey` sends a capability request for 2 desired counts of `survey`, the following happens (assuming that the existing feature is renewable and the requested counts are available):

- If the capability request is not marked as “incremental”, the capability response includes simply the requested 2 counts of `survey`.
- If the request is marked as “incremental”, the response contains 3 counts of `survey`—1 count of the renewed `survey` feature and the requested 2 new counts of `survey`.

Example 3: Renew Existing Features But Reduce Counts for Selected Renewed Features

Additionally, a capability request marked as “incremental” allows the client to renew all existing features but reduce the count of (or not renew at all) selected existing features. The request must explicitly include each of these features as a desired feature with a negative count.

For example, a client has previously been served 10 counts of the `survey` feature, 4 counts of the `highres` feature, and 1 count of the `lowres` feature. If the client sends a capability request for -3 desired counts of `survey`, -1 desired count of `lowres`, and 2 desired counts of `medres`, the following happens (assuming that the existing features are renewable and the requested counts for the new feature are available):

- If the capability request is not marked as “incremental”, the license server responds with only what you ask for in the request. Hence, the response includes the 2 counts of new feature `medres` (and indicates that the negative counts for `survey` and `lowres` are invalid). Best practice is to avoid including negative counts for features when the capability request is not marked as “incremental”.
- If the request is marked as “incremental”, the response includes 7 renewed counts of `survey` (the original 10 counts decremented by 3), 4 renewed counts of `highres`, and 2 counts of the new feature `medres`. The original 1 count of `lowres` was negated by the requested -1 count for this feature, and thus `lowres` was not renewed.

Considerations and Limitations

Note the following about incremental capability requests:

- An incremental capability request is compatible with only the “request” operation type and concurrent features.
- Incremental capability requests are compatible with license reservations on the license server. For more information about license reservations, refer to the *FlexNet Embedded License Server Producer Guide*, specifically the “More About Basic License Server Functionality” chapter and the “Effects of Special Request Options on the Use of Reservations” appendix.
- If the borrow interval for an existing feature has expired, that feature must be explicitly included in the capability request as a desired feature.
- The license server processes desired features in the order in which they are listed in the capability request. This order can be important when an incremental capability request includes both negative and positive counts that decrement and add to the existing counts of selected features being renewed.
- When an incremental capability request includes a negative count that is greater than the current count for the specified version of an existing feature, the server first negates (that is, does not renew) all counts of the specified feature version. It then attempts to complete the decrement from the counts of a greater version for that feature.

  For example, a client might have 1 count of `f1 version 1.0` and 2 counts of `f1 version 2.0`. If the incremental request asks for -2 counts for `f1 version 1.0`, the server would negate the 1 count of `f1 version 1.0` first and then decrement 1 count of `f1 version 2`. The remaining 1 count of `f1 version 2.0` would be renewed.

  When no greater version of the specified feature exists, all counts of the specified feature version are negated, and a status message is generated to state that the requested negative count was greater than the actual count for the feature. For example, if the client has 1 count of `f1 version 1.0` and the incremental request asks for -2 counts for `f1 version 1.0`, the server would negate the 1 count of `f1 version 1.0` and provide the status message.
- When an incremental capability request includes a negative count for a concurrent feature for which the client currently has a 0 count, the license server sends the error message `The feature cannot be returned because it is not reusable`. (This specific error message is issued because the server perceives the feature as metered, not as incremental since there is no count to increment on the client.)

Attribute to Check Out All Available Quantity for a Feature If Requested Count Cannot Be Satisfied

By default, the license server grants a given “desired feature” only if the count requested for that feature is available on the server. As an alternative to this default behavior, the FlexNet Embedded client can mark a feature in a capability request as “partial”, indicating that the license server should go ahead and send whatever is available for that feature should the available count for the feature on the server fall short of the requested count.

The following describes more about desired features marked with the "partial" attribute, also called *partial-checkout* features:

- Marking a Feature as "Partial"
- How the Request is Processed
- Considerations and Limitations

Marking a Feature as "Partial"

To mark one or more desired features as partial in the capability request, provide this basic flow in the code:

1. Create an `IDesiredFeatureOptions` object using the `licensing.LicenseManager.CreateDesiredFeatureOptions` method, and set the `PartialFulfillment` property to `true`.
2. For each desired feature you want to mark as "partial", use the `AddDesiredFeature` method overload, which takes a reference to the `IDesiredFeatureOptions` object.

The following shows a sample implementation that requests three desired features—two marked as "partial" and one (`lowres`) not marked with this attribute:

```
IDesiredFeatureOptions featureOptions = licensing.LicenseManager.CreateDesiredFeatureOptions();
featureOptions.PartialFulfillment = true;
options.AddDesiredFeature(new FeatureData("survey", "1.0", 15, featureOptions));
options.AddDesiredFeature(new FeatureData("highres", "1.0", 5, featureOptions));
options.AddDesiredFeature(new FeatureData("lowres", "2.0", 5));
```

How the Request is Processed

When the license server processes a capability request that contains a feature marked as "partial", the server attempts to satisfy the count requested for that feature. If the server does not have a sufficient count to satisfy the requested count, it sends whatever remaining count is available for that feature in the capability response. The following examples demonstrate what happens when given features are marked or not marked as "partial".

Example 1

The FlexNet Embedded client sends a capability request for 5 counts of the `highres` feature and 15 counts of `survey`. The license server currently has 5 counts of `highres` but only 10 counts of `survey`. The following happens:

- If neither feature is marked as "partial", the license server sends the 5 counts of `highres` only. No `survey` licenses are included in the capability response because the license server cannot satisfy all 15 counts requested.
- If both features are marked as "partial", the license server sends the 5 counts of `highres` and the available 10 `survey` licenses in the capability response.

Example 2

The FlexNet Embedded client sends a capability request for 5 counts of the `highres` feature and 15 counts of `survey`. The license server currently has only 4 counts of `highres` and 10 counts of `survey`. The following happens:

- If neither feature is marked as "partial", the license server sends no features in the capability response since it cannot satisfy the requested count for either feature.
- If the `highres` feature is marked as "partial" but the `survey` feature is not, the license server sends the remaining available 4 counts of `highres` in the capability response but includes no `survey` licenses.

- If both features are marked as "partial", the license server sends the remaining 4 counts of `highres` and the remaining 10 counts of `survey` in the capability response.

Considerations and Limitations

Note the following about partial-checkout features:

- The availability of features on the license server depends on the collective activities of all FlexNet Embedded clients in the enterprise. Therefore, if a client resends a capability request for partial-checkout features, the resulting capability response can include counts different from those returned when the request was sent previously.
- Partial-checkout features can be metered or concurrent and are compatible with the use of license reservations on the license server.

  For more information about license reservations, refer to the *FlexNet Embedded License Server Producer Guide*, specifically the "More About Basic License Server Functionality" chapter and the "Effects of Special Request Options on the Use of Reservations" appendix.
- These features are compatible with incremental capability requests (see Incremental Capability Requests).
- They are compatible with the capability requests defined with the "request" operation type only.

Feature Selectors in a Capability Request

The license server always tries to satisfy a FlexNet Embedded client's request for a desired feature by serving a feature that matches three basic criteria—feature name, version, and count—as specified in the capability request. However, in some cases, additional criteria might be needed to ensure proper feature distribution to clients. For example, if the cost or availability of a feature varies by region and department, the client can include these attributes in the capability request. The license server then uses these attributes to filter versions of the desired feature and serve the feature appropriate to the client's region and department.

To enable filtering on a feature by one or more attributes in addition to the basic criteria, the producer must set up each additional attribute as a separate *feature selector*—a key-value structure included as part of the feature's definition created in the back office and stored with the feature on the license server. The client code can then use the `AddFeatureSelectorItem` method in the `ICapabilityRequestOptions` interface to specify the feature selectors in the capability request. The feature is served only if all its criteria, including the selectors, in the capability request match the feature's criteria on the license server.

For information about specifying feature selectors in the capability request, see the following:

- Specifying Feature Selectors in the Capability Request
- Considerations and Limitations

For more information about the setup of feature selectors in the back office and their storage on the license server, refer to the *FlexNet Embedded License Server Producer Guide*

Specifying Feature Selectors in the Capability Request

The following sample implementation shows how to specify feature selectors in the capability request—in this case, one selector using the key "`REGION`" and value "`EMEA`" and the other using the key "`DEPARTMENT`" and value "`Acct`". (Feature selectors are defined as capability-request options.)

```
ICapabilityRequestOptions options =
        licensing.LicenseManager.CreateCapabilityRequestOptions();
```

```
options.AddFeatureSelectorItem("REGION", "EMEA");
options.AddFeatureSelectorItem("DEPARTMENT", "Acct");
```

Considerations and Limitations

Note the following about including feature selectors in the capability request:

- If feature selectors are included in the capability request, they must match all selectors stored for the feature on the license server; otherwise, the feature is not served. No partial matching is performed.
- If *no* feature selectors are included in the capability request, no filtering takes place on the license server; features are served based on their match to the basic criteria only—feature name, version, and count—defined in the request.
- The *value* element in the key-value pair specified in the `FlcCapabilityRequestAddFeatureSelectorStringItem` API to identify a feature selector is case-insensitive and supports UTF-8 characters. Additionally, *value* must be a string, not an integer.
- The order of the feature selectors in the capability request does not affect the matching process on the license server.
- The set of feature selectors in the capability request applies to all desired features listed in the request. Only those features matching all the criteria are served.
- Best practice is to *not* include feature selectors in capability requests for clients that have reservations on the license server. This practice helps to avoid possible waste of reserved licenses.
- Feature selectors are compatible with the capability-request operation types “request” and “preview”, but are *not* compatible with “report” and “undo”.

Secondary Hostids

The client code can use the `AddAuxiliaryHostId` method in the `ICapabilityRequestOptions` interface to add a secondary hostid to the capability request. A secondary hostid is called as such because it is “secondary” to the main hostid, typically a unique client-device hostid, to which licenses are bound on the client. In short, the secondary hostid is simply a value that provides information that you want to save in the client record on the license server (and thus synchronize to the back office for reporting purposes).

Used to Implement User-Based License Reservations

One use for a secondary hostid in the capability request is to identify an entity, typically a user, for which the license server can search for license reservations when satisfying the request. (Compare user-based reservations with device-based reservations, which are identified by a unique client-device hostid. Features reserved for a user using the secondary hostid can be requested from any machine, whereas features reserved for a device must be requested from that device only. The license server administrator can set up a combination of both types of reservations on the license server.)

Note • *Because user-based reservations can be shared across different client devices (hence, different device hostids) and device-based reservations are bound to a single device hostid, capability requests should not attempt to specify the same hostid as a main hostid in one request and a secondary hostid in another.*

If multiple secondary hostids are included in the capability request, the license server uses the main hostid (to identify the client device) and only the *first* secondary hostid listed (to identify the user) for which to search for reservations. However, all secondary hostids included in the request are synchronized to the back office.

Secondary hostids can be of any hostid type (for example, ETHERNET, STRING, USER, or other). As producer, you must inform the license server administrator which hostid types you allow for secondary hostids used for reservations.

For more information about license reservations, see the "More About Basic License Server Functionality" chapter in the *FlexNet Embedded License Server Producer Guide*. For information about the license server administrator's role in setting up reservations, see the *FlexNet Embedded License Server Administration Guide*.

Including Secondary hostids in the Capability Request

The following sample implementation shows how to add a secondary hostid to the capability request:

```
// create the capability request options object
ICapabilityRequestOptions options =
          licensing.LicenseManager.CreateCapabilityRequestOptions();
// add an auxiliary hostid
options.AddAuxiliaryHostId(HostIdEnum.FLX_HOSTID_TYPE_USER, "ralph");
```

When multiple secondary hostids are added, the license server considers only the first secondary hostid (along with the main hostid) in its search for license reservations to satisfy the request.

Option to Force a Capability Response

The capability request can include a "force response" flag, set with the `ForceResponse` method in the `ICapabilityRequestOptions` object, to indicate that the FlexNet Embedded client always requires a capability response. However, this option is mainly used in capability exchanges with the back office. The local license server and the CLS license server ignore the "force response" flag and always send a capability response to the client.

Note however that, if a capability request is sent to a CLS license server *and* client registration is enabled in the back office, the CLS license server makes an additional call to the back office to register the client before sending the initial capability response to the client. Normally, this is a one-time registration call that is not repeated for subsequent capability requests. However, if the "force response" flag is included in the capability request, a call to the back office is made (and back-office response information generated) *each time* a capability request is sent to the CLS license server, resulting in increased response time. Hence, using the flag in this situation is strongly discouraged and typically unnecessary.

Borrow Interval and Granularity Overrides

Features checked out from the license server operate under a "borrow interval". The borrow interval is the maximum amount of time that a client can borrow a feature from the license server. Once the borrow period expires, the feature is no longer available for acquisition on the client. The license server checks for served expired features at regular intervals to ensure that the counts for any expired features are added back into the server's license pool. The client must send another capability request to the license server to borrow the feature again.

The following sections provide more information about how the borrow interval is ultimately determined:

- How the Borrow Interval Is Determined
- Borrow Interval Overrides

How the Borrow Interval Is Determined

The borrow interval can be set for a particular feature in FlexNet Operations, either through back-office configuration, which sets a default value for the License Fulfillment Service, or through the license model. In addition, a borrow interval can be set in a client request or using a configuration parameter. If the feature borrow interval has been set in the back office, the actual borrow interval is the lowest of the following values:

- feature borrow interval (set in the back office)
- client borrow interval (set in a client capability request)
- admin borrow interval (set using the configuration parameter `licensing.borrowIntervalMax`); default value: 0 (not configured)

However, if no borrow interval is set for a feature in the back office, then the borrow interval is the lowest of the following values:

- server borrow interval (defined in `producer-settings.xml` by the property `licensing.borrowInterval`); default value: 7 days
- client borrow interval (set in a client capability request)
- admin borrow interval (set using the configuration parameter `licensing.borrowIntervalMax`); default value: 0 (not configured)

If requested features end up having different borrow intervals, then the lowest borrow interval assigned to a given feature is applied to all the features served from that request.

Additionally, a borrow-interval granularity is applied to the borrow interval. The granularity is the time unit (day, hour, minute, or second) by which the license server *rounds up* the borrow interval. By default, this is set on the license server, and the default is **second**. For example, if the borrow interval is 1 minute, and the borrow granularity is **day**, then a license issued at 5:05:01 PM expires at 11:59:59 PM—which is the borrow interval (5:06:01 PM) rounded to the *end of the nearest day*.

A feature's current borrow expiration can never exceed the final expiration time for that feature. Should the borrow expiration be greater than the feature's final expiration, the borrow period is shortened to the final expiration time.

Borrow Interval Overrides

The capability request can override the borrow interval at the request-message level for all the features being requested as long as the override value *in conjunction with the borrow interval granularity* is less than the borrow interval defined for the individual features in the back office. Set this override using the `BorrowInterval` method in the `ICapabilityRequestOptions` object.

Additionally, the capability request can override the granularity defined on the license server for the borrow interval. This override is set using the `ExpirationValidationIntervalEnum` method in the `ICapabilityRequestOptions` object.

For more information about these APIs, consult the API reference.

License Checkout from the License Server

If the license server has sufficient counts to satisfy the requested counts for the desired features specified in a capability request, the server sends the licenses for the requested features in the capability response. However, by default, if the license server has an insufficient count to satisfy a given desired feature, the license for that feature is not granted. Also, by default, if no desired features are included in the capability request, no licenses are served to the client.

When a capability response is processed into the client's trusted storage, it replaces any previously available licenses.

Exceptions to any of this behavior can occur when license reservations are used during the checkout process (see the "More About Basic License Server Functionality" chapter in the *FlexNet Embedded License Server Producer Guide*) or when certain options that affect license checkout are included in the request (see the previous section Additional Capability-Request Options).

The FlexNet Embedded client can also request a preview of available licenses on the license server before sending a capability request to check out features. See the later section, Capability Preview.

Capability Preview

The FlexNet Embedded client application can preview features currently available to it on the license server by sending a capability request marked for "preview" purposes. In return, the license server sends a capability response specifying the available features, but the features are for preview only. In a "preview" capability exchange, the licensing state of the license server and the client does not change. That is, no feature, reservation, or client records are updated on the license server; and no licenses are processed into the client's trusted storage.

In setting up a capability request to preview licenses, the client application can request the availability of specific desired features or can request a preview of all available features (with all versions and available counts). Additionally, feature selectors can be included in the preview capability request to enable the license server to filter the available features.

The following sections describe the capability preview feature:

- Types of Preview Counts
- Creating a Preview Capability Request
- Processing the Preview Capability Response
- Creating a Regular Capability Request Based on Preview Features
- Other Considerations

Types of Preview Counts

The preview capability response returns two types of count for each feature available to the client:

- **Count**—The feature count, as determined by what the capability request has asked to preview:
  - If it requested to preview a particular desired feature (with a specific version and count), the returned value is the count that would be served had the client provided the license server with a regular (that is, non-preview) capability request for the same desired feature.
  - If it requested to preview all available features, the returned value for each feature shows the immediately available count—that is, the count reserved for the client plus all shared counts that are not currently served to other clients.
- **Maximum count**—The potentially available count that includes the count reserved to the client plus all shared counts, whether currently served to other clients or not. (In other words, this count assumes that all shared counts are available.)

See Other Considerations for factors that can affect the calculation of these two counts.

Based on the preview results, your application can then generate a regular capability request to check out the available features.

Creating a Preview Capability Request

The **CapabilityRequest** example includes sample code that sets up a preview capability request. The following sections highlight the code used for this process:

- Set the Preview Operation
- Request to Preview Specific Features or All Features
- Specify Feature Selectors

Set the Preview Operation

To generate a capability request to preview features available to the client, set the "preview" operation in the `ICapabilityRequestOptions` interface, as shown in the following sample implementation:

```
ICapabilityRequestOptions options = licensing.LicenseManager.CreateCapabilityRequestOptions();
options.Operation = CapabilityRequestOperation.Preview;
```

The "preview" operation is incompatible with the "incremental" attribute, features set as "partial", and the correlation ID. Hence, do not specify these properties when setting up the preview capability request:

- `IDesiredFeatureOptions.PartialFulfillment` set to `true`
- `ICapabilityRequestOptions.CorrelationId`
- `ICapabilityRequestOptions.Incremental` set to `true`

However, the "preview" operation is compatible with feature selectors. See Specify Feature Selectors.

Request to Preview Specific Features or All Features

The capability request can specify a preview of one or more desired features (with a specific version and count). Alternatively, the request can specify a preview *all* available features. However, the request cannot specify a preview of both specific desired features and all available features, as pointed out in the next sections.

Preview Availability of Specific Desired Features

To request a preview of a desired feature with a specific version and count, use the `AddDesiredFeature` method in the `ICapabilityRequestOptions` interface, as shown in the following sample implementation:

```
ICapabilityRequestOptions options = licensing.LicenseManager.CreateCapabilityRequestOptions();
options.Operation = CapabilityRequestOperation.Preview;
options.AddDesiredFeature(new FeatureData("f1", "1.0", 5));
```

This is the same method used in regular capability requests to check out particular count of a desired feature; but, for the "preview" operation, the capability response returns this feature count for viewing purposes only. (The preview count is the same as the count returned in a regular capability response for the same desired feature.)

Do not use this method with the `ICapablityRequestOptions.RequestAllFeatures` property set to `true`. You can either request all features or add desired features, not both.

Preview All Available Features

To request a preview of all available features (including all versions and total available counts), use the `RequestAllFeatures` property (set to **`true`**) in the `ICapablityRequestOptions` interface, as shown in the following sample implementation:

```
ICapabilityRequestOptions options = licensing.LicenseManager.CreateCapabilityRequestOptions();
options.Operation = CapabilityRequestOperation.Preview;
options.RequestAllFeatures = true;
```

Note the following restrictions when using the `RequestAllFeatures` property (set to **`true`**) in a preview capability request:

- Do not use this property with the `ICapablityRequestOptions.AddDesiredFeature` method in the same preview capability request. You can either request all features or add desired features, not both.
- This property can be used only with the "preview" capability request operation (see Set the Preview Operation).

Specify Feature Selectors

The license server processes any feature selectors provided in the preview capability request as a means of filtering features to include in the preview capability response. Feature selectors are applied to candidate features whether the request calls the `ICapablityRequestOptions.AddDesiredFeature` method or uses the `ICapablityRequestOptions.RequestAllFeatures` property.

See Feature Selectors in a Capability Request for information about providing these selectors in the request.

Processing the Preview Capability Response

The following sections highlight code excerpts in the **CapabilityRequest** example to demonstrate how to set up the preview output:

- Determine the Response Type
- Inspect the Preview Features
- Display the Preview Features

Determine the Response Type

When the capability response is received by the client, your code can access the `IsPreview` property in the `ICapabilityResponse` interface to determine whether the response is marked as "preview". If it is, the response cannot be processed into trusted storage, but the feature details within the response can be inspected. The following sample code shows an implementation of this functionality:

```
if (response.IsPreview)
{
    // Inspect returned preview response feature information
}
```

Inspect the Preview Features

The **CapabilityRequest** example code uses the custom method ShowPreviewResponse to initiate an inspection of the preview features. This method creates a feature collection from the preview capability response and then calls another custom method, ShowCapabilityResponseFeatures, to format and display the preview feature details. This second custom method iterates through the feature collection and extracts details for each preview feature, including its count and maximum count. It then prints this information as the preview output.

Two important details to retrieve for each preview feature are the counts. The code can get these counts using the following properties:

- The IFeature.Count property retrieves the count for the preview feature. (See Types of Preview Counts for a description of this count. The Count property is also used in regular capability responses to retrieve the served count for the feature.)
- The IFeature.MaxCount property retrieves the maximum count for the feature. (See Types of Preview Counts for a description of the maximum count.)

This code excerpt from the custom ShowCapabilityResponseFeatures method in the **CapabilityRequest** example shows a sample implementation for examining the preview capability response:

```
IFeatureCollection collection = response.FeatureCollection;
int index = 1;
foreach (IFeature feature in collection)
{
    StringBuilder builder = new StringBuilder();
    builder.Append(String.Format("{0}: {1} {2}", index, feature.Name, feature.Version));
    if (feature.IsPreview)
    {
         builder.Append(String.Format(" TYPE=preview COUNT={0} MAXCOUNT={1}", feature.IsUncounted
                                 ? "uncounted" : feature.Count.ToString(),
            feature.MaxCount == feature.UncountedValue ? "uncounted" :
                                      feature.MaxCount.ToString()));
    }
```

Display the Preview Features

The following shows sample output from the **CapabilityRequest** example when it is run with a “preview” capability request:

```
INFO: Features loaded from trusted storage: 2
INFO: Creating the capability request
INFO: Sending the capability request to: http://localhost:7070/request
INFO: Response received
INFO: Examining preview capability response
INFO: Obtaining capability response details
INFO: Machine type: PHYSICAL
INFO: Capability response contains 0 vendor dictionary items
INFO: Capability response contains 0 status items
INFO: Confirmation request is not needed
INFO: ============================================
INFO: Features found in preview capability response:

INFO: 1: f1 1.0 TYPE=preview COUNT=5 MAXCOUNT=10 EXPIRATION=7/21/2025 2:14:23 PM
VENDOR_STRING="vendor defined" ISSUED=7/12/2016 12:00:00 AM START=7/11/2016 12:00:00 AM
```

The output includes the following “preview” information:

- Statements indicating that the capability response is a preview response.
- A line for each preview feature if the feature is available (that is, it does not have a count of 0). The line includes the feature's name (`f1`), version (`1.0`), count (5), and maximum count (10). For a description of these counts, see Types of Preview Counts. Additionally, for factors that affect these counts, see Other Considerations.

Creating a Regular Capability Request Based on Preview Features

The preview information can be used as a basis for setting up a regular (non-preview) capability request to obtain available features. The following describes some methods for doing this:

- Set Up a Regular Request
- Determine Desired Features for the Regular Request

Set Up a Regular Request

To generate a regular capability request to check out features that you have previewed, your application code can create a new `ICapabilityRequestOptions` object and build the desired features from scratch.

Alternatively, your code can modify the original `ICapabilityRequestOptions` object, adjusting the desired features as needed and changing the operation type of the capability request to "request". If the original `ICapabilityRequestOptions` object used the `RequestAllFeatures` property, your code needs to change this property to "false" (and build the desired features from scratch) when reusing the object:

```
ICapabilityRequestOptions options = licensing.LicenseManager.CreateCapabilityRequestOptions();
options.RequestAllFeatures = false;
options.Operation = CapabilityRequestOperation.Request;
```

Determine Desired Features for the Regular Request

The application can use information from the preview feature collection to build the desired features for the regular capability request.

The application developer must be aware that the counts returned in the preview capability response represent a "moment in time" on the license server. These counts can change anytime between the preview and a regular capability exchange that attempts to check out features available in the preview. Feature availability on the license server can fluctuate when events such as these occur:

- The license server receives an update from the back-office server.
- Other clients check out the available shared counts.
- The license server administrator changes reservations.

Best practice is to act on the preview information within a certain window of time instead of storing it for future use.

Other Considerations

Note the following about interpreting the capability preview:

- Because the "incremental" attribute is not supported for preview capability requests, the count and the maximum count do not depend on the counts currently served to the client. (See Incremental Capability Requests for information.)

- Expired or non-started features are not included in the preview capability response.
- The count and maximum count are computed from both regular and overdraft counts.
- Only features with non-zero counts are included in the capability response. This behavior can limit the usefulness of the maximum count in certain circumstances, such as when all counts of a feature are currently being served to other clients or when multiple features with the same name are present on the license sever.
- For a metered feature, the immediate count is computed the same as it is for a concurrent feature. However, the maximum count is based on the premise that "served counts" are considered permanently consumed. See the *FlexNet Embedded License Server Producer Guide* for examples of how the license server handles counts for metered features in a capability preview.

Limited-duration Trials

Trials allow producers to enable functionality on a client system for a specified duration, after which the functionality becomes disabled. This is useful in situations where the producer may want to allow users to try out new features before buying them. The following is an overview of the steps required by the implementer to enable trial functionality in license-enabled code running on the client system.

Overview of the Trial Scenario

Trial Preparation

- Create the Binary Trial License Rights

Getting and Using the Trial on the Client System

- Step 1: Create and Populate the License Sources
- Step 2: Get Trial Data from the Binary Trial File

Trial Preparation

Create the Binary Trial License Rights

Analogous to creating a binary signed license file for scenarios involving binary buffer licenses, license rights to be stored in a client system's trial storage are stored in a binary file to be processed by license-enabled code. In addition to containing feature definitions (`INCREMENT` lines), trial rights include a duration, a product ID, and a unique numeric trial ID.

(By default, trial license rights are defined as load-once, meaning that once the trial is processed and the trial duration begins, the trial cannot be loaded again. Trials can also be defined as load-always, meaning that the trial can be loaded multiple times to extend the trial duration, but this type of trial is rarely used.)

The FlexNet Embedded Client .NET XT or .NET Core XT toolkit provides a `trialfileutil` utility for creating signed binary trial license rights, based on an unsigned text license file. Using `trialfileutil`, the process is the following:

First, create the unsigned license file, adding the following two feature definitions to a text file called `test.lic`:

```
INCREMENT survey demo 1.1 permanent uncounted
INCREMENT highres demo 1.1 permanent uncounted
```

To create the binary trial file, run the following command to create a trial with a ten-day duration:

```
trialfileutil -id IdentityBackOffice.bin -product SampleProduct -duration 864000 -trial 1 test.lic
    trialtest.bin
```

See Trial File Utility for more information about the `trialfileutil` command-line arguments.

The *`install_dir`*`\examples\client_samples\Trials` directory contains the source code for the **Trials** example.

Running the `Trials` executable with the `-h` or `-help` switch displays usage information.

Getting and Using the Trial on the Client System

The following steps are to be performed in the code that will be compiled to run on the client system, in order to get the license rights defined in the trial source. It is assumed the code has initialized the main licensing objects as described in Creating Core Licensing Objects.

Perform the following steps:

- Step 1: Create and Populate the License Sources
- Step 2: Get Trial Data from the Binary Trial File

Step 1: Create and Populate the License Sources

This step is identical to the corresponding step in Buffer Licenses, Step 1: Create and Populate the License Sources, except that a trials license source must be created.

```
// Add trial license source
licensing.LicenseManager.AddTrialLicenseSource( );
```

Note that some operations, such as processing a binary trial file, automatically create a trials license source.

If trials storage has been previously used to store trials, the trials license source will contain license rights from these trials, and these rights are immediately available for acquisition (assuming the trial duration has not elapsed). If trials storage has not been previously used, the trials license source will have no license rights. In either case, to use functionality specified in the new (so far unused) trial, the license-enabled code must process the trial into the trial license source as described below.

Step 2: Get Trial Data from the Binary Trial File

This step is similar to the corresponding steps in Using the License on the Client, except that the data read from the binary file represents the trial data and the license source used to process the data is a trials license source.

For this sample code, the trial data is read directly from a specified input file into an input buffer. Trial information can instead be embedded in the executable binary, by converting the binary trial file into a C#-compatible array (using the `printbin` utility) and including the array in the application code, similar to how identity data is handled.

In order to use license rights from the trial, the license-enabled code processes it into a trials license source.

```
// determine if trial has been loaded
DateTime expirationDate;
if (licensing.LicenseManager.TrialIsLoaded(inputFile, out expirationDate))
{
```

```
    if (expirationDate.CompareTo(DateTime.Now) > 0)
    {
        Util.DisplayInfoMessage(
            string.Format("Trial has already been loaded and will expire on {0}", expirationDate));
    }
    else
    {
        Util.DisplayInfoMessage("Trial has already been loaded and has expired");
    }
}

// process trial into trials license source
try
{
    licensing.LicenseManager.ProcessTrial(trialFile);
}
catch (PublicLicensingException licensingException)
{
    switch (licensingException.ErrorCode)
    {
        case ErrorCode.FLXERR_TRIAL_ALREADY_LOADED:
        case ErrorCode.FLXERR_TRIAL_EXPIRED:
            HandleException(licensingException);
            return false;
        default:
            throw licensingException;
    }
}
Util.DisplayInfoMessage(
    string.Format("Number of features loaded from trials: {0}", CheckNumberOfFeatures( )));
```

(To combine the operations of adding a trial license source and processing the trial, some forms of the `AddTrialLicenseSource` method accept the trial byte array, a stream, or a file path. Similarly, `ProcessTrial` will create the license source if not already created.)

When a new trial is stored in the trial license source, it does not overwrite the previously processed trials in this license source. If the particular trial has been already processed into the license source, no changes are made to the trial license source. The license rights from the processed trial will reside in the trial license source, even after the license source is deleted or the license-enabled executable code is terminated. (Note that calling `AddTrialLicenseSource` multiple times for the same load-always trial—as opposed to the more typical load-once trial type—will result in multiple copies of the trial's features in trials storage.)

Once license rights have been processed into trials trusted storage, the rights are available to be acquired from license-enabled code. The process is the same as the other scenarios (such as Step 2: Acquire the License(s) in Buffer Licenses).

See the `Trial.cs` source code in *`install_dir`*`\examples\client_samples\Trials` for more information.

Secure Re-hosting

Secure re-hosting enables producers to support moving functionality from one client system to another, with the option of verifying that functionality has been removed from the source system.

In order to make re-hosting secure, the involvement of the back office is required; re-hosting is based on the capability request/response functionality. See Licenses Obtained from the Back-Office Server for details on how to create license-enabled code that supports capability request and response processing.

The following description provides an overview of the steps required by the producer to perform secure re-hosting of capabilities from Host A to Host B.

Removing Capabilities from Host A:

- Step 1: Start License-Enabled Code on Host A
- Step 2: Submit Capability Request from Host A to the Back-Office Server
- Step 3: Back-Office Server Processes Request and Sends "Reduced" Response Back to Host A
- Step 4: Process "Reduced" Capability Response on Host A
- Step 5: Submit Another Capability Request from Host A to the Back-Office Server
- Step 6: Back-Office Server Processes Capability Request from Host A

Adding Capabilities to Host B:

- Step 7: Start License-Enabled Code on Host B
- Step 8: Submit Capability Request from Host B to the Back-Office Server
- Step 9: Back-Office Server Processes Request and Sends Response Back to Host B

Removing Capabilities from Host A

Use the following steps to remove capabilities from Host A.

Step 1: Start License-Enabled Code on Host A

Since the re-hosting scenario is based on capability request/response processing, the license-enabled code on Host A must support a trusted storage license source. See Licenses Obtained from the Back-Office Server for details.

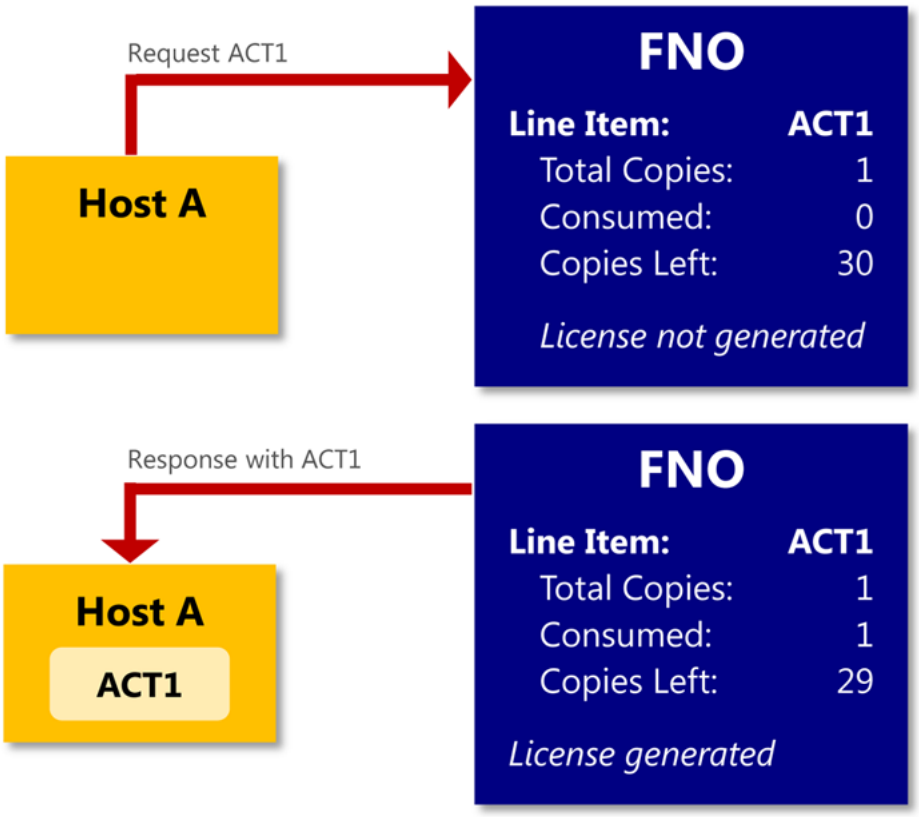
After the trusted storage license source is created, the license-enabled code can verify that functionality targeted for re-host is available on Host A. This can be done by acquiring licenses and reading license details from the trusted storage license source. See the corresponding steps in the previous examples for details. This part is optional.

(FlexNet Operations displays the state "License generated" for the add-on, when the host has the original license rights.)

Step 2: Submit Capability Request from Host A to the Back-Office Server

This step is similar to the corresponding step in the Licenses Obtained from the Back-Office Server walkthrough.

The following figure summarizes the initial activation steps between the host and FlexNet Operations.



Step 3: Back-Office Server Processes Request and Sends "Reduced" Response Back to Host A

This step is similar to the corresponding step in the Licenses Obtained from the Back-Office Server scenario, except that the capability response that is sent back to the client system contains no license rights (or at least reduced license rights, in the case of a partial re-host operation). This is done using logic implemented by the producer in the back-office server that recognizes that capabilities residing on Host A are targeted to be moved to Host B. (In FlexNet Operations, this is handled by removing the corresponding line item from Host A, which changes the line item state to "Marked for removal". For a partial re-host, lowering the number of copies changes the item state to "Copies Decreasing".) The first step in this re-hosting transaction is to take away capabilities from Host A, which is achieved with an "empty" or reduced capability response.

(In FlexNet Operations, a scenario where only some license rights have been removed from a client system corresponds to removing only some add-on line items from a client system. FlexNet Operations displays the add-on state "Removed from license" after sending the capability response without the removed line item.)

Step 4: Process "Reduced" Capability Response on Host A

This step is similar to the corresponding step in Licenses Obtained from the Back-Office Server scenario. Because the response is "empty" or contains reduced license rights, only the reduced licenses—along with those defined in local license files or trials—are available on Host A after the capability response is processed. (A response with no license rights is different from a zero-byte or missing response, which can signify that license rights have not changed since the last response.)

This can be verified by the attempt to acquire licenses previously available in the trusted storage license source. The attempt should fail indicating that no matching license was found. See the corresponding steps in the previous examples for details on license acquisition. This part is optional.

Step 5: Submit Another Capability Request from Host A to the Back-Office Server

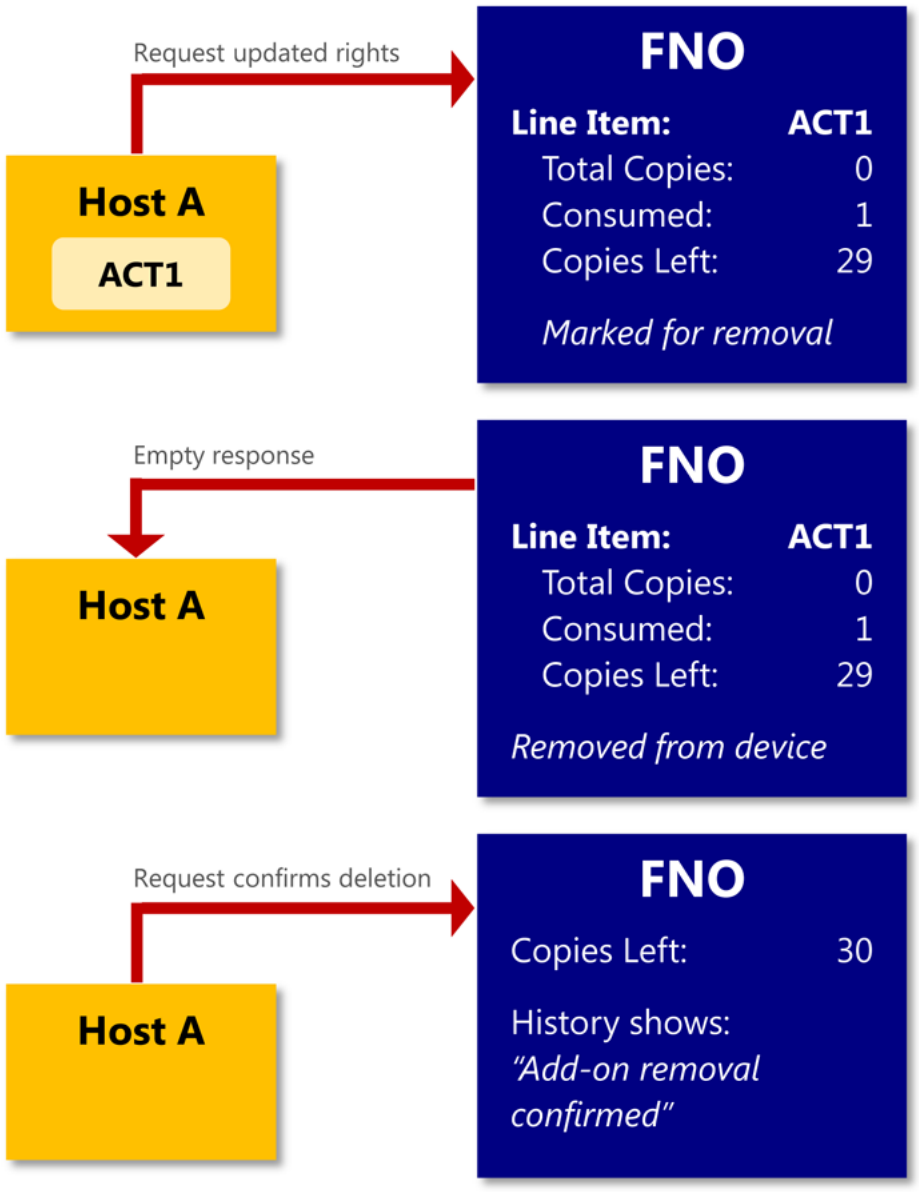
This step is similar to Step 2: Submit Capability Request from Host A to the Back-Office Server.

Step 6: Back-Office Server Processes Capability Request from Host A

This step is similar to the corresponding step in Licenses Obtained from the Back-Office Server scenario, except for additional logic implemented by the producer in the back-office server. This logic is based on the fact that capability request includes the reference to the last capability response processed on the client system generating the request. This is important for the secure re-host of capabilities.

The last response processed on the Host A was “empty” or contained reduced license rights, and the new request generated on Host A has a reference to this reduced response. The re-host logic in the back-office server recognizes this reference and uses it as evidence that Host A has indeed processed the reduced response and contains only the license rights from the reduced response. (FlexNet Operations completely removes the line item from Host A only after receiving this confirmation, and the host’s history shows the action “Add-on Removal Confirmed”.)

The following figure summarizes the communications between the host and FlexNet Operations to complete the transfer of licenses from Host A.



At this point, the back-office server is ready to grant capabilities to Host B when requested.

Adding Capabilities to Host B

Use the following steps to add capabilities to Host B.

Step 7: Start License-Enabled Code on Host B

This step is similar to Step 1: Start License-Enabled Code on Host A, except that after the trusted storage license source is created, the code can verify that functionality targeted for re-host is not yet available on Host B. This can be done by an attempt to acquire licenses targeted for re-host. The attempt should fail, indicating that no matching license was found. See the corresponding steps in the previous examples for details on license acquisition. The license verification step is optional.

Step 8: Submit Capability Request from Host B to the Back-Office Server

This step is similar to Step 2: Submit Capability Request from Host A to the Back-Office Server.

Step 9: Back-Office Server Processes Request and Sends Response Back to Host B

This step is similar to the corresponding step in the Licenses Obtained from the Back-Office Server scenario.

Note that the back-office server can send license rights through a capability response to Host B, based on the previously received evidence that corresponding capabilities were removed from Host A.

Capturing Feature Usage on the Client

To support various pay-for-use and pay-for-overage license models, FlexNet Embedded functionality—in conjunction with FlexNet Usage Management—supports *metered* licenses. Usage information for these metered licenses can be captured and sent to the back office (FlexNet Operations), after which reporting, reconciliation and billing operations can be performed.

The **UsageCaptureClient** example demonstrates how a usage data is captured on the client and channeled through a license server—either a CLS (Cloud Licensing Service) license server or the FlexNet Embedded local license server—to the back office. The example simulates the expected deployment architecture, which is a client that has a persistent online connection to the license server.

The two primary scenarios illustrated by the **UsageCaptureClient** example are:

- Uncapped Usage Capture: The software can report any amount of feature usage, with no upper limit on the amount of usage allowed.
- Capped Usage Capture: The software can report feature usage up to a limit, possibly with some amount of overage allowed. If the cap is exceeded, the license server responds to the client that the cap has been exceeded, and the implementer can decide what to do in such an event. Note that capped usage is expected to be used in an environment in which the client has a persistent connection with the license server.

More About the Example

The source code the **UsageCaptureClient** example described in this walkthrough can be found in the directory *`install_dir`*`\examples\client_samples\UsageCaptureClient`, in the source file `UsageCaptureClient.cs`.

To display command-line usage for the example, run the example's executable with the `-h` or `-help` switch. Demonstrations of certain usage are presented later in this section.

Topics Covered in this Section

The following is the complete list of topics about usage capture covered in this section:

- Capability Requests and Usage Capture
- Preparation in FlexNet Operations
- License Source Creation
- Client Registration with the Cloud Licensing Service
- Uncapped Usage Capture
- Capped Usage Capture
- Post-Usage-Capture: Managing Usage Data
- Additional Metered License Attributes

Capability Requests and Usage Capture

The usage-capture client uses capability requests to communicate with the CLS license server or the FlexNet Embedded local license server, similar to capability requests described in Licenses Obtained from a License Server. The primary difference is that usage-capture clients use capability requests to transport usage information to the server.

The following sections describe capability request attributes used with usage capture:

- Operation Type
- Correlation ID
- Other Optional Identifiers
- Desired Features and Rights IDs

Operation Type

The capability request can indicate the type of operation it is meant to perform.

- **Request:** A “request” operation indicates that the client expects a response that contains license rights from the license server. In a usage-capture scenario, the “Request” operation is used mainly for *capped* usage capture, where the response informs the client whether the cap for feature usage has been exceeded. (In a non-usage-capture scenario, the response enables license rights on the client so that client code can acquire and return features as needed.)

  If no explicit operation type is specified, it is assumed to be a “request” operation.

- **Report:** A “report” operation indicates that the client does not expect a response that contains license rights; the capability request is simply reporting usage data. However, the license server might still send an error response with any error information.

- **Undo:** An “undo” operation can recall an erroneous prior request containing usage data. (There is typically a limited amount of time during which an “undo” operation can be used. The **Undo Interval** is defined by the producer as a feature attribute, typically as part of a license model.) An “undo” operation requires a correlation ID, which is described in the following section.

An implementation calls `CreateCapabilityRequest` with the appropriate `ICapabilityRequestOptions Operation` property to specify the operation type.

Limitation: No Mixing of Operation Types

When a given FlexNet Embedded client sends a mix of "request" and "report" capability requests, its trusted storage might not reflect accurate feature counts. Therefore, capability requests from a given client should either all use the "request" operation or all use "report". (When both metered and non-metered features are available to a client, only "request" capability requests should be used.)

The "undo" operation can always be used as needed to recall usage data sent in "request" capability requests. See Recall a "Used" Metered Feature.

Correlation ID

The *correlation ID* is generated for any capability request that results in updates to client data on the license server—whether the request is for metered features or concurrent features. However, only an "undo" operation, which recalls an erroneous or canceled previous request *containing usage data*, actually uses this ID so that it can identify the request being recalled.

The correlation ID—an arbitrary string, possibly a UUID value—is automatically generated for a capability request by the license server. It is stored in the client record on the license server (and eventually synchronized to the back office) and is sent back in the capability response, if a response is required. For more information about specifying the correlation ID when requesting an "undo" operation, see Recall a "Used" Metered Feature.

Other Optional Identifiers

The following optional identifiers can be set in a capability request. These identifiers do not affect licensing behavior, but can be used for reporting purposes. In the example **UsageCaptureClient** code, these identifiers are hard-coded for use in the capability request.

- **Acquisition ID:** Identifies the resource that was acquired.
- **Requestor ID:** Identifies the user associated with the client device.

A third optional identifier can be used in the capability request with caution:

- **Enterprise ID:** Identifies the end-user account on behalf of the acquisition performed. If a producer sends the end user an "enterprise ID" to use, the user can include it in the capability request for additional security. Otherwise, the user should not set one; the capability request will be rejected if the enterprise ID included in the request is different from the value expected by the license server.

Desired Features and Rights IDs

In usage-capture scenarios, the capability request specifies one or more desired features to indicate captured usage. (In non-usage-capture scenarios, desired features are used with license servers to indicate what features are to be sent in a response to a client request.)

To add desired-feature information to a capability request, the code uses `ICapabilityRequestOptions.DesiredFeatures.Add(new FeatureData(`*`data`*`))`.

Usage-capture scenarios that use a CLS license server might also require an initial capability request that sends a rights ID value to register the client with the server. To add the rights ID value to the capability request, the code uses `ICapabilityRequestOptions.AddRightsId(rightsId, copies)`. (The rights ID sent in the client request is equivalent to an *activation ID* in FlexNet Operations and to an *activation code* in the older FlexNet Operations On-Demand.) See Client Registration with the Cloud Licensing Service for more information.

Preparation in FlexNet Operations

FlexNet Operations, when coupled with FlexNet Usage Management, enables you to support usage-based licensing and compliance models. Whether using a CLS license server or the FlexNet Embedded local license server to channel usage information from the client to FlexNet Operations, you must prepare FlexNet Operations to work with the license server. This preparation involves creating and associating several entities using the FlexNet Operations Producer Portal. For a walkthrough on how to use the Producer Portal to prepare FlexNet Operations, see the section "Getting Started with Usage Management" in the *FlexNet Operations User Guide*.

As described in the section "Getting Started with Usage Management" in the *FlexNet Operations User Guide*, create your producer identity data in FlexNet Operations, and then download the client identity to a location where your FlexNet Embedded Client .NET XT or .NET Core XT toolkit can access it. (For the **UsageCaptureCient** example, download the file `IdentityClient.cs` to the directory *`install_dir`*`\examples\identity`.) You must build the example with this identity.

"Getting Started with Usage Management" shows how to add a feature (also called a *capability*) and define the license model associated with the feature. To run the scenarios in the **UsageCaptureClient** example, customize the feature and license-model setup as follows:

- Define a feature called "survey" with version 1.0.
- When defining the license model, set the **Is this a Counted Model?** attribute to **Yes** to establish a baseline count for setting a cap or for determining overage. See Additional Metered License Attributes for information about the baseline count.
- In the license model, set **Is this a Metered Model?** to **Yes** to identify the model as one used for usage capture and management.
- To demonstrate uncapped usage, set the **Overdraft** attribute to **`Unlimited`** in the license model, meaning that no limit exists to the number of licenses that can be used beyond the baseline count. See Additional Metered License Attributes for details.
- To demonstrate capped usage, set the **Overdraft** attribute to (for example) **`Not Used`** in the license model. (The **`Not Used`** value indicates that no overdraft is used—that is, no overage is allowed. For capped usage, this value can also be a fixed number or a percentage of the entitled amount.) See Additional Metered License Attributes for details.

In addition, when a CLS license server is used, the FlexNet Embedded client device might be required to register with the Cloud Licensing Service before it can request features or report usage. See the Client Registration with the Cloud Licensing Service for more information.

License Source Creation

In license-enabled code that has created the core objects, this step is identical to the corresponding step in Licenses Obtained from the Back-Office Server, Step 1: Create the License Source for creating a trusted-storage license source.

```
licensing.LicenseManager.AddTrustedStorageLicenseSource( );
```

Note that some operations, such as processing a capability response, automatically create a trusted storage license source.

Note • *Best practice is to reset existing trusted storage before running the example. To do so, call the method "ILicensing.Administration.Delete(TrustedStorage)", which deletes the contents of existing trusted storage.*

Client Registration with the Cloud Licensing Service

The configuration of a CLS license server can require that the FlexNet Embedded client device register with the Cloud Licensing Service as an extra security measure before allowing the client to request features or report usage. However, by default, this requirement is disabled. If using a CLS license server, consult the FlexNet Operations administrator to determine whether registration is required. If it is, use the information in this section to set up a separate capability request that initiates the registration.

Note • *This registration step is not needed when the client application is sending captured usage data to the FlexNet Embedded local license server.*

To perform the registration, the client application sends an initial capability request to the server, specifying a rights ID for an unmetered, uncounted license for the purpose of simply identifying the client device to the server. The rights ID used in this request has been previously conveyed to the end user—typically through an email message from the producer. (The rights ID sent in the client request is equivalent to an *activation ID* in FlexNet Operations and to an *activation code* in the older FlexNet Operations On-Demand.)

The **UsageCaptureClient** example accepts a `-rightsid` switch for specifying the rights ID for registration. When the switch is present, the example code uses the following implementation to call the method `options.AddRightsId`, which adds the specified rights ID to the capability request:

```
if (operation != CapabilityRequestOperation.Report && !string.IsNullOrEmpty(rightsId))
{
    options.AddRightsId(rightsId, 1);
}
```

The capability request used to register the client device must specify the "request" operation, also shown in this code implementation. (The **UsageCaptureClient** example accepts the `-request` switch to set the *request* operation type.) Specifying a rights ID is not compatible with the "report" operation in a capability request—a restriction enforced by the `if`-statement used in the sample code excerpt.

The command for the initial execution of **UsageCaptureClient** to register the client device is similar to the following:

```
UsageCaptureClient -request -rightsid ACT-ID-1 -server https://
    siteID-uat.compliance.flexnetoperations.com/instances/instId/request
```

The URL will typically have been conveyed to the end user by email from the producer. It identifies the *siteID*, which is the producer's specific site ID supplied by Revenera, and the *instId*, which is the server's instance ID on the Cloud Licensing Service. Note that the URL above points to a User Acceptance Test (UAT) environment indicated by the `-uat` part following the *siteID*. For production environments, the `-uat` is omitted.

Once this initial capability exchange to register the client device is complete, the end user can proceed to send capability requests for features. Subsequent runs of the **UsageCaptureClient** example sends capability requests that capture metered-license usage, as described next.

Uncapped Usage Capture

To implement uncapped-usage capture for a feature, your client code sends a capability request to the CLS license server or the FlexNet Embedded local license server to report the feature's usage, but the license server does not return license rights in the response. (The response is processed only to determine any error conditions.) The **UsageCaptureClient** example reports a hard-coded 1 unit of usage for the survey feature to the server each time the example is launched.

To indicate that no feature information needs to be returned, the code calls `CreateCapabilityRequest` with the `ICapabilityRequestOptions Operation` property set to `CapabilityRequestOperation.Report`. (In capped-usage scenarios, where a capability response is expected to return feature information to be processed into trusted storage, the `CapabilityRequestOperation.Request` value is used instead.)

```
// Create the capability request options object
ICapabilityRequestOptions options=licensing.LicenseManager.CreateCapabilityRequestOptions();

// Set the capability request operation
options.Operation = CapabilityRequestOperation.Report;

// Specify usage data; to return a metered feature, set the count to a negative value
options.DesiredFeatures.Add(new FeatureData("survey", "1.0", 1));

// Generate the request
ICapabilityRequestData capabilityRequestData =
    licensing.LicenseManager.CreateCapabilityRequest(options);

// Send the capability request to the server and receive the server response
CommFactory.Create(url).SendBinaryMessage(capabilityRequestData.ToArray(), out binCapResponse);
Util.DisplayInfoMessage("Successfully sent capability request");
if (operation != CapabilityRequestOperation.Report ||
    (binCapResponse != null && binCapResponse.Length > 0))
{
    ProcessCapabilityResponse(binCapResponse);
}
```

The example's `SendCapabilityRequest` method uses the `SendBinaryMessage` method from the `IComm` interface to send the binary capability request and receive the response. This same process is used by **CapabilityRequest** example. See Licenses Obtained from the Back-Office Server for more information about these communication methods.

To run an uncapped-usage capture using the **UsageCaptureClient** example, issue a command similar to one of the following, specifying the `-report` switch.

When using the CLS license server, specify its URL as the value for the `-server` command-line argument, where *siteID* is the producer's specific site ID supplied by Revenera and *instId* is the server's instance ID on the Cloud Licensing Service:

```
UsageCaptureClient -report -server https://siteID-uat.compliance.flexnetoperations.com/instances/
    instId/request
```

Note that the URL above points to a User Acceptance Test (UAT) environment indicated by the `-uat` part following the *siteID*. For production environments, the `-uat` is omitted.

When using the FlexNet Embedded local license server, specify the URL for the license server, as for example:

```
UsageCaptureClient -report -server http://localhost:7070/request
```

Capped Usage Capture

Implementing capped-usage capture is similar to uncapped-usage capture, except that the code expects feature information in the capability response from the CLS license server or the FlexNet Embedded local license server. The response is processed and queried to determine whether the usage quota has been exceeded. (Features are added to trusted storage if usage is still under quota.)

Preparation of FlexNet Operations is similar to preparation for uncapped-usage capture, except that the license model must specify a value other than **`Unlimited`** for the **Overdraft** setting. See Preparation in FlexNet Operations and Additional Metered License Attributes for details.

To implement this type of usage capture, the code calls `CreateCapabilityRequest` with the `ICapabilityRequestOptions` `Operation` property set to `CapabilityRequestOperation.Request` to request a response from the license server.

```
// Create the capability request options object
ICapabilityRequestOptions options=licensing.LicenseManager.CreateCapabilityRequestOptions();

// Set the capability request operation
options.Operation = CapabilityRequestOperation.Request;
```

As with the uncapped-usage capture scenario, the capped-usage capture scenario requests a hard-coded count of 1 copy of the feature survey, specified as a desired feature in the capability request. The capability request is then created and sent to the license server.

```
// Specify usage data; to return a metered feature, set the count to a negative value
options.DesiredFeatures.Add(new FeatureData("survey", "1.0", 1));

// Generate the request
ICapabilityRequestData capabilityRequestData =
    licensing.LicenseManager.CreateCapabilityRequest(options);

// Send the capability request to the server and receive the server response
CommFactory.Create(url).SendBinaryMessage(capabilityRequestData.ToArray(), out binCapResponse);
Util.DisplayInfoMessage("Successfully sent capability request");
if (operation != CapabilityRequestOperation.Report ||
    (binCapResponse != null && binCapResponse.Length > 0))
{
    ProcessCapabilityResponse(binCapResponse);
}

Util.DisplayInfoMessage("Processing capability response");
```

In this case, the feature information is returned in the response from the license server. The client processes the response, and, if usage quota has not been exceeded, the desired features are loaded in trusted storage.

```
ICapabilityResponse response =
    licensing.LicenseManager.ProcessCapabilityResponse(binCapResponse);
Util.DisplayInfoMessage("Capability response processed");
ShowCapabilityResponseDetails(response);
ShowTSFeatures();
AcquireReturn(surveyFeature, version);
```

The example then attempts to acquire the survey feature to determine whether the usage quota has been exceeded.

To run a capped-usage capture using the **UsageCaptureClient** example, issue a command similar to one of the following. The available `-request` switch, specifying a “request” operation, is the default option and therefore implicit in the command.

When the CLS license server, specify its URL as the value for the `-server` command-line argument, where *siteID* is the producer's specific site ID supplied by Revenera and *instId* is the server's instance ID on the Cloud Licensing Service:

```
UsageCaptureClient -server https://siteID-uat.compliance.flexnetoperations.com/instances/instId/
    request
```

Note that the URL above points to a User Acceptance Test (UAT) environment indicated by the `-uat` part following the *siteID*. For production environments, the `-uat` is omitted.

When using the FlexNet Embedded local license server, specify the URL for the license server, as for example:

```
UsageCaptureClient -server http://localhost:7070/request
```

Recall a "Used" Metered Feature

The client code can recall a "used" metered feature that was captured with a "request" operation and is associated with the **Undo Interval** attribute in the license model. To recall the feature, the code generates a capability request that specifies an "undo" operation by calling `CreateCapabilityRequest` with the `CapabilityRequestOperation.Undo` property. (The **Undo Interval** attribute, which typically sets a limited amount of time during which this operation can be used, is defined as part of the license model in FlexNet Operations.)

The "undo" operation also requires a correlation ID to indicate which capability request is being recalled (see Correlation ID for details). The code uses `options.CorrelationId = correlationId` to identify the specific correlation ID (as demonstrated by specifying an ID with the `-correlation` switch in the example). The response indicates whether the operation was accepted or rejected. The operation can be rejected, for example, if the correlation ID from the prior "request" operation is invalid, or the "undo interval" has elapsed.

Post-Usage-Capture: Managing Usage Data

After the client has captured usage data and sent it to the license server, the data is available for viewing and exporting to other entities, such the producer's billing system, from the Producer Portal in FlexNet Operations. (The data sent to the FlexNet Embedded local license server is not available on the Producer Portal until the license server has performed a synchronization to FlexNet Operations. Data sent to the CLS license server is almost instantly synchronized to FlexNet Operations and is soon after made available on the Producer Portal.)

See the *FlexNet Operations User Guide*, in particular section "Getting Started with Usage Management", for details about viewing and exporting usage data in FlexNet Operations. For information about FlexNet Embedded local license server's synchronization process, see the *FlexNet Embedded License Server Producer Guide*.

Additional Metered License Attributes

The section "Getting Started with Usage Management" in the *FlexNet Operations User Guide* describes the license attributes used with metered licenses. For example, the license model definition has the **Is this a Metered Model?** attribute set to **Yes** for usage-capture scenarios.

Additional attributes of a license model include:

- **Is this a Counted Model?:** Whether to count the number of licenses available to customers. For usage-capture scenarios, this value should be set to **Yes**. The "counted model" establishes the baseline count used as the cap for capped usage and by the `Overdraft` property to determine overage. The baseline count is calculated as follows:

  feature_count x *product_count* x *entitlement_count*

- **Overdraft:** Whether to grant additional licenses beyond the entitled amount, possibly to be charged at a different rate. The value can be **`Unlimited`**, a fixed number, or a percentage of the baseline count. The **`Unused`** value indicates that no overdraft is used.
- **Undo Interval (Seconds):** Amount of time, starting when the capability response is generated, that the user can perform an "undo" operation to recall erroneous or canceled usage data previously sent in a "request" operation.

An additional attribute that can be set at the model level or for a given metered feature (capability) is **Reusable**. If set to **`Yes`**, the feature can be returned after being acquired from trusted storage, and can also be "returned" to the license server by adding a desired feature with a negative count. (The time between acquiring the feature and returning it counts as usage.) If set to **`No`**, the feature cannot be returned after a local acquisition; the license must instead be released with `licensing.LicenseManager.ReturnAllLicenses`.

The FlexNet Embedded functionality provides APIs for reading attributes related to metered licenses and features. For example, the `isMetered` and `isMeteredReusable` properties in the `ILicense` interface respectively detect if an acquired license is metered or has the reusable attribute. (Similar `IFeature` properties indicate if a feature in a feature collection uses those attributes.) Related properties can give information about a license's or feature's "undo interval". The **View** example illustrates how to use these properties to determine attributes related to metered features. For more information, consult the API reference.

Note • *When using metered features, the license model's borrow interval attribute is not taken into account.*

Examining License Rights in a License Source

The following functionality can be used as license-enabled diagnostic code running on the client system, as well as a license-enabled diagnostic tool used in the development process. The purpose of this logic is to examine license rights existing on the client system and provide information regarding their availability for acquisition.

The difference between this type of license source and the non-diagnostic version is that a diagnostic source loads *all* features, even those that fail validation. While the diagnostic functionality provides additional information, it can produce overhead not acceptable in a production environment. For example, it will load all expired features into the license source, which consumes memory resources and slows the search for valid licenses at the time of the acquisition request.

Running the **View** example with the `-h` or `-help` switch displays usage information.

The source code for the implementation described in this walkthrough can be found in the directory *`install_dir`*`\examples\client_samples\View`, in the source file `View.cs`.

Assuming you have already created your core objects as described in Creating Core Licensing Objects, perform the following steps:

- Step 1: Create and Populate a Diagnostic License Source
- Step 2: Examine Features in the Feature Collection

Step 1: Create and Populate a Diagnostic License Source

This step is similar to code in the Step 1: Create and Populate the License Sources step in Using the License on the Client, except for the use of the diagnostic version of the `GetFeatureCollection` method, which you indicate by passing `true` to the *`diagnostic`* argument. FlexNet Embedded supports the diagnostic functionality for buffer, trusted-storage, trial, and certificate license sources.

```
IFeatureCollection bufferFeatureCollection = null;
if (licenseData != null)
{
    MessageTypeEnum licenseDataType = licensing.LicenseManager.MessageType(licenseData);
    if (licenseDataType != MessageTypeEnum.FLX_MESSAGE_TYPE_BUFFER_LICENSE &&
        licenseDataType != MessageTypeEnum.FLX_MESSAGE_TYPE_BUFFER_CAPABILITY_RESPONSE)
    {
        Console.WriteLine(invalidFile);
    }
    bufferFeatureCollection = licensing.LicenseManager.GetFeatureCollection(licenseData, true);
}
IFeatureCollection trustedStoreFeatureCollection =
   licensing.LicenseManager.GetFeatureCollection(LicenseSourceOption.TrustedStorage, true);
IFeatureCollection trialsFeatureCollection =
   licensing.LicenseManager.GetFeatureCollection(LicenseSourceOption.Trials, true);

// Print buffer feature information
StringBuilder builder = new StringBuilder();
builder.AppendLine(string.Empty);
builder.AppendLine("============================================");
if (bufferFeatureCollection != null)
{
    builder.AppendLine(string.Format("Features found in {0}:", inputFile));
    builder.AppendLine(string.Empty);
    GetFeatures(builder, bufferFeatureCollection);
    builder.AppendLine("============================================");
    builder.AppendLine(string.Empty);
    Util.DisplayInfoMessage(builder.ToString());
    builder = new StringBuilder();
    builder.AppendLine(string.Empty);
    builder.AppendLine("============================================");
}

// Print trusted storage feature information
builder.AppendLine("Features found in trusted storage:");
builder.AppendLine(string.Empty);
GetFeatures(builder, trustedStoreFeatureCollection);
builder.AppendLine("============================================");

// Print trial storage feature information
builder.AppendLine("Features found in trials storage:");
builder.AppendLine(string.Empty);
GetFeatures(builder, trialsFeatureCollection);
builder.AppendLine("============================================");
```

Step 2: Examine Features in the Feature Collection

This step defines helper methods that loop over the features in a collection and displays the attributes of each feature.

```
private static void GetFeatures(StringBuilder builder, IFeatureCollection featureCollection)
{
    int index = 0;
    // List all of the features for a given feature collection
    foreach (IFeature feature in featureCollection)
    {
        GetFeatureInfo(builder, index++, feature);
    }
}

private static void GetFeatureInfo(StringBuilder builder, int index, IFeature feature)
{
    FeatureAttributes featureAttributes = new FeatureAttributes();

    featureAttributes.Index = index;
    featureAttributes.Name = feature.Name;
    featureAttributes.Version = feature.Version;
    featureAttributes.StartDate = feature.StartDate;
    featureAttributes.ExpirationDate = feature.Expiration;
    featureAttributes.IsPerpetual = feature.IsPerpetual;
    featureAttributes.IsUncounted = feature.IsUncounted;
    featureAttributes.Count = feature.Count;
    featureAttributes.IsMetered = feature.IsMetered;
    featureAttributes.IsMeteredReusable = feature.IsMeteredReusable;
    featureAttributes.VendorString = feature.VendorString;
    featureAttributes.Issuer = feature.Issuer;
    featureAttributes.IssuedDate = feature.Issued;
    featureAttributes.Notice = feature.Notice;
    featureAttributes.SerialNumber = feature.SerialNumber;
    featureAttributes.AcquireStatus = feature.ValidStatusForAcquisition();
    featureAttributes.ServeStatus = feature.ValidStatusForServing();
    featureAttributes.IsMetered = feature.IsMetered;
    featureAttributes.IsMeteredReusable = feature.IsMeteredReusable;
    featureAttributes.MeteredUndoInterval = feature.MeteredUndoInterval;
    featureAttributes.MeteredAvailableCount = featureAttributes.IsMetered ?
        feature.AvailableAcquisitionCount : 0;
    featureAttributes.HostIds = feature.HostIds;

    // Check clock windback
    if (licensing.LicenseManager.ClockWindbackDetected)
    {
        throw new ApplicationException("Clock windback detected. Restore the clock to run the
            application.");
    }
    PrintFeatureAttributes(builder, ref featureAttributes);
}

private static void PrintFeatureAttributes(StringBuilder builder,
        ref FeatureAttributes featureAttributes)
{
    builder.Append(featureAttributes.Index + 1);
    builder.Append(string.Format(": {0} ", featureAttributes.Name));
```

```
builder.Append(featureAttributes.Version);
if (featureAttributes.ExpirationDate.HasValue)
{
    if (featureAttributes.IsPerpetual)
   {
        builder.Append(" permanent");
    }
    else
    {
        builder.Append(" ");
        builder.Append(featureAttributes.ExpirationDate.Value.ToString("dd-MMM-yyyy"));
    }
}
builder.Append(" ");
builder.Append(featureAttributes.IsUncounted ? "uncounted" :
     featureAttributes.Count.ToString());
if (featureAttributes.IsMetered)
{
    builder.Append(string.Format(" MODEL=metered{0}",
        featureAttributes.IsMeteredReusable ? " REUSABLE" : string.Empty));
    if (featureAttributes.MeteredUndoInterval.HasValue
        && featureAttributes.MeteredUndoInterval.Value.Ticks > 0)
    {
        builder.Append(string.Format(" UNDO_INTERVAL={0}",
            featureAttributes.MeteredUndoInterval.Value.TotalSeconds));
    }
}
if (!string.IsNullOrEmpty(featureAttributes.VendorString))
{
    builder.Append(string.Format(" VENDOR_STRING={0}", featureAttributes.VendorString));
}
if (!string.IsNullOrEmpty(featureAttributes.Issuer))
{
    builder.Append(string.Format(" ISSUER={0}", featureAttributes.Issuer));
}
if (featureAttributes.IssuedDate.HasValue)
{
    builder.Append(string.Format(" ISSUED={0}",
        featureAttributes.IssuedDate.Value.ToString("dd-MMM-yyyy")));
}
if (!string.IsNullOrEmpty(featureAttributes.Notice))
{
    builder.Append(string.Format(" NOTICE={0}", featureAttributes.Notice));
}
if (!string.IsNullOrEmpty(featureAttributes.SerialNumber))
{
    builder.Append(string.Format(" SN={0}", featureAttributes.SerialNumber));
}
if (featureAttributes.StartDate.HasValue)
{
    builder.Append(string.Format(" START={0}",
        featureAttributes.StartDate.Value.ToString("dd-MMM-yyyy")));
}

// Hostid(s)
if (featureAttributes.HostIds.Keys.Count > 0)
```

```
        {
            string hostIdString = string.Empty;
            string decorateLeft = featureAttributes.HostIds.Count > 1 ? " Hostids=[" : " Hostid=";
            string decorateRight = featureAttributes.HostIds.Count > 1 ? "]" : string.Empty;
            foreach (KeyValuePair<HostIdEnum, List<string>> HostId in featureAttributes.HostIds)
            {
                string lparen = HostId.Value.Count > 1 ? "(" : string.Empty;
                string rparen = HostId.Value.Count > 1 ? ") " : " ";
                if (HostId.Key != HostIdEnum.FLX_HOSTID_TYPE_ANY)
                {
                    hostIdString += HostId.Key.ToString().Remove(0, 16) + "=";
                }
                string value = "";
                foreach (string str in HostId.Value)
                {
                    value += (string.IsNullOrEmpty(value) ? string.Empty : ",") + str;
                }
                hostIdString += lparen + value + rparen;
            }
            builder.Append(decorateLeft + hostIdString.TrimEnd() + decorateRight);
        }
        builder.AppendLine(string.Empty);
        if (featureAttributes.AcquireStatus == 0)
        {
            if (featureAttributes.IsMetered && featureAttributes.MeteredAvailableCount == 0)
            {
                builder.AppendLine("     Entire count consumed");
            }
            else if (featureAttributes.IsMetered && featureAttributes.MeteredAvailableCount > 0)
            {
                builder.AppendLine(string.Format("     Available for acquisition: {0}",
                    featureAttributes.MeteredAvailableCount));
            }
            else
            {
                builder.AppendLine("     Valid for acquisition");
            }
        }
        else
        {
            builder.AppendLine(string.Format("     Not valid for acquisition: {0}",
                licensing.GetErrorDescription(featureAttributes.AcquireStatus)));
        }
    }
```

The **View** example project accepts the name of a binary license file as a command-line argument, and prints the name, version, expiration, and any optional keyword values for each feature in the license. If a feature is invalid, the executable displays the reason: the start date is in the future, the feature has expired, the feature was issued by a different producer, the entire license count has been consumed, and so forth.

In addition, the **View** example displays a feature's hostids by calling `getHostIds` and displays any vendor-dictionary items in trusted storage by calling `getVendorString`. Moreover, if a feature uses a metered license model, the **View** example displays attributes related to metering. (For more information about metering, see Capturing Feature Usage on the Client.) Note that the **View** example also enables clock-windback detection.

Advanced Topic: FlexNet Publisher Certificate Support

To assist producers who are beginning a transition from FlexNet Publisher to FlexNet Embedded, FlexNet Embedded functionality provides partial support for a FlexNet Publisher certificate as a license source. In particular, FlexNet Embedded supports unserved, uncounted certificates, with a restricted set of keywords and hostid types.

The following sections describe this support for certificate licensing:

- Preparing Your Identity Data for Certificate Support
- Using the Lmflex Example
- Differences in Certificate Licensing Behavior

Preparing Your Identity Data for Certificate Support

Your client identity data must be specially prepared to include some FlexNet Publisher information in order to validate FlexNet Publisher certificate signatures.

1. First, obtain the file `lmpubkey.h` from your FlexNet Publisher toolkit. This file is generated by running the command **`lmnewgen -pubkey`**. (If your FlexNet Publisher certificates use the older license-key signature type, obtain the `lmseeds.h` file from your FlexNet Publisher toolkit.)
2. Next, create or process your identity data using the FlexNet Embedded Client toolkit utility `pubidutil`, passing appropriate values for the certificate-related settings. The certificate-related switches are:

   - `-certificate` *`certificate-file`*: Enables FlexNet Publisher certificate support by including the public keys specified in your `lmpubkey.h` (or `lmseeds.h`) file in your identity data.
   - `-certificateSigType` *`sig-type`*: Specifies your FlexNet Publisher signature type, one of "`sign`", "`sign2`", or "`lk`".
   - `-certificateSigStrength` *`strength`*: Specifies your FlexNet Publisher signature strength. The value is 0 for LK signatures; for TRL signatures, the value is 0 for the lowest (113-bit) strength, 1 for medium (163-bit) strength, and 2 for the highest (239-bit) strength.

   For example, if you have already generated your unprocessed `IdentityClient.bin` file, the following command will process it:

   `pubidutil -console -certificate lmpubkey.h -certificateSigType sign -certificateSigStrength 1`

You can then use `printbin` with the `-cs` switch to generate the `IdentityClient.cs` file that you will compile into your FlexNet Embedded code. (An `IdentityClient.cs` header file based on an `IdentityClient.bin` file that has been correctly prepared will contain a `CertificatePublicKey` entry in the initial comment.)

If you have not previously created your binary identity files, you can additionally supply the `-identityName`, `-name`, `-keys`, and other switches to `pubidutil`.

Using the Lmflex Example

To illustrate the use of a FlexNet Publisher certificate as a license source, the FlexNet Embedded Client .NET XT or .NET Core XT toolkit includes the **Lmflex** example. This example accepts the path to a signed FlexNet Publisher certificate file, and attempts to acquire features called `f1` and `f2`. The FlexNet Publisher certificate is expected to have been signed using `lmcrypt` or back-office server such as FlexNet Operations. A typical FlexNet Publisher certificate is a text file with contents similar to the following:

```
INCREMENT f1 demo 1.0 1-jan-2025 uncounted HOSTID=USER=SampleUser SIGN="..."
INCREMENT f2 demo 1.0 1-jan-2025 uncounted HOSTID=USER=SampleUser SIGN="..."
```

Open and build the Visual Studio project file `examples\client_samples\Lmflex\Lmflex.csproj`. For implementation details, refer to the source code in the *`install_dir`*`\examples\client_samples\Lmflex` directory, in the source file `Lmflex.cs`. The identity data (`IdentityClient.cs`) compiled into the example must have been prepared to include the FlexNet Publisher public keys, as described in the previous section.

Running the `Lmflex` executable with the `-help` switch displays usage information.

Assuming you have already created your producer and identity objects as described in Creating Core Licensing Objects, the implementation is similar to other examples described in this chapter. In particular, the implementation must:

- Create the Certificate License Source
- Acquire Features from the Certificate License Source

Create the Certificate License Source

The `Lmflex` implementation reads the signed FlexNet Publisher certificate specified as a command-line argument, and creates the certificate license source with that file using `AddCertificateLicenseSource`.

```
// add legacy certificate license source
licensing.LicenseManager.AddCertificateLicenseSource(legacyCertificateFile, "CertificateFile");
```

If the certificate contains an unsupported or unknown keyword, `AddCertificateLicenseSource` throws an exception, skipping lines in the certificate that contains the keyword. See Differences in Certificate Licensing Behavior for information about FlexNet Embedded support for license certificates.

Acquire Features from the Certificate License Source

You acquire a feature from a certificate license source the same as with any other type of license source, using the `licensing.LicenseManager.Acquire` method.

To run the example, supply a signed FlexNet Publisher certificate with a command similar to the following:

```
Lmflex legacy.lic
```

If license acquisition is successful, the `Lmflex` output should appear similar to this:

```
INFO: License acquisition for feature f1 version 1.0 successful.
INFO: License for feature f1 version 1.0 successfully returned.
INFO: License acquisition for feature f2 version 1.0 successful.
INFO: License for feature f2 version 1.0 successfully returned.
```

Exceptions thrown by the `Acquire` method are the same as those thrown when acquiring licenses from other types of license sources.

Differences in Certificate Licensing Behavior

As described in Feature Definitions, FlexNet Embedded functionality supports a subset of the feature keywords supported by FlexNet Publisher. Any feature definition supporting an unknown or unsupported keyword will be skipped. This section describes additional differences between FlexNet Embedded and FlexNet Publisher with respect to license certificates.

Only unserved, uncounted certificates are supported. FlexNet Publisher certificates containing a `SERVER` line or `VENDOR` line are not supported.

FlexNet Embedded treats feature names as case sensitive, whereas most hostid values are not case sensitive. For more information, see Hostids. In your FlexNet Embedded code, verify that the feature name passed to the `Acquire` method uses the same capitalization as used in the certificate. (When preparing identity data to include certificate support, `pubidutil` supports a `-certificateCaseSensitive` switch, but this applies only to feature signatures, and not to feature names or hostid values.)

Composite hostids and vendor-defined hostids are not supported.

FlexNet Embedded version comparisons are performed field by field, unlike FlexNet Publisher which treats the version number as a single real number. Thus FlexNet Embedded treats version 1.1 as less than 1.10, while FlexNet Publisher treats them as equal. (When generating FlexNet Embedded buffer licenses or capability responses with the testing tools `licensefileutil` and `capresponseutil`, you can force the FlexNet Publisher behavior by passing the `-legacyFeatureVersioning` switch.)

As an optimization, FlexNet Embedded normally does not load certain invalid features into a license source when the license source is created. For example, by default an expired feature will not be added to a license source, and therefore an attempt to acquire the expired feature will return a “feature not found” error code instead of a “feature has expired” error code. To modify the behavior so that expired and other invalid features are included, create a diagnostic license source, as illustrated in the **View** example.

FlexNet Embedded treats `FEATURE` lines and `INCREMENT` lines identically, as opposed to ignoring all but one `FEATURE` line.

Advanced Topic: Multiple-Source Regenerative Licensing

Traditionally, trusted storage for a FlexNet Embedded client is provisioned with licenses from a single source. The source can be either a back office, as described in Licenses Obtained from the Back-Office Server, or a single license server—either a FlexNet Embedded local license server, which is a local license server, or a CLS (Cloud Licensing Service) license server—as described in Licenses Obtained from a License Server. Based on the regenerative nature of trusted storage, each time a new capability response from the server is processed, the current licenses in trusted storage are replaced with those sent in the response.

However, due to evolving requirements in customer enterprises, trusted storage has been enhanced with the capability to store licenses from multiple servers—FlexNet Embedded local license servers, CLS license servers, and the back office (FlexNet Operations only)—in separate locations within trusted storage. When the client receives a capability response from a one of the server sources, it stores the response in the source’s dedicated location in trusted storage, regenerating licenses in that location only (and refreshing the server’s in-memory license source).

The following sections describe more about multiple-source regenerative licensing and how to support it in the client code:

- Use Cases for Multiple-Source Regenerative Licensing

- Providing Support for Multiple-Source Regenerative Licensing in the Client Code
- Considerations

Use Cases for Multiple-Source Regenerative Licensing

The ability of the client to obtain licenses from multiple servers—each with its dedicated location in trusted storage for storing and regenerating licenses—provides options for managing how a client is provisioned with licenses.

For example, when multiple-source regenerative licensing is used, an enterprise client can borrow additional licenses from a second license server (in another department, for example), while maintaining its current set of licenses obtained from the main license server. In another scenario, enterprise clients can be provisioned with node-locked licenses from the back office for basic product functionality and then with enterprise-shared licenses from one or more license servers for high-value product functionality. Multiple-source regenerative licensing might also help a service engineer who, in attempting to repair a customer's machine, can install temporary licenses for trouble-shooting purposes without wiping out the customer's purchased licenses.

Providing Support for Multiple-Source Regenerative Licensing in the Client Code

In general, to implement support for multiple-source regenerative licensing in your client code requires planning. For example, you need to determine how many (and which) servers to use as sources for provisioning clients in the enterprise and how licenses are to be distributed across the different servers. Additionally, you need to decide which server instance ID to assign each given source in order to create its license source and dedicated location in trusted storage.

Then, to create your FlexNet Embedded client code, use the information in Licenses Obtained from the Back-Office Server and Licenses Obtained from a License Server as your guide; but refer to the following sections for specific information about incorporating the functionality needed to support multiple-source regenerative licensing:

- Creating the License Source for a Server Instance
- Identifying the Server Instance in the Capability Request
- Processing the Response from a Server Instance

The example code implementations shown in the following sections are constructed using the context of the **CapabilityRequest** example.

Creating the License Source for a Server Instance

To set up trusted storage to store licenses from multiple servers, you need a license source for each server, identified by the specific *server instance ID* you assign the given server. This same ID in turn identifies the specific trusted-storage location where licenses for this server will be stored.

Processing the capability response for a given server instance ID will automatically create the corresponding trusted-storage license source and add it to the license source collection, if the source is not already present (see Processing the Response from a Server Instance).

However, if your implementation requires that a license source be created and added to the collection explicitly, use the following method, providing the server instance ID for which the license source is being created. (No two license sources can have the same server instance ID.)

```
licensing.LicenseManager.AddTrustedStorageLicenseSource(LicenseServerInstance.serverInstanceID)
```

Optionally, you can omit the server instance ID in this method to create a "default" license source—that is, a source created for the traditional trusted-storage location not specified by a server instance ID—as one of the multiple sources. However, when you do not specify a server instance ID to create for one of the license sources, the remaining trusted-storage license sources must each be created with the instance ID specified.

The following example implementation adds two trusted-storage license sources to the license source collection—a default license source and a license source for Server5 (server instance 5):

```
// Add trusted storage license sources
// AddTrustedStorageLicenseSource() is equivalent to
// AddTrustedStorageLicenseSource(LicenseServerInstance.Default)
licensing.LicenseManager.AddTrustedStorageLicenseSource();
licensing.LicenseManager.AddTrustedStorageLicenseSource
          (LicenseServerInstance.Server5);
```

Identifying the Server Instance in the Capability Request

Each server in this multiple-source model is assigned a specific server instance ID to identify the trusted-storage location in which licenses from this server are to be stored (as described in the previous section Creating the License Source for a Server Instance). Your implementation must ensure that, when the capability response from a specific server is processed, the licenses contained in the response are stored in the correct location for that server.

One method that FlexNet Embedded uses to help you verify the proper storage of licenses is to compare the server instance ID in the capability response with the instance ID used to process the response. If the IDs do not match, an error is raised. (For more information about processing the capability response for a specific server instance ID, see the next section Processing the Response from a Server Instance)

To enable FlexNet Embedded to perform this check, you must initially include the instance ID in the capability request so that it can be returned in the capability response. However, the back office and pre-2016 license servers are not capable of returning the instance ID in the response. A best practice might then be always to include the instance ID in the request so that FlexNet Embedded performs a comparison check whenever the ID is available in the response.

To specify the instance ID for the target server in the capability request, create the ICapabilityRequestOptions object using the factory method overload, which takes a server instance ID value:

```
licensing.LicenseManager.CreateCapabilityRequestOptions(LicenseServerInstance.serverInstanceID)
```

The following shows a sample implementation of this functionality (in this case, to specify server instance 5):

```
// create a capability request options object for the intended server
// instance value
ICapabilityRequestOptions options =
     licensing.LicenseManager.CreateCapabilityRequestOptions(LicenseServerInstance.Server5);
```

Processing the Response from a Server Instance

Use the following method to process the capability response from a given server—identified by the server instance ID specified in the method—into the trusted-storage location and license source with that same ID:

```
licensing.LicenseManager.ProcessCapabilityResponse(binCapResponse,
          LicenseServerInstance.serverInstanceID)
```

If the license source for the server does not exist, this process additionally creates the source and adds it to the license-source collection.

The following shows a sample implementation of this functionality (which processes the capability response into the trusted-storage location and license source for server instance 5):

```
// Process the binary capability response into the trusted storage license source
// correstponding to server instance 5.
ICapabilityResponse response =
     licensing.LicenseManager.ProcessCapabilityResponse(binCapResponse,
         LicenseServerInstance.Server5);
```

Validations

To help ensure that the licenses in the capability response are installed in the correct location in trusted storage, FlexNet Embedded generates an error when certain conditions exist, such as:

- The server hostid included in the capability response already exists for another server-instance location.
- The server instance ID, when available in the capability response, does not match the one specified for `licensing.LicenseManager.ProcessCapabilityResponse`.

Considerations

Consider the following when you implement support for multiple-source regenerative licensing in your FlexNet Embedded client code:

- **Maximum license sources**—To determine the maximum number of license sources allowed, refer to the API reference for the current enumerator values available for use as server instance IDs.
- **Metered licenses**—Metered licenses are supported in multiple-source regenerative licensing.
- **Back office as a source**—Multiple-source regenerative licensing allows only one source that is a back office. Additionally, the only back office supported as a source is FlexNet Operations.
- **License acquisition**—When a client attempts to acquire licenses, FlexNet Embedded aggregates license counts across *all* license sources in the client's license-source collection—in the order in which the sources were added to the collection. License sources in a collection can include the one or more trusted-storage sources *and* any non-trusted sources, such as those for trial and buffer licenses. (The licenses are aggregated according to FlexNet Embedded internal rules.)
- **Upgrade from a pre-2016 version**—If you have upgraded your FlexNet Embedded .NET XT toolkit from a version previous to the 2016 release and have rebuilt your client product with this toolkit, the legacy APIs continue to support the client's traditional single-source regenerative licensing. However, with the 2016 or later toolkit, you have the option to add support for multiple-source regenerative licensing in your existing code, should you decide to do so.

7

Using the Updates and Insights APIs

The Updates and Insights component included in the .NET XT SDK comprises a set of .NET functions that you can integrate in product code to retrieve notifications—product messages or updates—from the Updates and Insights notification server (located in Revenera-hosted FlexNet Operations). For product-update notifications, you can use Updates and Insights functionality to download and install the updates.

Note • *Currently, FlexNet Operations supports only product-update notifications and leverages the "manifest file" update-type format to provide information about the file or files involved in a given update (see About the Manifest File for An Update Notification).*

The following sections describe the general flow of Updates and Insights API calls used when implementing a scenario that retrieves all update notifications for a given product package, downloads the file (currently just a manifest file) required for installing a given update, and then installs the update. The sections walk you through the functionality used in this scenario, referring to the source code for the example **Notification** project. (The source code is found in *`install_dir`*`\examples\uai_client_samples`.)

This chapter includes the following implementation descriptions:

- Common Preparation Steps
- About the Manifest File for An Update Notification
- Creating Core Notification Objects and Registering the Client
- Obtaining Notifications and Downloading and Installing the Updates
- One-time Event: Client Device Registration with FlexNet Operations
- About Client Communications for the Updates and Insights

The information in this chapter applies only to the FlexNet Embedded Client .NET XT toolkit, which supports both FlexNet Embedded and Updates and Insights functionality. (The FlexNet Embedded Client .NET Core XT toolkit supports FlexNet Embedded functionality only.)

Common Preparation Steps

The following steps need to performed to ensure that the Updates and Insights client code has access to the information it needs to perform notification operations successfully:

- Obtaining Your Producer Identity Files
- Adding a Product and Its Update to Your Producer Site

Obtaining Your Producer Identity Files

The **Notification** example assumes that you have your producer identity files in place. That is, either you have generated these files in FlexNet Operations and have downloaded the `IdentityClient.h` file (or comparable file) to the *`install_dir`*`\examples\identity` directory (or a directory accessible by the **Notification** project); or you have used toolkit utilities to perform the following tasks to create and distribute your producer identity data:

- Run the `pubidutil` utility to create the producer identity binary files for FlexNet Operations and the Updates and Insights client (by default called `IdentityBackOffice.bin` and `IdentityClient.bin`).
- Uploaded the back-office identity binary file to Revenera-hosted FlexNet Operations.
- Run the `printbin` utility on the client-identity binary file to generate a header file (for example, `IdentityClient.h`) containing compiler-readable (C byte array) identity information and have copied this file to *`install_dir`*`/examples/identity` (or a directory accessible by the **Notification** project).

The process of using `pubidutil` to generate identity information and then distributing this information is described in the section Creating the Producer Identity in the *Quick Start* chapter.

Adding a Product and Its Update to Your Producer Site

Before you can successfully run the **Notification** example, a sample product package must be defined and its update published in FlexNet Operations for access by the notification server. For instructions, refer to the *FlexNet Operations User Guide* that is available in the Producer Portal.

Tip • *For best results, follow the exercises in the sections "Getting Started with Entitlement Management" and "Getting Started with Updates and Insights" in the FlexNet Operations User Guide. There you can create a product, an entitlement, download packages, and an update that you can use to test the Updates and Insights API implementations.*

Then, before executing the **Notification** example, you can access the Package Products and Updates pages in FlexNet Operations to obtain the following information for the product package and update:

- The package ID for the product for which you will request update notifications. This information is required for running the example.
- The Microsoft Language Locale Identifier (LCID) for the language of the product package if the default value (1033) used by **Notification** example is not applicable. See Running "Notification" in the *Quick Start* chapter for details.

About the Manifest File for An Update Notification

Currently, the notification server generates a manifest file as the only content type for a product update. A manifest file contains a list of the one or more files (along with other relevant information such as file locations) that, in turn, need to be downloaded and possibly executed to install the update successfully. For an explanation of an example a manifest file, see the appendix Manifest File Contents for a Product Update.

Creating Core Notification Objects and Registering the Client

Before your Updates and Insights client code can retrieve notifications, it must create the core objects required for notification operations. Additionally, the client code must register the client device with FlexNet Operations if the device has not been previously registered. The following sections look at sample code implementations that accomplish these preliminary tasks:

- Setting Up the Updates and Insights Client Object
- Registering the Client Device with FlexNet Operations
- Setting Up the Product Package Object

Setting Up the Updates and Insights Client Object

Notification-enabled code should first create the core Updates and Insights client object, also called the *IUAIClient object*. This object maintains a reference to each product package for which notifications are retrieved and is used to register the client device with the back office. This object must be initialized with your producer client identity, as created in the previous section, Common Preparation Steps.

The following excerpt from `Notification.cs` shows a sample code implementation of the `IUAIClient` object creation:

Table 7-1 • Excerpt from `Notification.cs`: Creating the IUAIClient Object

```
using (uaiClient = UAIClientFactory.GetUAIClient(
        IdentityClient.IdentityData, null, null))...
 ...
```

Note • *The Notification examples wraps most Updates and Insights operations in a "try-catch" block. These blocks have been omitted from many of the code excerpts presented in this chapter to focus on the pertinent code.*

The UAI client object is initialized by calling `UAIClientFactory.GetUAIClient`. The following describes the arguments used in this method.

- The first argument to `GetUAIClient` is your client-identity information used for verifying notifications from the notification server. The implementation uses the `IdentityClient.IdentityData` expression to pass the binary client identity (found in `IdentityClient.cs`) as the argument to `GetUAIClient`. The `IdentityClient.cs` file was created by running the `printbin` utility against the `IdentityClient.bin` file (created in FlexNet Operations or by using `pubidutil`). Note that the client identity data contains the public key information used to authenticate the notification messages digitally signed by the notification server.

  Note • *For security reasons, it is strongly recommended that your producer client identity be stored as a buffer in the updates-enabled code, and not as an external file. The "printbin" toolkit utility can convert a binary producer identity file (on a development system) into a format that can be used in Updates and Insights code.*

- The second argument is currently ignored (and passed as null).
- The third argument is used to specify a custom hostid, should the user want to use one, to identify the Updates and Insights client device to the notification server. However, if the client code also uses the `IUAClient.SetHostid (type, value)` method to set the default hostid for the client, the default hostid takes precedent over the custom hostid specified for this argument.

  Note that a request for notifications or client registration (see the next section) sent to the notification server always contains a hostid, chosen by Updates and Insights functionality in this order:

  - The default hostid set for the implementation using `IUAClient.SetHostid (type, value)`
  - The custom string hostid
  - The first ethernet hostid detected on the client

Registering the Client Device with FlexNet Operations

If the hostid for the Updates and Insights client device has not been previously registered with Revenera-hosted FlexNet Operations, either through Updates and Insights or FlexNet Embedded client functionality, the client code must perform this process before it can obtain notifications. See One-time Event: Client Device Registration with FlexNet Operations for details.

Before you can register the client device or determine whether it has been previously registered, the `IUAIClient` object must already exist. See Setting Up the Updates and Insights Client Object.

Setting Up the Product Package Object

The second core object required to retrieve notifications is the product package object. Your code must create a separate object for each product package for which the client will retrieve notifications. This object identifies the specific product package to the `IUAIClient` object. Although a product package object is associated with a single `IUAIClient` object, a `IUAIClient` object can have multiple product-package objects associated with it.

The following excerpt from `Notification.cs` shows a sample implementation that creates a product-package object:

Table 7-2 • Excerpt from `Notification.cs`: Creating the Product Package Object

```
using (uaiClient = UAIClientFactory.GetUAIClient(
                        IdentityClient.IdentityData, null, null))
            {

                if (registerClient)
                {
                    RegisterClient();
                }

                using (productPackage = uaiClient.GetProductPackage(myProductPackageId, myLanguage,
                            myPlatform))
```

The `GetProductPackage` method in the `IUAIClient` interface either locates an existing product package or creates a new product package object and associates it to the `IUAIClient` object through the product's package ID.

The `myProductPackageId` parameter identifies the download ID for the product package currently installed in the local environment. You can find this download ID on the Package Products page and the Updates page in FlexNet Operations. The `myLanguage` and `myPlatform` parameters define additional product attributes required for the object to determine the appropriate notifications to retrieve for the current product package. However, if these parameters are omitted, defaults are used, as described for the **Notification** example in Running "Notification" in the *Quick Start* chapter.)

Obtaining Notifications and Downloading and Installing the Updates

The following scenario describes how you might use the Updates and Insights .NET XT functionality in your client code to retrieve available product notifications and then download and execute those notifications that are updates. The scenario is based on the **Notification** example and refers to the source code in the file `Notification.cs`, located in the `examples\uai_client_samples\Notification` directory.

Additionally, running the `Notification.exe` executable with the `-h` or `-help` switch displays usage information for the example.

This walkthrough assumes that your source code has already set up the core objects and registered the client with the back office (if it was not already registered), as described Creating Core Notification Objects and Registering the Client. The walkthrough is discussed in these phases:

- Obtaining Notifications
- Downloading and Installing an Update

Obtaining Notifications

The sample code retrieves the notifications for a product (that is, a specific product package ID) from the notification server by sending a notification request for a notification collection. The notification server, in turn, responds with the notification collection for the product. Only one notification collection can exist for a product package ID at any one time. The following sections describe the process of obtaining notifications:

- Step 1: Send the Notification Request
- Step 2: Inspect the Collection

Step 1: Send the Notification Request

Communications between the Updates and Insights client and the notification server are asynchronous and handled internally by FlexNet Embedded client functionality (see About Client Communications for the Updates and Insights). This excerpt from the example calls the following methods (from the `IProductPackage` interface) required to retrieve a notification collection from the server:

- The `RequestNotifications` method that sends the request to the notification server (referenced by `nsServerURL`) to obtain all available notifications for the product package.
- The `NotificationsReady` method that polls the notification server (at an implementation-defined interval) until the client receives a response message from the server containing the notification collection data or until an error occurs. Once the client receives the response message, the `NotificationsReady` method processes the response so that notification collection can be accessed. Additionally, a status message is internally generated and sent to the notification server, indicating that the notification collection has been delivered to the Updates and Insights client.

Note that the example code leverages the .NET `BackgroundWorker` class to spawn a new thread for the asynchronous communications with the notification server to retrieve notifications. This implementation is for demonstration only. Should you want the client to spawn a separate communications thread for notification retrieval process, you can use any method appropriate for your application.

Table 7-3 • Excerpt from `Notification.cs`: Requesting Notifications

```
    {
        BackgroundWorker notificationsThread = sender as BackgroundWorker;
        bool notificationsReady = false;
        int ii = 0;
        e.Result = -1;
        try
        {
            productPackage.RequestNotifications(nsServerUrl);
            do
            {
                Thread.Sleep(1000);
                notificationsThread.ReportProgress(++ii < 100 ? ii : 100);
                notificationsReady = productPackage.NotificationsReady;
            } while (!notificationsReady);
            e.Result = 0;
```

Table 7-3 • Excerpt from `Notification.cs`: Requesting Notifications (cont.)

```
            }
            catch (Exception exc)
            {
                threadException = exc;
            }
        }
```

Step 2: Inspect the Collection

The following excerpt from `Notification.cs` shows sample code that inspects the notification collection and displays the properties for each notification item on the console. If the code determines that a specific item is an "update", it uses the `INotificationUpdate` interface to access that item. (The other interface `INotificatioMessage` is used to access "message"-type notifications, which are currently not supported by the notification server.)

Note that the code first determines the collection size. A notification collection size of 0 (zero) means that no notifications for this product were available.

Table 7-4 • Excerpt from `Notification.cs`: Parsing the Notification Collection

```
            List<INotification> notifications = productPackage.NotificationCollection;
            uint i = 1;
            int ii = notifications.Count;
            Util.DisplayMessage(String.Format("{0} notification items returned for requested product package
                              id.", ii == 0 ? "No" : ii.ToString()), ii == 0 ? "INFO" : null);
            foreach (INotification notification in notifications)
            {
                StringBuilder builder = new StringBuilder();
                builder.AppendLine("");
                builder.AppendLine(String.Format("Notification item {0} of {1} attributes:", i, ii));
                if (notification is INotificationUpdate)
                {
                    INotificationUpdate update = notification as INotificationUpdate;
                    builder.AppendLine("  Type:                          Update");
                    List<String> updateProperties = new List<String>
                        {update.ID, update.ProductPackageId, update.ToProductPackageId, update.Name,
                                update.Title,
                        update.ProductName, update.ContentType.ToString(), update.Description,
                                update.Details,
                        update.DetailsURL, update.DownloadType.ToString(), update.DownloadURL,
                                update.DownloadSize.ToString(),
                        update.AvailabilityDate, update.ExpirationDate, update.LanguageCode,
                                update.ElevationRequired.ToString(),
                        update.CommandLine, update.TargetDirectory, update.SecurityType,
                                update.SecuritySignature};
                    for (int j = 0; j < updateProperties.Count; j++)
                    {
                        if (!String.IsNullOrEmpty(updateProperties[j]))
                        {
                            builder.AppendFormat(updatePropertyDescriptions[j], updateProperties[j]);
                        }
                    }
                }
```

Downloading and Installing an Update

The code reviewed in this part of the scenario extracts those notification items that are of the "update" type, downloads the payload associated with a given item, and installs the update, as described in these sections:

- Download Payload and Install the Update
- (Optional) Use a Callback Function to Track the Update Progress

Note that, currently, the only notification type supported by the notification server is the "update" type, and the only content type that the notification server supports for a payload is a manifest file (see Manifest File Contents for a Product Update). When the payload is a manifest file, the manifest file and each item in the file must be downloaded before the installation begins, as described in more detail the next section.

Download Payload and Install the Update

The next excerpt shows a sample implementation for the `DownloadAndExecute` method (in the `INotificationUpdate` interface) used to download the "update" payload associated with a notification item and then install the update. For a "manifest file" payload, this method first downloads the manifest file and then, to install the update, downloads all files associated with the manifest items and finally executes all downloaded files marked for execution. Note the following about the `DownloadAndExecute` method:

- The `downloadLocation` and `downloadFilename` parameters in the `DownloadandExecute` method point to the target location to which to download the payload (in this case, a manifest file) on the client. Note that these parameters do not have to be specified. By default, the payload is downloaded to the user's `Downloads` folder; the default file name is determined by the URL or the HTTP headers.
- This method can point to an `IComm` object that your code creates to define custom settings for notification-item downloads (see About Client Communications for the Updates and Insights). In the excerpt, the method points to such an object called `dlComm`.
- The method can supply a producer-defined callback (in the excerpt, `MyStatusDelegate`), which allows the end-user to keep track of the progress of the download and execution process. This callback is discussed in more detail in the next section (Optional) Use a Callback Function to Track the Update Progress.

Finally, during the download and installation phases for an update, status messages are internally generated and sent to the notification server to indicate the start and completion of these phases. For a manifest file, the download phase starts with the download of the manifest file itself and ends when all files associated with the manifest items have been downloaded. The installation phase begins with the execution of the first downloaded file marked for execution and completes when all files marked for execution have been executed.

Table 7-5 • Excerpt from `Notification.cs`: Downloading and Installing an Update

```
private static void DownloadUpdates(IComm dlComm)
{
    string downloadLocation = null;
    string downloadFilename = null;
    List<INotification> notifications = productPackage.NotificationCollection;
    for (int i = 0; i < notifications.Count; i++)
```

Table 7-5 • Excerpt from `Notification.cs`: Downloading and Installing an Update (cont.)

```
        {
            if (notifications[i] is INotificationUpdate)
            {
                INotificationUpdate updateNotification = notifications[i] as INotificationUpdate;
                try
                {
                    updateNotification.UserData = new ProgessUserData();
                    Util.DisplayMessage(String.Format("Downloading notification #{0}", i + 1));
                    updateNotification.DownloadAndExecute(downloadLocation, downloadFilename,
                                MyStatusDelegate, dlComm);
                    if (updateNotification.DownloadPaused)
                    {
                        Util.DisplayMessage("The download was successfully paused. Automatically
                                  resuming...", "Download paused");
                        try
                        {
                           updateNotification.ResumeDownload(MyStatusDelegate, dlComm);
                        }
                        catch (Exception exc)
                        {
                            Util.DisplayErrorMessage(String.Format("{0}Unable to resume download or execute
                                notification item #{1}", Environment.NewLine, i + 1));
                            HandleException(exc);
                        }
                    }
                    if (!String.IsNullOrEmpty(updateNotification.DataLocation))
                    {
                        Util.DisplayMessage(String.Format("Download successful to: {0}",
                               updateNotification.DataLocation));
                    }
                }
                catch (Exception exc)
                {
                   Util.DisplayErrorMessage(String.Format("{0}Unable to download or execute notification item
                                #{1}", Environment.NewLine, i + 1));
                    HandleException(exc);
                }
                finally
                {
                    updateNotification.UserData = null;
                }
            }
        }
```

(Optional) Use a Callback Function to Track the Update Progress

The sample code takes advantage of the ability to supply a producer-defined download callback delegate method, called `MyStatusDelegate` in this excerpt. (This delegate method is of the type `UpdateStatusDelegate`, which returns a `NotificationUpdateStatus` enumerator.) The callback receives the information about the download and execution operations and their progress and, in this example, provides a visual display of the download progress. The callback also enables the opportunity to pause or cancel the download.

Note that the example uses a `ProgressUserData` object to carry implementation-contextual data to the delegate callback. However, your implementation can use any appropriate custom object to provide access to application-specific information.

The percentage of progress at certain points in the download process is calculated using the example `ReportPercent` method. The last lines in this code excerpt show how to set a pause at a certain percentage point in the download process (`userDefinedPauseAtPercentage = 0;`) and save what has been completed up to this point. Note that the returned status `NotificationUpdateStatus.PAUSE` will cause a current download to be paused. The `NotificationUpdateStatus.CONTINUE` instructs the download to continue processing downloaded data, while a returned status `NotificationUpdateStatus.CANCEL` will cause the current download or execution to be canceled without the option of resuming.

Note • *This user callback is an example only. Your own callback function can perform whatever actions you require.*

Table 7-6 • Excerpt from `Notification.cs`: Example Callback Function

```
private static NotificationUpdateStatus MyStatusDelegate(NotificationUpdateStage stage, long currentSize,
                  long totalSize, INotificationUpdate update, Exception exc)
{
    if (((ProgessUserData)update.UserData).lastUpdateStage != stage)
    {
        switch (stage)
        {
            case NotificationUpdateStage.DOWNLOAD_START:
                Util.DisplayMessage("Download started"); break;
            case NotificationUpdateStage.DOWNLOADING:
                if (totalSize > 0)
                {
                    return ReportPercent(currentSize, totalSize, update);
                }
                break;
            case NotificationUpdateStage.DOWNLOAD_END:
                Util.DisplayMessage("Download ended"); break;
            case NotificationUpdateStage.DOWNLOAD_FAIL:
                Util.DisplayMessage(((ProgessUserData)update.UserData).executionStarted ? "Execution
                                failed" : "Download failed"); break;
            case NotificationUpdateStage.EXECUTION_START:
                Util.DisplayMessage("Execution started");
                ((ProgessUserData)update.UserData).executionStarted = true;
                break;
            case NotificationUpdateStage.EXECUTION_END:
                Util.DisplayMessage("Execution ended"); break;
        }
        ((ProgessUserData)update.UserData).lastUpdateStage = stage;
    }
    return NotificationUpdateStatus.CONTINUE;
}
```

Table 7-6 • Excerpt from `Notification.cs`: Example Callback Function (cont.)

```
        private static NotificationUpdateStatus ReportPercent(long currentSize, long totalSize,
                              INotificationUpdate update)
        {
            uint completionPercent = (uint)((currentSize * 100)/totalSize);
            // Display notable progress
            if (completionPercent > ((ProgessUserData)update.UserData).lastPercentComplete)
            {
                string progress = "[";
                uint progressBarPos = (uint)((progressBarWidth * currentSize) / totalSize);
                for (uint i = 0; i < progressBarWidth; ++i)
                {
                    progress += (i < progressBarPos ? "=" : (i == progressBarPos ? ">" : " "));
                }
                progress = String.Format("{0}] {1} percent complete{2}", progress, completionPercent,
                                         completionPercent >= 100 ? Environment.NewLine : "\r");
                Console.Write(progress);
                // Update percent complete value in the user data.
                ((ProgessUserData)update.UserData).lastPercentComplete = completionPercent;
                if (userDefinedPauseAtPercentage > 0 && completionPercent >= userDefinedPauseAtPercentage)
                {
                    userDefinedPauseAtPercentage = 0;
                    return NotificationUpdateStatus.PAUSE;
                }
            }
            return NotificationUpdateStatus.CONTINUE;
        }
```

One-time Event: Client Device Registration with FlexNet Operations

If the hostid for the Updates and Insights client device has not been previously registered with Revenera-hosted FlexNet Operations, either through Updates and Insights or FlexNet Embedded client functionality, the client code must perform this process as pre-requisite to obtaining notifications.

Registration with FlexNet Operations is a one-time event for a client device. It involves sending a message that contains a valid rights ID for the customer's account to the back office. The back office is then able to associate the device hostid sending the message with the customer account. Once the client device has been registered with the back office, it does not need to be re-registered.

Note • *In FlexNet Embedded, a client device is automatically registered when the client requests licenses directly from the Revenera-hosted FlexNet Operations. To register when using a Cloud Licensing Service instance to obtain licenses, the client device must send a valid rights ID to the back office, as described in Register the Client with the Cloud Licensing Service. Either way, if the device has already been registered through FlexNet Embedded, it does not have to be re-registered through Updates and Insights and vice-versa.*

The following sections describe the registration process initiated by the Updates and Insights client:

- The Registration Process
- Requesting the Client Registration

The Registration Process

If an end user knows or cannot confirm that the device has not been registered, the Updates and Insights client code can initiate the registration by sending a registration request to the notification server, which in turns checks with FlexNet Operations to obtain the registration status. If the device has been previously registered (for example, through FlexNet Embedded), the notification server sends a response back to the client with a message stating as such. If the device has not been registered, the registration process proceeds in FlexNet Operations. Once the registration completes or fails, the notification server sends a response to the client with a message indicating the status.

However, if the end user knows that the client device has already been registered, best practice is not to initiate the a registration process as matter of course. This process consumes resources as the client continues to poll the notification server until the server sends a response with the registration status.

Requesting the Client Registration

This excerpt from the example calls the following functions (from the `IUAIClient` interface) required to retrieve a notification collection from the server:

- The `Register` function that sends the registration request to the notification server (referenced by `nsServerURL`) to obtain all available notifications for the product package. The rights ID (labeled with the `ActivationId` pointer) must be a valid rights ID associated an entitlement in FlexNet Operations that is mapped to the customer account under which the device is being registered. The entitlement itself must contain an unmetered, uncounted license for the purpose of simply identifying the client device.

  In a production environment, the producer should provide the customer with the appropriate rights ID with which to register the device. This information is typically conveyed through an email message.

- The `RegistrationComplete` function that polls the notification server (at an implementation-defined interval) until the client receives a response message from the server containing the registration status or until an error occurs.

Note that the example code leverages the .NET `BackgroundWorker` class to spawn a new thread for asynchronous communications with the notification server to register the client. This implementation is for demonstration only. Should you want the client to spawn a separate communications thread for the registration process, you can use any method appropriate for your application.

Table 7-7 • Excerpt from `Notification.cs`: Requesting Client Registration

```
private static void registrationThread_DoWork(object sender, DoWorkEventArgs e)
{
    BackgroundWorker registrationThread = sender as BackgroundWorker;
    bool registrationComplete = false;
    int ii = 0;
    e.Result = -1;
    try
    {
        BackOfficeErrorCode serverStatus;
        uaiClient.Register(nsServerUrl, activationId, null);
        do
        {
            Thread.Sleep(1000);
            registrationThread.ReportProgress(++ii < 100 ? ii : 100);
            registrationComplete = uaiClient.RegistrationComplete(activationId, null, out serverStatus);
        } while (!registrationComplete);
        if (serverStatus == BackOfficeErrorCode.FLX_BOS_ERR_SERVER_CLIENT_ALREADY_REGISTERED)
        {
            Util.DisplayMessage("Client is already registered with the back office.");
        }
        e.Result = 0;
    }
    catch (Exception exc)
    {
        threadException = exc;
    }
}
```

About Client Communications for the Updates and Insights

Updates and Insights client communications are handled internally by FlexNet Embedded client functionality. However, in some environments, the default communications settings might need to be adjusted. In this case, best practice is to create a communications (`IComm`) object and specify the customized settings—such as proxy settings, transfer rates, user credentials, and others. You can create a customized `IComm` object to tailor communications settings for the following:

- To pass in with the `IUAIClient.Register` method
- To associate with a product package object
- To pass in with the `Download`, `DownloadAndExecute`, or `ResumeDownload` method in the `INotificationUpdate` interface

For example, this excerpt shows how to create an `IComm` object and associate it with a product. Additional properties are used to associate various settings to the communications object. (For more information, see the API reference.)

Table 7-8 • Excerpt from `Notification.cs`: Creating a Custom Communications Object for a Product Package

```
                    using (productPackage = uaiClient.GetProductPackage(myProductPackageId, myLanguage,
                                        myPlatform))
                {
                    if (createProductPackageComm)
                    {
                        using (IComm comm = CommFactory.Create())
                        {
                            productPackage.Comm = comm;
                            // Set notification server communications options such as
                            // proxy server or client credentials here.
                            GetNotifications();
                        }
                    }
```

This excerpt show how to create an `IComm` object for the download of payloads for "update" notification items and pass it in with a method (in this example, `DownloadAndExecute`):

Table 7-9 • Excerpt from `Notification.cs`: Creating a Custom Communications Object for Downloads

```
                        if (downloadUpdates)
                        {
                            if (createDownloadComm)
                            {
                                using (IComm dlComm = CommFactory.Create())
                                {
                                    // Set content delivery server communications options such as
                                    // proxy server or client credentials here.
                                    dlComm.MaxTransferRate = maxRate;
                                    DownloadUpdates(dlComm);
                                }
                            }...
...updateNotification.DownloadAndExecute(downloadLocation, downloadFilename, MyStatusDelegate, dlComm);
```

8

Utility Reference

Both the FlexNet Embedded Client .NET XT and FlexNet Embedded Client .NET Core XT toolkits include a set of test and configuration utilities, located in the *install_dir*/bin/tools directory, to help you test and prepare your toolkit for production use, as described in this chapter.

Note • *Only the utilities specific to FlexNet Embedded are applicable to the FlexNet Embedded Client .NET Core XT toolkit since this toolkit supports FlexNet Embedded, but not Updates and Insights.*

These toolkit utilities require Java 1.8 or later to be available on your development or test system. Java is *not* a requirement for a host running license-enabled or updates-enabled code.

Tools shared by FlexNet Embedded and Updates and Insights:

- Publisher Identity Utility
- Print Binary Utility

Tools specific to FlexNet Embedded:

- Identity Update Utility
- License Conversion Utility
- Trial File Utility
- Capability Server Utility
- Capability Request Utility
- Capability Response Utility
- Secure Profile Utility
- .NET XT Toolbox

Tools Shared by FlexNet Embedded and Updates and Insights

The following tools are available to assist you with either FlexNet Embedded or Updates and Insights functionality:

- Publisher Identity Utility
- Print Binary Utility

Publisher Identity Utility

The Publisher Identity utility `pubidutil` enables you to create producer-specific identification data in binary format. The back-office identity data is used by your back-office tools for digitally signing license rights and notification messages, the client–server identity is used for served licenses, and the client identity data is used by your FlexNet Embedded or Updates and Insights code to validate your license rights or notification messages, respectively, and to perform other validation operations.

Purpose

The Publisher Identity utility `pubidutil` enables producers to create API-compatible producer-specific identification data in binary format. This binary data will be passed as a buffer to the appropriate methods that create the core Licensing and Updates and Insights objects. The identity is also used to sign and validate binary messages sent between the FlexNet Embedded client and FlexNet Operations, the license server, or the Updates and Insights notification server.

The Publisher Identity utility is provided to enable each producer to create unique identity files that are used to digitally sign the following:

- FlexNet Embedded license files, trial rights, and capability responses
- Product notifications sent from the notification server

Three files are generated:

- The producer's back office identity, containing all public and private key information (by default called `IdentityBackOffice.bin`)
- The license server identity (by default called `IdentityClientServer.bin`)
- The client identity (used by the client code), containing only the public key used to validate signatures (by default called `IdentityClient.bin`)

Important • *It is essential that your back-office identity file (like "IdentityBackOffice.bin"), which contains your private-key information, and the license-server identity file (like "IdentityClientServer.bin") be kept secure.*

Usage

The Publisher Identity utility can be run in a command-line shell or in an user interface. The command takes optional arguments that specify where the identity files will be written and whether it runs in console mode (no GUI).

```
pubidutil [-backOffice backofficeidfile.bin] [-clientServer clientserveridfile.bin]
    [-client clientidfile.bin] [-console] [-listRsaTypes]
```

The default value for the `-backOffice` option is `IdentityBackOffice.bin`, the default for `-clientServer` is `IdentityClientServer.bin`, and the default for the `-client` option is `IdentityClient.bin`.

To generate your binary identity files, run the `pubidutil` script in the `bin\tools` subdirectory of the toolkit:

```
pubidutil
```

With no arguments, it will bring up a user-interface form that can be used to generate the identity files.

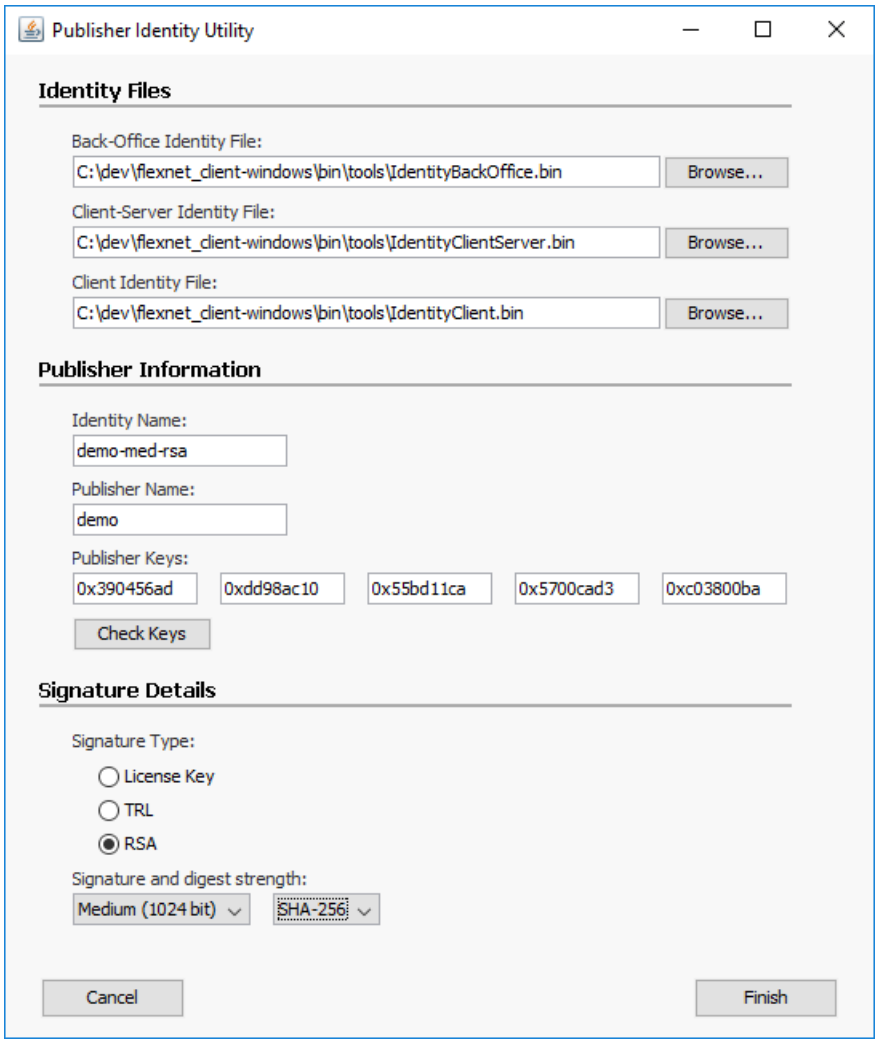


Figure 8-1: Pubidutil in GUI Mode

To avoid the use of the user interface, use the `-console` switch:

```
pubidutil -console
```

You will prompted for all of the required information at the command line. (You can run `pubidutil -?` to obtain additional usage information.)

If you specify existing identity files, you can modify the previous settings and then regenerate the identity files. Naturally, if you update your producer identity files, it will likely be necessary to update the client-identity buffer data in your license-enabled or notification-enabled code to correspond with the back-office identity data used in your back-office server to sign licenses.

***Caution** • In a production environment, best practice is to generate the identity information in FlexNet Operations and then export the client, client–server, and back-office identity files for use with the FlexNet Embedded Client .NET XT or .NET Core XT toolkit. The identity files you use throughout an environment must all be generated at the same time; mixing identity data generated at different times or with different tools—for example, using client-identity data generated with "pubidutil" with back-office identity data generated with FlexNet Operations—will result in a run-time error.*

Entering Your Identity Data

The utility prompts you to enter the following producer-specific data:

- **Identity Name:** Enter a unique name for this collection of identity settings (the name can be used by your back-office server to distinguish among different collections representing different types of clients, for example).
- **Publisher Name:** Enter the producer name provided by Revenera, or `demo` for the evaluation toolkit.
- **Publisher Keys:** Enter the producer keys (five hexadecimal numbers) provided by Revenera.
- **Signature Type:** Choose `RSA`, `TRL`, or `License Key` (the demo toolkit uses RSA signatures on most architectures). The digital signature algorithm and signature strength you select are used for digitally signing licenses, capability response envelopes, and product notification messages.
- **Signature and digest strength**—For signature types that have multiple encryption strength options, choose the signature strength you prefer. Keep in mind that the higher the strength, the more computational overhead is incurred by the signature.

  For RSA, SHA-1 and SHA-2 hash algorithms are available. For a list of all RSA signature strengths and digests, use `pubidutil -listRsaTypes`.

When you have entered all of the required data, `pubidutil` creates the output binary files you specified, or uses the default names if none were specified. (The utility reports an error if the producer keys are invalid or evaluation keys have expired.) The back-office identity file is used when signing licenses and notification messages, the client–server identity is used with served licenses, and the client identity file is used in license-enabled and updates-enabled code.

Further Tasks and Considerations

Once you have generated the identity data, notes the following:

- You can configure the client identity binary to include hostid filtering and caching parameters for use during hostid detection on the client device. For more information, see Identity Update Utility. (This configuration is available for identities generated for FlexNet Embedded C XT, .NET XT, .NET Core XT, and Java XT client applications only.)
- The `printbin` utility, described below, converts the binary client-identity data into C# code that can be copied into license-enabled code.

- For security reasons, the identity data in `IdentityClient.bin` or `IdentityClientServer.bin` should generally not be read from an external binary file into a buffer at run time, unless the file containing the identity data is in a locked-down part of the client storage.
- If you are using the Capability Server Utility (which is the test back-office server, `capserverutil`) for FlexNet Embedded licensing, you need to specify the back-office identity data when starting the server.

Print Binary Utility

The print-binary utility `printbin` displays human-readable contents of binary files such as the following:

- FlexNet Embedded binary license file, capability request, capability response, or trial file created with `licensefileutil`, `caprequestutil`, `capresponseutil`, or `trialfileutil`, respectively
- Request and response messages sent between the Updates and Insights client and the notification server

The print-binary utility also displays the contents of a binary identity file created with `pubidutil`, and optionally converts it to C-compatible format for use in your compiled license-enabled and notification-enabled code.

Viewing Contents

To view the binary's contents (mainly keys and their values), use the command:

```
printbin binaryFile.bin
```

To display all the contents of the binary file, use the `-full` switch.

Caution • *When using the "-full" switch, be aware that some information displayed might be used internally by Revenera for troubleshooting purposes and is subject to change between releases. Do not rely on this information for your own troubleshooting or coding purposes.*

Viewing Contents and Validating Signatures

To view contents of a binary file (including a FlexNet Embedded license file or Updates and Insights message file) and validate any signatures contained in the file, use the command:

```
printbin -id IdentityBackOffice.bin binaryFile.bin
```

where `IdentityBackOffice.bin` is the producer identity file that you previously created with the Publisher Identity Utility and used to sign and convert the binary file.

Displaying Binary-File Contents in Compiler-Readable Format

To display contents of a binary identity file, license file, or trial file in a format that can be used in C# code (typically as a hard-coded array to be passed to FlexNet Embedded or Updates and Insights functions that require the binary data), use the `-cs` switch (and optionally the `-package` switch, for the name of a .NET namespace):

```
printbin -cs IdentityClient.bin
```

The resulting array will be directed to the console, or you can add the `-o outputFile` switch to save the output in a file. The output for a client-identity file should look similar to the following:

```
/*
    PublisherIdentityVersion=0
    IdentityName=demo-med-rsa
    PublisherName=...
    ...
    SignatureType=RSA
    SignatureStrength=1
*/
static const unsigned char identity_data[] = {
    0x00, 0x00, 0x0f, 0x68, 0x00, 0x00, 0xa9, 0x2f, 0xff, 0xff,
    0xff, 0xff, 0x00, 0x00, 0x00, 0x0b, 0x00, 0x72, 0x00, 0x00,
    ...
    0xaa, 0x87, 0x8f, 0xc3, 0xf1, 0xf5, 0x3c, 0x63, 0x2f, 0x29,
    0x10, 0x4b, 0x0e, 0x60
};
```

You can then compile the client identity data in your FlexNet Embedded or Updates and Insights code.

You can use this technique if your version of FlexNet Operations does not directly export C#-compatible identity data. First, download your binary client-identity data `IdentityClient.bin`, and then run **`printbin -cs IdentityClient.bin -o IdentityClient.cs`**.

Converting License Data to Base 64 Format in FlexNet Embedded

For FlexNet Embedded functionality, the `printbin` utility can convert binary licensing data into a base-64 encoding, which is useful in cases where a binary file cannot be conveyed to the end user of a client system, but where an encoded text representation can be used. To display a binary license file in Base 64 format, for example, use the switch `-base64`:

```
printbin -base64 license.bin -o license64.txt
```

In order to query or acquire a Base 64 license, the license-enabled code must decode the base-64 data before passing it to the FlexNet Embedded functionality.

Note • *FlexNet Embedded uses the base-64 encoding used by MIME. The encoded data supports the letters A–Z and a–z, numerals 0–9, and "+" and "/" characters. It also uses "=" as padding character, encodes lines with a maximum of 76 characters, and uses CRLF as line separator.*

Additional printbin Switches

Additional switches to `printbin` include:

- `-ident` *`identifier`*: An array variable identifier other than `IdentityData` when using the `-cs` switch.
- `-compact`: Option that displays file contents in a compact format, which can be useful for large license files, for example.
- `-long`: Option that displays contents in an expanded, multi-line format.
- `-raw`: Option that displays contents using internal property names instead of "friendly" names (`-raw` and `-compact` cannot both be used).

Tools Specific to FlexNet Embedded

The following tools are available specifically to help you use FlexNet Embedded functionality:

- Identity Update Utility
- License Conversion Utility
- Trial File Utility
- Capability Server Utility
- Capability Request Utility
- Capability Response Utility
- Secure Profile Utility
- .NET XT Toolbox

Identity Update Utility

The FlexNet Embedded Identity Update utility sets up filtering and caching parameters for use during hostid detection on a FlexNet Embedded client device. This configuration is injected into the binary containing the identity data for your FlexNet Embedded client applications and is retrieved whenever a FlexNet Embedded client function, such as the `getHostid` API or method, is called to detect available hostids on the client device. The configuration helps to reduce hostid retrieval time by limiting the detection process to specific hostid types and (optionally) by caching retrieved hostids for future hostid-detection calls.

This utility supports the configuration of client identities for applications that you create with the FlexNet Embedded C XT, .NET XT, .NET Core XT, or Java XT SDK. It does not support the configuration of client identities for applications created with the FlexNet Embedded C SDK; nor does it support the configuration of a FlexNet Embedded license server identity.

The following describes the Identity Update utility:

- Usage
- Device Hostid Types Used to Restrict Hostid Detection
- Example Identity Update

For more information about generating the identity binary for a FlexNet Embedded client application, see Publisher Identity Utility.

Usage

Usage for the Identity Update utility is as follows:

```
identityupdateutil -help |
            [-restrict-device-id-detection type]
            [-enable-device-id-caching type]
            [-caching-duration seconds]
            input-identity-file output-identity-file
```

The following is a description of the utility arguments:

- `-restrict-device-id-detection` *type*: The hostid type to which to restrict hostid detection on the client device. Repeat this argument for each additional hostid type to which you want to restrict detection. See Device Hostid Types Used to Restrict Hostid Detection for information about the hostid types you can specify.
- `-enable-device-id-caching` *type*: (Optional) The hostid type for those detected hostids that you want to cache for future detection on the client device. (The hostid type must be specified for a `-restrict-device-id-detection` argument in the current command.) Repeat this argument for each hostid type you want to specify for caching. See Device Hostid Types Used to Restrict Hostid Detection for more information.

  Note the following:

  - To cache all detected hostids, use the `all` value.
  - When caching any removable hostid such as a dongle or a removable Ethernet adapter, Revenera recommends that you also specify a cache duration to avoid license leakage.
  - If the `-enable-device-id-caching` argument is not included in the command, hostid caching is disabled.
- `-caching-duration` *seconds*: (Optional) The duration in seconds for which detected hostids are held in cache, after which cache is reset. Specify this argument only if caching is enabled (that is, one or more `-enable-device-id-caching-type` arguments are specified). Note the following:

  - The duration value is applied to all cached hostids.
  - The maximum value is 2^{32} seconds (specified in decimal format only).
  - If caching is enabled and this argument is set to `0` or omitted, the detected hostids remain cached until the current process is exited.
- *input-identity-file*: The relative path and name of the binary file containing the client identity you are updating.
- *output-identity-file*: The relative path and name of the binary file to which you are outputting the updated client identity. (The utility creates or overwrites this file as needed.)

Device Hostid Types Used to Restrict Hostid Detection

The `-restrict-device-id-detection` argument restricts the type the hostids that you want to retrieve during hostid detection on the client device. The following table provides a brief description of the hostid-type values you can specify for this restriction and discusses any special considerations for a given value. You can specify more than one hostid-type value to enlarge the range of detected hostids.

Table 8-1 • Values for Hostid Types Used to Restrict Hostid Detection

Value for Hostid Type Restriction	Detects...
mac	Certain types of MAC (Ethernet) hostids. Hostid detection using this value is generally faster than the detection process using `mac_ecmc`, which also detects MAC hostids (as described next). However, the `mac` value might not detect certain MAC hostids and therefore is not so reliable in ensuring MAC hostid retrieval. If hostid detection fails with the `mac` value, use `mac_ecmc` instead. Alternatively, to detect the greatest number of MAC hostid types, use `mac` and `mac_ecmc` in parallel.

Table 8-1 • Values for Hostid Types Used to Restrict Hostid Detection

Value for Hostid Type Restriction	Detects...
mac_ecmc	Almost any type of MAC hostid. Hostid detection with this value might be slower than a detection process that uses `mac`, but it is more reliable in ensuring the retrieval of MAC hostids on the client machine. If the detection process for this hostid type seems slow, consider caching the detected hostids (that is, include the argument `-enable-device-id-caching mac_ecmc`).
vmuuid	The UUID of the virtual machine on which the client application is running. If the virtual-machine detection is disabled on the client device (through the appropriate FlexNet Embedded API or method available in your SDK), specifying `vmuuid` for `-restrict-device-id-detection` will not retrieve a hostid.
flexid9 **OR flexid10**	The hostid of the Aladdin dongle or the Wibu-Systems dongle, respectively, on the client device. If restricting hostid detection with either of these hostid types, you are strongly recommended not to enable the hostid type for caching.
ip4, ip6, OR ip_all	The IP version 4, IP version 6, or all IP addresses, respectively, on the client device.
user	User hostids (user IDs used to log on to the client device).
container_id	The ID of the container in which the client application is running.
all	Hostids for all hostid types valid for retrieval.

Example Identity Update

The following is an example command for the Update Identity utility:

```
identityupdateutil -restrict-device-id-detection mac -restrict-device-id-detection ipv4 -enable-
    device-id-caching mac -caching-duration 500 IdentityClient.bin IdentityClient_out.bin
```

The command will configure the FlexNet Embedded client to limit device hostid detection to MAC and IPv4 hostids only and will cache all detected MAC hostids for 500 seconds. The utility is run against the client identity data in `IdentityClient.bin` and the updated identity is output to `IdentityClient_out.bin`.

The following shows the contents of `IdentityClient_out.bin` when you run `printbin` (see Print Binary Utility). The client-identity configuration information, represented as a 32-bit unsigned, encoded integer, is displayed for the `XtConfiguration` property.

```
IdentityName=demo-med-rsa
PublisherName=demo
PublisherKey=9ddcd080
PublisherKey=99aa8309
PublisherKey=33995896
PublisherKey=86971320
PublisherKey=b165dcb
```

```
SignatureType=RSA
SignatureStrength=1
XtConfiguration=00020005010100002f403
```

License Conversion Utility

The FlexNet Embedded license conversion utility `licensefileutil` converts a human-readable, text-based, unsigned license file into the FlexNet Embedded binary format. This binary format is commonly used when pre-loading license rights on a client.

Before converting any license files, you must create a producer back-office identity file that specifies how the converted license file will be signed, using the Publisher Identity Utility.

The tool syntax is:

```
licensefileutil -id id-file text-license binary-license
```

The arguments include:

- `-id` *`id-file`*: Name of the file containing your producer back-office identity, required to digitally sign license
- *`text-license`*: Name of the unsigned license text file containing one or more feature definitions
- *`binary-license`*: Name of the signed binary license file to create

For example, create a text license file (called `unsignedInput.lic`, for example) using the feature-definition syntax described in Feature Definitions.

```
INCREMENT survey demo 1.0 permanent uncounted HOSTID=ID_STRING=1234567890
INCREMENT lowres demo 1.0 1-jan-2025 uncounted HOSTID=ID_STRING=1234567890
```

Next, run `licensefileutil` to generate a binary version of the license file:

```
licensefileutil -id IdentityBackOffice.bin unsignedInput.lic signedLicenseOutput.bin
```

Copy the binary output file—`signedLicenseOutput.bin`, in this example—to the client system. Now you can run license-enabled code that acquires the license rights, such as the **Client** example program. The diagnostic **View** example also prints a summary of feature information contained in a binary license file.

You can also use `licensefileutil` to sign a license file to be served by a license server; that is, a license file containing a `SERVER` line along with one or more `INCREMENT` lines containing count values.

Trial File Utility

The FlexNet Embedded trial file utility `trialfileutil` enables you to generate signed binary trial license rights, which can then be processed by license-enabled code such as the **Trials** example. Such trial license rights can be loaded on a client to enable a customer to test product functionality for a limited duration. Trial license rights can also be loaded to act as an emergency license.

Its usage is:

```
trialfileutil -id id-file -product product-id [-expiration date] [-duration seconds]
              [-trial trial-id] [-once | -always] lic-file binary-file
```

The arguments include:

- `-id` *`id-file`*: Name of the file containing your producer back-office identity, required to digitally sign the trial rights
- `-product` *`product-id`*: Product ID for the trial
- `-expiration` *`date`*: Optional expiration date for the trial, in *`dd-mmm-yyyy`* format (e.g., 1-jan-2020), or "permanent" (the default)
- `-duration` *`seconds`*: Trial duration in seconds, rounded up to the nearest day (default is 1 day; each day is 86,400 seconds)
- `-trial` *`trial-id`*: Unique numeric trial ID for the trial (defaults to 1; valid values are integers from 1 through 65535)
- `-once`: Option indicating that the trial can be loaded only once on a client system; default unless `-always` is specified
- `-always`: Option indicating that the trial can always be loaded on a client system (rarely used)
- *`lic-file`*: The text-formatted license file listing features included in trial
- *`binary-file`*: Name of the binary trial file to create

For example, create a text license file—naming it `unsignedInput.lic`, for example—using the feature-definition syntax described in Feature Definitions.

```
INCREMENT survey demo 1.0 permanent uncounted
INCREMENT highres demo 1.0 permanent uncounted
```

For a trial, the feature need not include a `HOSTID` value. Similarly, the trial's expiration, whether based on duration or explicit expiration date, overrides the expiration date of all features in the unsigned text license; and the trial's activation date overrides any start date (`START` keyword value) specified for a feature.

Next, run `trialfileutil` to generate a binary version of the trial:

```
trialfileutil -id IdentityBackOffice.bin -product SampleApp -trial 1 -duration 86400
    unsignedInput.lic signedTrialOutput.bin
```

Copy the binary output file—`signedTrialOutput.bin`, in this example—to the client system. You can now run license-enabled code that processes and acquires the trial license rights, such as the **Trials** example program. The example command uses the default trial duration of one day from the time the trial rights are processed.

Capability Server Utility

The Capability Server utility is used to test and debug the online capability exchange functionality of a FlexNet Embedded client application. The utility functions as a simple back-office server that receives `HTTP POST` capability requests from clients—FlexNet Embedded license-enabled code or local license servers—and then generates and returns capability responses. The utility also accepts synchronization messages from license servers (but generates no client records).

Caution • *The Capability Server utility is provided for testing purposes only; it is not intended for use in a production environment.*

Refer to the following for more information about using the Capability Server utility:

- Considerations for Using the Utility
- Usage

- Starting and Stopping the Capability Server Utility
- About License Templates
- Endpoint for Sending Capability Requests to the Utility

Considerations for Using the Utility

The Capability Server utility operates as a simple back-office server. Before using this utility, consider the following:

- **No license accounting**—The utility performs no accounting of license rights. That is, it does not limit the number of licenses issued. If a client requests 100 copies of particular license right (that is, license template), the utility activates 100 copies. If another client requests 100 copies of the same license right, the utility activates another 100 copies. It never responds with an "insufficient count" message.
- **No CLS support**—The utility supports the FlexNet Embedded client application and the FlexNet Embedded local license server, but not the Cloud Licensing Service (CLS) license server, as clients.
- **No client records generated**—The utility does not create or manage client records during synchronization from a license server.
- **Console mode only**—The utility runs only in console mode, not as a daemon or service.
- **Other limitations**—It does not support HTTPS, synchronization recovery, or license-server failover.
- **Response lifetime**—The lifetime of a capability response generated by the Capability Server utility is 1 minute.

Usage

The following shows the usage for the Capability Server utility:

```
capserverutil -help | -id id-file -template template-dir [-port port] [-v]
```

Arguments include the following:

- `-id` *`id-file`*: The binary file containing the back-office identity, required to digitally sign the capability response.
- `-template` *`template-dir`*: The directory containing license templates that the utility uses to store and manage license rights. A sample `templates` directory is provided in the `bin\tools` directory where the utility resides, but you can specify your own location (making sure that you include the appropriate path). See About License Templates for more information.
- `-port` *`port`*: The port on which the utility listens. If no port is specified, 8080 is used.
- `-v`: The flag to provide more detail in the utility output.

Starting and Stopping the Capability Server Utility

To start the Capability Server utility, enter the **`capserverutil`** command similar to this (which, in this case, assumes the default port 8080 and specifies the detailed format for output):

```
capserverutil -id IdentityBackOffice.bin -template templates -v
```

To stop the utility, press Enter.

About License Templates

The utility uses license templates as means of organizing license rights to simulate a real back-office server system. Each template, identified by either a client device hostid (*device_hostid*.lic) or a rights ID (*rightsID*.lic), stores a set of licenses. You can create your own license templates or use the sample license templates found in the bin\tools\templates directory. These two sample templates, 1234567890.lic and li1.lic, work easily with the example applications and test tools (included in the SDK) that use a back-office server, but you can use the samples for your own tests.

The following sections provide more information about how the Capability Server utility uses the templates to activate license rights:

- Use of License Templates to Generate Responses
- Examples
- Creating a License Template

Use of License Templates to Generate Responses

When license-enabled client code sends a capability request to the Capability Server utility, the utility follows this general process to generate a capability response.

It first searches for a license template with a name that matches the hostid of the device sending the request. If a match exists, the utility returns a capability response with the licenses found in that license template, but ignores any rights ID sent in the request.

However, if no license template matches the device hostid sending the capability request, the utility then searches for a license template with a name that matches a rights ID sent in the request. If a match exists, the utility returns a capability response with the licenses found in the license template.

Examples

The examples described next use the sample license templates, located in the bin\tools\templates directory, to illustrate how the Capability Server utility activates license rights. Consider the contents of these sample license templates:

- The 1234567890.lic template contains the following licenses:

```
INCREMENT survey 1.0 permanent 1
INCREMENT highres 1.0 permanent 1
```

- The li1.lic template contains these licenses:

```
INCREMENT f1 1.0 permanent 5
INCREMENT f2 1.0 permanent 10
```

Keep in mind that the client in the following examples can be either FlexNet Embedded license-enabled code or a FlexNet Embedded license server.

Example 1: Activate all rights mapped to a client device

Suppose the client on device hostid “1234567890” sends a capability request without a rights ID. To simulate the search for all license rights mapped to the device hostid, the Capability Server utility looks for a license template with a name matching the hostid. When it locates license template 1234567890.lic, it sends a capability response containing a copy of the rights in that template (that is, **1** count of survey and **1** count of highres).

A rights ID sent in the same capability request would be ignored in this case.

Example 2: Activate a a specific rights ID

Suppose the client on device hostid "1111" sends a capability request for **2** copies of the rights ID `li1`. To simulate the search for a specified rights ID, the utility first looks for a license template with a name matching the device hostid ("1111"). Finding no matching template, it locates the license template (`li1.lic`) matching the rights ID and sends a capability response containing **2** copies of all rights in that template—that is, **10** counts of `f1` (2 times the initial 5 counts) and **20** counts of `f2` (2 times the initial 10 counts).

Creating a License Template

To create a license template, you can use one of the sample license templates as a basis. In a text editor, set up an `INCREMENT` line for each feature, providing the feature's name, version, expiration, and count, and defining any additional attributes as needed (see Feature Attributes to Consider Adding). The following shows sample contents for a license template. The same content format is used whether you are defining license rights for a FlexNet Embedded client or a license server.

```
INCREMENT f1 1.0 permanent 6
INCREMENT f2 1.0 permanent 10 VENDOR_STRING="global"
INCREMENT f3 1.0 1-jan-2025 5
INCREMENT m4 1.0 permanent 15 METERED UNDO_INTERVAL=120
```

Save the file, using a device hostid or a rights ID for the file name and adding the `.lic` extension. The utility does not discern the hostid type; it simply processes it as a string.

Keep in mind that, when a license template containing both non-metered and metered features is used to satisfy a capability request from a license server, both the metered and the non-metered features are activated on the server. If the same template is used to satisfy a request from a FlexNet Embedded client application, only the non-metered features are activated on the client (since the client can obtain metered features only through a license server).

Feature Attributes to Consider Adding

The following lists some feature attributes you might want to add for a given feature. (This is not an exhaustive list, just a list of suggestions. For more information about attributes, see Feature Definitions in the *Toolkit Overview* chapter.)

- `START` (if not explicitly identified, the start date is set by the Capability Server utility to one day prior to the date of issue)
- `ISSUED`
- `VENDOR_STRING`
- `SN`
- `ISSUER`
- `METERED` (metered feature only)
- `REUSABLE` (metered feature only)
- `UNDO_INTERVAL` (metered feature only, incompatible with `REUSABLE`)

To keep the license template generic for testing purposes, best practice is to not add the vendor (producer) name, such as `demo`, as feature attribute.

Endpoint for Sending Capability Requests to the Utility

Use the following endpoint when sending a capability request to the Capability Server utility:

```
http://capserverutil_IP_address:capserverutil_port/request
```

The endpoint includes the following components:

- ***capserverutil_IP_address***—The IP address on which the Capability Server utility is running; or, if it is running on your local machine, the value `localhost`.
- ***caperservertutil_port***—The port on which the utility listens (by default, 8080).

The following is an example endpoint:

```
http://localhost:8080/request
```

Capability Request Utility

The FlexNet Embedded capability request utility `caprequestutil` enables you to manually generate a capability request and save it as a file or send it to a back-office server. Its common usage is as follows:

```
caprequestutil [-idtype idtype] -host host_id
    [-id pubidfile | -publisher name identity identity-name] [-name machine-name]
    [-type host-type] [-server | -client] [-serverInstance instance-number]
    [-attr key1 val1 ...] [-machine machine-type] [-vmname name]
    [-vmattr key value] [-selector key value] [-force] [-incremental]
    [-timestamp seconds | -response file | storage file | storageDir dir]
    [-activation activation-id [copies]...] [-feature name version [count]...]
    [-requestor requestor-id] [-acquisition acquisition-id] [-enterprise enterprise-id]
    [-correlation correlation-id] [-bindingBreakType binding-break-type]
    [-bindingGracePeriodEnd binding-grace-period-end] [-operation op-type] binary_file|server_url
    [response_file]
```

This utility is intended for quick testing involving capability requests. It is implemented using Java, and therefore does not make use of the callouts or other information used in "native" FlexNet Embedded code, but instead uses a text file as a substitute for trusted storage.

The following are commonly used arguments and switches. (For a complete list of arguments to the capability request utility, run `caprequestutil` with the `-help` switch. Some switches are used only for infrequently encountered scenarios.)

- `[-id` *`pubidfile`* `| -publisher` *`name`* `identity` *`identity-name`*`]`: The binary producer client identity file (normally called `IdentityClient.bin`), created using `pubidutil`; or the producer and identity name used to create the identity.
- `-host` *`host_id`*: The client hostid or transaction ID to use.

  To specify one or more secondary hostids, repeat this argument for each additional hostid. The first hostid used with this option will be considered the main hostid. Each subsequent hostid is considered secondary. When license reservations are used, only the main hostid and the first secondary hostid are used in the reservation search. (For more information, see in the Secondary Hostids section of the *Using* FlexNet Embedded *APIs* chapter.)
- `-idtype` *`type`*: The hostid type, one of `any`, `ethernet`, `flexid9`, `flexid10`, `internet` (for IPv4), `internet6` (for IPv6), `string` (the default), `vmuuid`, or `container_id`. (Some of these types are supported by certain license server versions only.) When specifying multiple hostids (see the previous `-host` description), you can repeat this argument to specify a different hostid type for a given hostid.
- `-server | -client`: Flag to identify the request as coming from a license server or client (for testing).

- `-serverInstance` *`instance-number`*: (For use in a multiple-source regenerative licensing environment) The number identifying the target license-server instance (for example, **5** for server instance 5) to which the capability request is being sent. This ID is optional in the capability request. When echoed back in the capability response, it is used to verify the location in client trusted storage where the requested licenses are to be stored. For more information, see Advanced Topic: Multiple-Source Regenerative Licensing in the *Using the* FlexNet Embedded *APIs* chapter.

- `-name` *`name`* and `-type` *`type`*: Optional host name (alias) and type, used in some logging and back-office-server scenarios.

- `-machine`: One of `physical`, `virtual`, or `unknown` (the default). If set to `virtual`, the `-vmname` and `-vmattr` switches populate the virtual-machine name and dictionary attributes.

- `-timestamp` *`seconds`*: Time stamp of the message; if unset, uses system time in seconds since midnight (UTC), 1 January 1970.

- `-storage` *`file`*: A text-formatted `.properties` file—used in place of trusted storage—containing the time stamp to use in the message (value 1 is used if the file does not contain a time stamp or does not exist); the contents of the file are updated with a new time stamp if a valid response is received.

- `-storageDir` *`dir`*: The directory where `.properties` files are placed. The directory is created if it does not already exist. The names generated for the `.properties` files are based on the producer name, hostid and hostid type, and whether the host sending the request is a server or a client.

- `-response` *`file`*: An existing capability response file containing the timestamp to be used in the message.

- `-activate` *`activation_id`* `[`*`copies`*`] [partial]...`: One or more activation IDs (also called *rights IDs*) to add to the request meant for a back-office server such as FlexNet Operations; each activation ID can specify an optional "number of copies" count (count is 1 if omitted).

  The optional `partial` attribute tells the back-office server to send however many copies are available for that activation ID should the available copy count in the back office fall short of the requested count. (Without the "partial" attribute, the back-office server does *not* include the features for the activation ID in the capability response if it cannot satisfy the requested copy count for that ID.) See Attribute to Obtain All Available Copies for a Rights ID If Requested Count Cannot Be Satisfied in the *Using the* FlexNet Embedded *APIs* chapter for details.

- `-feature` *`name version`* `[`*`count`*`] [partial]...`: One or more desired features to add to the request (meant for a local FlexNet Embedded server); count is 1 if omitted.

  The optional `partial` attribute tells the license server to send whatever count is available for that feature should the available count for the feature on the server fall short of the requested count. (Without the "partial" attribute, the server does *not* include the requested feature in the capability response if it cannot satisfy the requested count for the feature.) See Attribute to Check Out All Available Quantity for a Feature If Requested Count Cannot Be Satisfied in the *Using the* FlexNet Embedded *APIs* chapter for details.

- `-attr` *`key1 val1 ...`*: One or more key–value pairs for the request vendor dictionary.

- `-selector` *`key value`*: A key-value pair (called a "feature selector") sent in the request to filter the requested features on the license server. You can specify this option multiple times, one for each "feature selector". The value must be a string, not an integer. See Feature Selectors in a Capability Request in the *Using the* FlexNet Embedded *APIs* chapter for details.

- `-force`: The "force response" flag, which indicates that a server response is required even if license rights on the client have not changed since the last response was processed.

- `-incremental`: Flag to mark the request as "incremental" so that available non-expired licenses currently served to the client are automatically sent in the response along with the available desired features from the license server. See Incremental Capability Requests in the *Using the* FlexNet Embedded *APIs* chapter for details.
- `-requestor` *`requestor-id`*, `-acquisition` *`acquisition-id`*, `-enterprise` *`enterprise-id`*, `-correlation` *`correlation-id`*: Optional IDs in the capability request:
  - The requestor ID is used to associate the client device with a "device user".
  - The acquisition ID identifies the resource that was acquired.
  - The enterprise ID identifies the end-user account on behalf of the acquisition performed.
  - The correlation ID, generated and sent in the response by the license server for a client "request" operation, is used in "undo" operations in a usage-capture scenarios to specify which "request" operation to recall.

  For more details about these IDs, refer to the Using the FlexNet Embedded APIs chapter.
- `-bindingBreakType` *`binding-break-type`*: The type of binding break (`SOFT` or `HARD`) that is currently in effect for the license server.
  - A soft break indicates that a binding break is detected on the license server, but the server can continue to serve licenses. (If the `-bindingGracePeriodEnd` option is included in the request, the soft break changes to a hard break when the grace period expires.)
  - A hard break indicates that a binding break is detected, and the server can no longer serve licenses.

  This information is sent in capability requests from the license server to the back office. The back office can then choose to send a "reset binding flag" in the capability response to repair the break (see Capability Response Utility). For more information about the binding-break detection (with a grace period) feature, see the "Advanced License Server Features" chapter in the *FlexNet Embedded License Server Producer Guide*.
- `-bindingGracePeriodEnd` *`binding-grace-period-end`*: The timestamp indicating when the grace period for a binding break on the license server expires. When this option is included, the SOFT status changes to a HARD status when the grace period expires. For more information about the binding-break detection (with a grace period) feature, see the "Advanced License Server Features" chapter in the *FlexNet Embedded License Server Producer Guide*.
- `-operation` *`op-type`*: The operation of a capability request. The back-office server supports only the "request" operation (and for only concurrent features). The FlexNet Embedded license server supports all operations.
  - `request`—Request that concurrent features be included in the response or that usage for capped metered features be sent. (This is the default if no operation is specified.)
  - `report`—Report usage of metered features.
  - `undo`—Undo usage previously sent for the given correlation ID
  - `preview`—View available feature counts without deprecating counts on the license server or processing features into client trusted storage. Use this operation with either the `-feature` or `-requestAll` option (but not both) to specify features to preview. The operation is not compatible with the `-incremental`, `-feature...partial`, and `-correlation` options. For more information, see Capability Preview in the *Using the* FlexNet Embedded *APIs* chapter.
- *`binary_file`*: File name for the binary request output.

- *server_url*: URL of the back-office server or FlexNet Embedded license server. The request will be sent directly to this URL using HTTP POST—as opposed to using an intermediate file—and `caprequestutil` will parse and print the response received.
  - For FlexNet Operations, a typical value is **`http://hostname:8888/flexnet/deviceservices`** (with modifications to match your FlexNet Operations installation).
  - For the Capability Server Utility (the test back-office server `capserverutil`), provide the value **`http://localhost:8080/request`**.
  - For a local license server, a typical value is **`http://hostname:7070/request`**.
  - For a CLS license server, a typical value is the following: **`https://`***`siteID`***`-uat.compliance.flexnetoperations.com/instances/`***`instId`***`/request`**.
- *response_file*: Name of file in which to save the binary response received.

Capability Response Utility

The FlexNet Embedded capability response utility `capresponseutil` enables you to manually generate a capability response without using a back-office server. Its common usage is as follows:

```
capresponseutil -id pubidfile -host host_id [-idtype type] [-timestamp seconds]
    [-timestamp-milliseconds milliseconds] [-lifetime seconds] [-attr key1 val1...]
    [-status code detail] [-machine machine-type] [-vmname name] [-vmattr key value]
    [-clone] [-server server-id] [-serverIdType type] [-resetBinding]
    [-feature name version [count] [maxCount]...] [-serverInstance instance-number]
    [-preview] text_license binary_response
```

The following describes commonly used arguments and switches. (For a complete list of arguments to the capability response utility, run `capresponseutil` with the `-help` switch. Some switches are used only for infrequently encountered scenarios.)

- `-id pubidfile`: Name of the producer's back-office identity binary file, created using `pubidutil`.
- `-host host_id`: Client host ID to be used.
- `-idtype` *type*: The hostid type, one of `any`, `ethernet`, `internet` (for IPv4), `internet6` (for IPv6), `flexid9`, `flexid10`, `string` (the default), `vmuuid`, `publisher_defined` (for a producer-defined hostid), or `container_id`. If using a producer-defined hostid, include **`PUBLISHER_DEFINED=`***`hostid_value`* in the text license file specified in the command.

  Note • *Some of these types are supported only by the specific version of the license server.*
- `-timestamp` *seconds*: The timestamp (with second granularity) to use in the capability response. The current system time is used if this option is omitted. To provide a timestamp other than the current system time, do one of the following:
  - For a timestamp without a clock time, enter a value in the format `m/d/y`, such as **`02/22/2017`**. (The system generates the clock time as 12:00:00 AM.)
  - For a timestamp that includes a clock time, provide the total number seconds from Unix epoch (00:00:00 January 1, 1970 UTC). For example, the value **`1485820908`** represents January 31, 2017 00:01:48 GMT.

A given FlexNet Embedded client application uses the timestamp to determine whether the capability response is in proper sequence—that is, has a timestamp later than the previous response's timestamp stored in client trusted storage. A response with a timestamp earlier than or equal to the timestamp in trusted storage is rejected as stale. However, you can use this `-timestamp` option in conjunction with the `-timestamp-milliseconds` option (described next) to avoid stale responses that occur when timestamps sent to the given client application are rounded off to the same second.

- `-timestamp-milliseconds` *`milliseconds`*: A millisecond value between (and including) 0 and 999 that adds a millisecond precision to the timestamp indicated by the `-timestamp` value. This precision allows multiple capability exchanges to occur within a second between the local license server and a given FlexNet Embedded client application. If the `-timestamp-milliseconds` option is used without the `-timestamp` option, the response uses the system time including the milliseconds. (If you omit the `-timestamp-milliseconds` value when a `-timestamp` value exists, the response time is rounded to whole seconds.)

  If either the local license server or the client application does not support the millisecond-precision feature, the client application continues to use only the `-timestamp` value (or system time) to process responses.

- `-lifetime seconds`: Lifetime of the response, in seconds (defaults to one day), after which the response is considered "stale" and cannot be processed by the client; a lifetime value of zero indicates a response that will never expire.

- `-attr` *`key1 val1 ...`*: One or more key–value pairs for vendor dictionary.

- `-status` *`code detail`*: A status code and its associated value or detail to include in the capability response. You can repeat this option multiple times.

  An example use of this option is to alert a FlexNet Embedded client of a binding break on the license server. You would supply one of these arguments:

  - `-status SERVER_BINDING_BREAK_DETECTED soft` (soft break)
  - `-status SERVER_BINDING_BREAK_DETECTED hard` (hard break)
  - `-status SERVER_BINDING_BREAK_DETECTED` *`interval`* (break with grace period in effect, specified by *`interval`* shown in `s`, `w`, or `d` units, as in **`1d`** for 1 day)
  - `-status SERVER_BINDING_GRACE_PERIOD_EXPIRED 0` (grace period expired)

  For more information about the binding-break detection (with a grace period) feature, see the "Advanced License Server Features" chapter in the *FlexNet Embedded License Server Producer Guide*.

- `-machine`: One of `physical`, `virtual`, or `unknown` (the default), indicating the type of machine intended to process the response. If set to `virtual`, the `-vmname` and `-vmattr` switches populate the virtual-machine name and dictionary attributes.

- `-clone`: Designation that the client is a clone suspect.

- `-server` *`server-id`*: The hostid of the license server serving the licenses. If not specified, the response is generated as if it is from the back office.

- `-serverIdType`: The hostid type for the license server if it is other than "string" (which is the default.)

- `-resetBinding`: The flag sent in a capability response from the back office to the license server, enabling the license server to repair its broken binding so that it can continue to serve licenses. For more information about the binding-break detection (with a grace period) feature, see the "Advanced License Server Features" chapter in the *FlexNet Embedded License Server Producer Guide*.

- `-preview`: The flag sent in a capability response from the license server (identified by the `-server` and `-serveridtype` options) indicating that the response is in "preview" mode and therefore is not be processed into trusted storage on the client. Use the *`text_license`* file (and the `-feature` option if needed) to specify features to preview. For more information about the capability preview feature, see Capability Preview in the *Using the* FlexNet Embedded *APIs* chapter.
- `-feature` *`name version`* `[`*`count`*`] [`*`maxCount`*`]...`: A feature listed in the *`text_license`* file that you want to include explicitly in the capability response (instead of using all features in the *`text_license`* file). You can repeat this option for multiple features; only features listed with this option are included in the capability response. You can override the current counts for a feature specified with this option. If counts are omitted, `count` defaults to `1`, and `maxCount` defaults to `0`. This option is used only when a license server (identified by the `-server` and `-serveridtype` options) is serving the features.
- `-serverInstance` *`instance-number`*: (For use in a multiple-source regenerative licensing environment) The number of the target license-server instance being echoed back from the capability request (see the previous section Capability Request Utility). This ID is used to verify the location in client trusted storage where the requested licenses are to be stored. For more information, see Advanced Topic: Multiple-Source Regenerative Licensing in the *Using the* FlexNet Embedded *APIs* chapter.
- `text_license`: Name of the unsigned-license text file used as input. If you want the capability response to include only specific features defined in this file (instead of using all features in the file), also use the `-feature` option to identify these features.
- `binary_response`: Name of signed binary response file to create as output.

After generating a capability response, the binary capability response file created with `capresponseutil` should be conveyed to the client system and then processed using code similar to that in the **CapabilityRequest** example included with the FlexNet Embedded functionality. Once the response has been processed, the license rights described in the response are written to trusted storage or a buffer and are available for acquisition.

Secure Profile Utility

The FlexNet Embedded profile utility `secureprofileutil` configures existing client-identity binary data to enable a greater level of anchor security than what is normally provided for trusted storage on machines running your license-enabled applications. (See Advanced Topic: Secure Anchoring for more information.) Before using this utility, you must obtain the file containing your producer client-identity binary data (for example, `identityClient.bin`) from FlexNet Operations or by running the Publisher Identity Utility.

Viewing Available Security Profiles

The `secureprofileutil` utility embeds a secure-anchoring configuration into the identity data, enabling a certain level of anchor security. The configuration is defined by a security profile, which you specify when you run the utility. (Currently, FlexNet Embedded has only one security profile available, called `xt-medium`, which implements medium-level secure anchoring.)

To display the list of available security profiles, use the following command:

```
secureprofileutil -profilelist
```

Enabling Secure Anchoring

Once you have determined which security profile to use, run `secureprofileutil` against your producer client-identity data with a specific security profile designated. The following shows the command syntax:

```
secureprofileutil -profile profilename input_clientidentityfile.bin output_clientidentityfile.bin
```

The arguments include:

- `-profile` *`profilename`*: Name of the security profile used to implement secure anchoring. (Currently, only the `xt-medium` security profile is available.)
- *`input_clientIdentityFile`*`.bin`: Name of the file containing the client-identity binary data obtained from FlexNet Operations or generated using `pubidutil`.
- *`output_clientIdentityFile`*`.bin`: Name you want to give the output file containing client-identity binary data configured for secure anchoring.

Once you have configured the client-identity binary file for secure anchoring, run the Print Binary Utility on the data to format it for compatibility with your license-enabled code. (To ensure that secure anchoring is enabled, check the `printbin` output; it will include an `AnchorConfiguration` element if secure anchoring is enabled.)

.NET XT Toolbox

The .NET XT Toolbox is a pre-built executable with which you can explore various licensing processes and scenarios. To run the .NET Toolbox, launch `bin\demo\toolbox\DotNetDemo.exe`.

Preparing the .NET XT Toolbox

Before testing any licensing scenarios, you must prepare the .NET XT Toolbox by providing client-identity data and specifying the hostid to use for license acquisition. After launching the executable, click the browse button next to the **Identity File** field and browse for your `IdentityClient.bin` file. You might have created this file with the Publisher Identity Utility, or used FlexNet Operations On Demand, FlexNet Operations, or another back-office server.

You can also specify a location where the .NET XT Toolbox's trusted storage files will be written. The default behavior is to use the same directory as the .NET XT Toolbox executable. (This enables multiple instances of the .NET XT Toolbox on the same system to keep their settings and license rights separate. To run the .NET XT Toolbox from a different directory, copy `DotNetDemo.exe`, `FlxDotNetClient.dll`, and `FlxCore.dll` to the new directory.)

When finished, click **Set**.

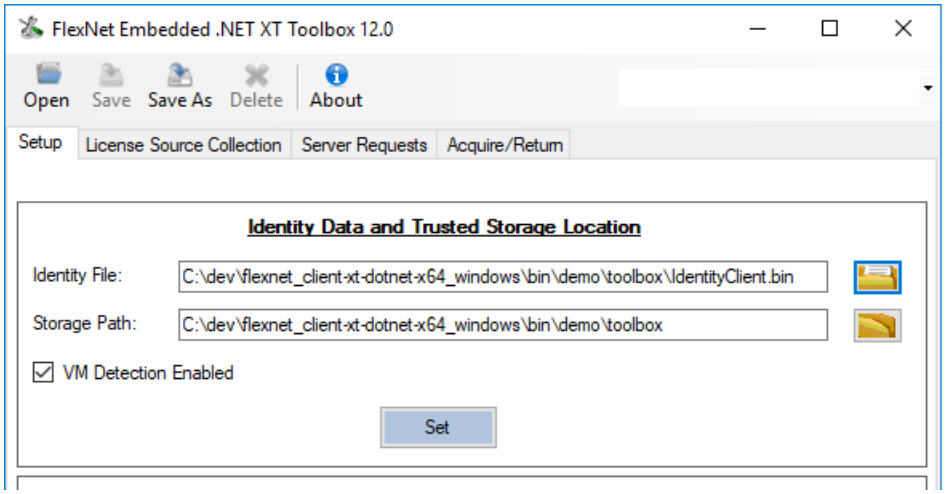


Figure 8-2: Specifying Client-Identity Data

Note • *As previously described, for security reasons an application intended for use in a customer environment should not read the identity data in "IdentityClient.bin" at run time, but should instead compile the identity data from "IdentityClient.cs".*

Next, specify the hostid that the .NET XT Toolbox should use for identifying itself in server communications and during license acquisition. The default is to use the string hostid **1234567890**, as used in the other toolkit examples, but you can change this to a different type if desired. If **VM Detection Enabled** is selected (default), the environment (virtual or physical) in which the client is running in is taken into account for applicable licensing operations. When finished, click **Set**.

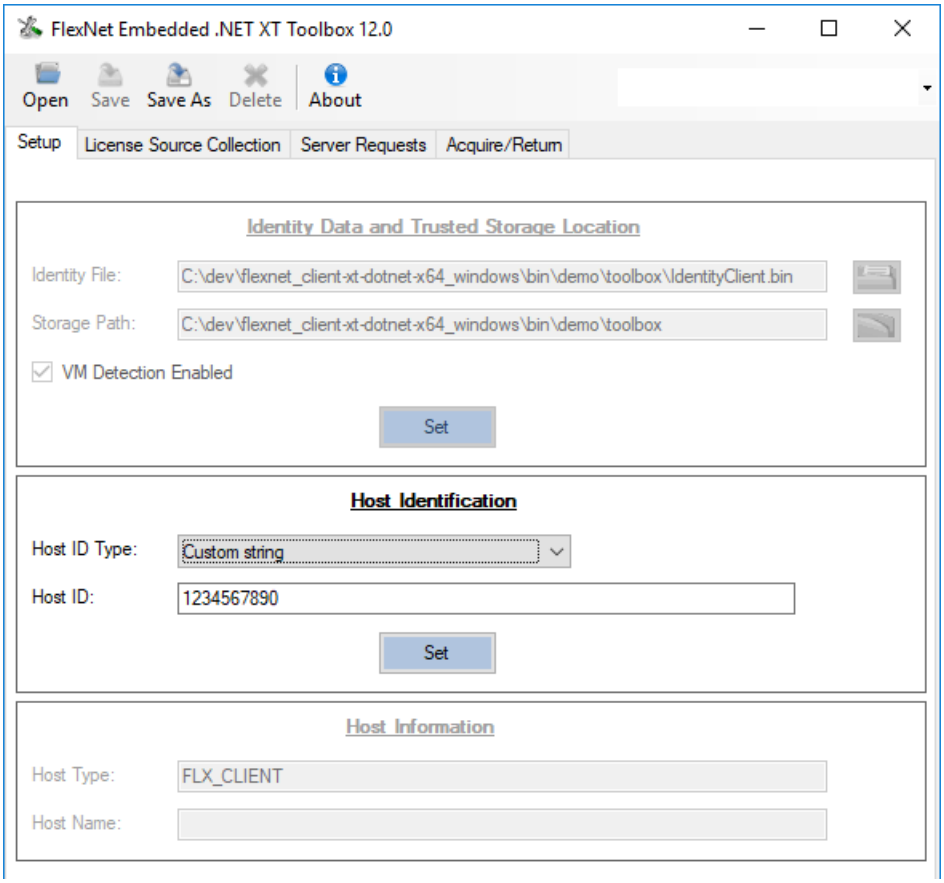


Figure 8-3: Specifying the Client Hostid Type and Value

If desired, you can also change the **Host Type** and **Host Name** values, which are included in server communications. To continue, you can click the **Continue** button at the bottom of the window, or select one of the tabs labeled **License Source Collection**, **Server Requests**, or **Acquire/Return**.

To avoid having to re-enter these settings on a subsequent launch of the .NET XT Toolbox, you can save a collection of settings into an XML file by clicking the **Save As** button at the top of the window and entering the file name and a name for the collection of settings. You can later open this configuration using the **Open** button.

Working with License Sources

The **License Source Collection** tab is where you define the types of license sources the .NET XT Toolbox will work with—any combination of buffer licenses, trusted storage, and trials—and displays the features found in the collection of sources.

For example, if you click the browse button next to the **File** field at the bottom of the window, browse for a binary capability response, and click **Load File**, the **License Source Collection** tab will appear similar to the following.

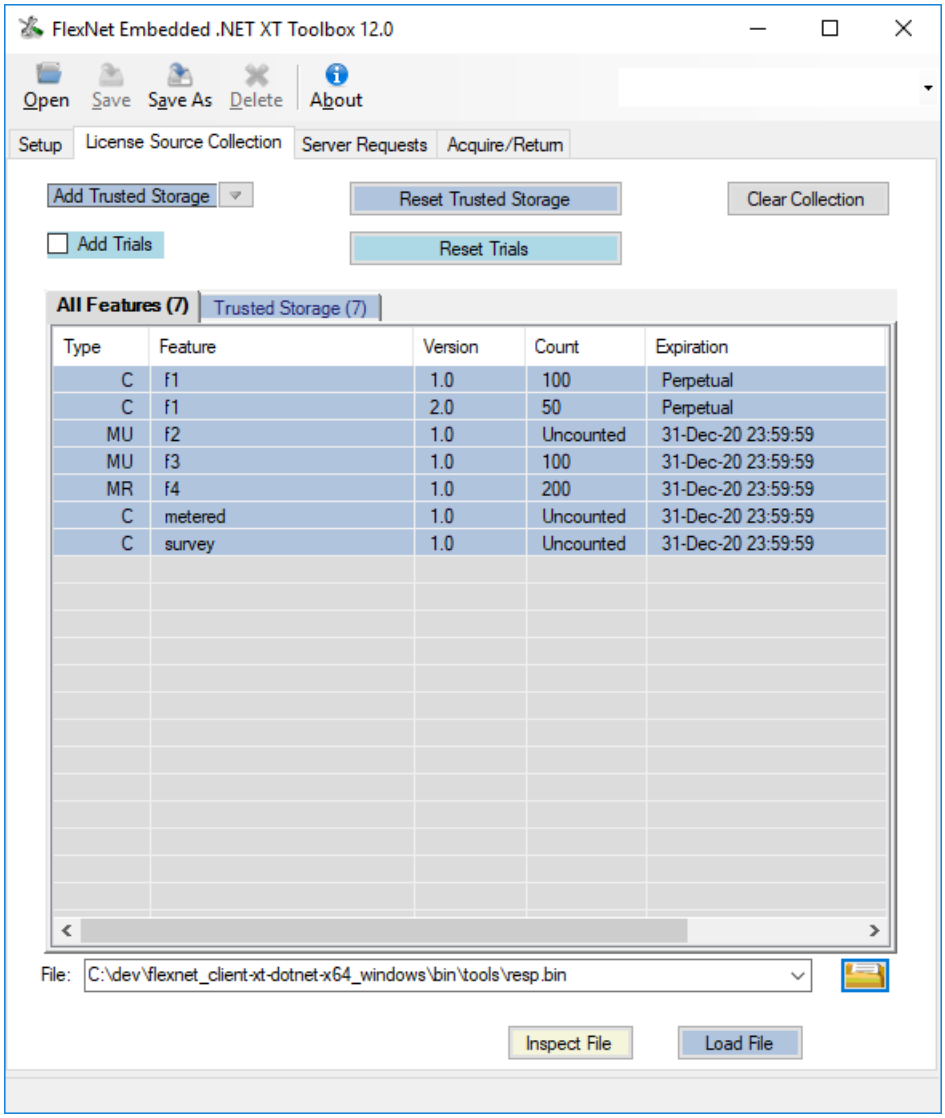


Figure 8-4: Contents of a License Source Collection

The **All Features** tab shows the features found in all the license sources, and the tab for each individual source shows features specific to that source. Browsing for and loading a binary trial file will add a tab for the trials license source, and add the trial's features to the **All Features** tab.

The information shown for a given feature on any of these tabs includes the feature's version, count, and expiration date and any vendor string defined. The **Type** column identifies the feature as either concurrent (**C**), metered (**M**), metered reusable (**MR**), or metered undoable (**MU**). (Metered features are found in trusted-storage license sources only.)

Move your mouse over a feature to display a tooltip containing this same information, along with other information defined for the feature, such as the feature's issued and start dates or its "undo interval". To view a complete list of details for the feature, double-click its entry to open the Feature Properties window. Additional details in the window might include (depending on the feature's type and definition) the current number of available licenses, the amount of "undo interval" time remaining, and other helpful information.

You can reset individual license sources by clicking the appropriate **Reset** button, or reset all of them by clicking **Clear Collection**.

Server Communications

The Server Requests tab enables you to perform direct or offline communication with a back-office server (either FlexNet Operations or a test back-office server, the latter being used in combination with the Capability Server Utility) or a license server.

In the **Server Information** section, enter the location of the server host. Changing the server type in the **Type** list changes the **URL** and **HTTPS** fields to typical default values for the selected server type, and you can override a setting—such as a custom port number—by typing in the appropriate value. Click **Advanced** if you want to configure a proxy server and provide custom HTTP headers.

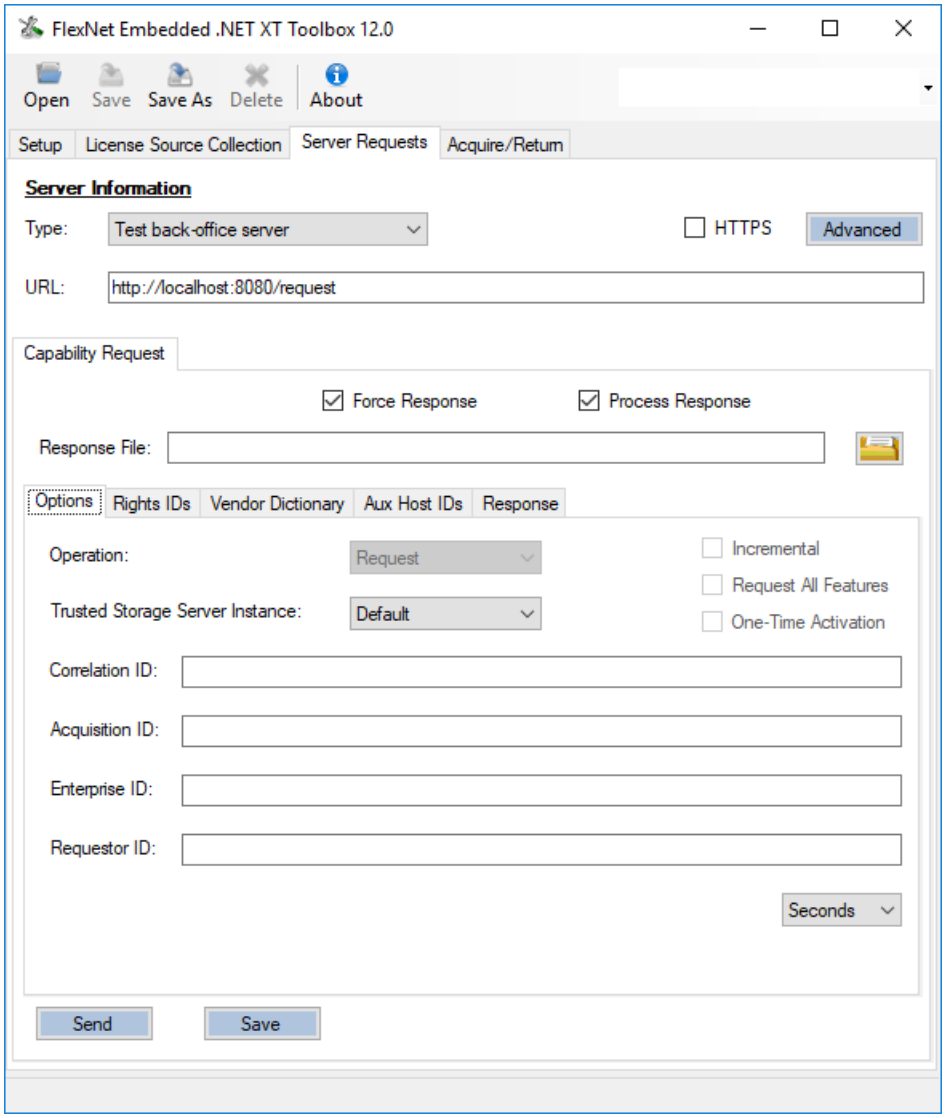


Figure 8-5: Preparing Communications with a Server

In the middle of the **Server Requests** tab, you can specify settings for various flags and behavior related to the capability request that the .NET XT Toolbox sends to the server, such as whether to set the force-response flag or process the response sent from the server.

Information in the bottom section of the **Server Requests** tab is enabled depending on the server type. For a back-office server (as pictured), you can optionally specify various IDs, including one or more rights ID values, to send in the request, along with any vendor dictionary items to include. For a license server (not pictured), you can specify "desired features" (but no rights IDs) and other options in the request.

When you have the desired settings, click **Send** at the bottom of the window to communicate directly with the specified server, or click **Save** to save the capability request message as a file.

If the communication is successful, the response details are available in the **Response** tab. (You can use the **Response File** field to save the response for later processing or examination.) The **Response Details** button in the **Response** tab displays further information about the server's response.

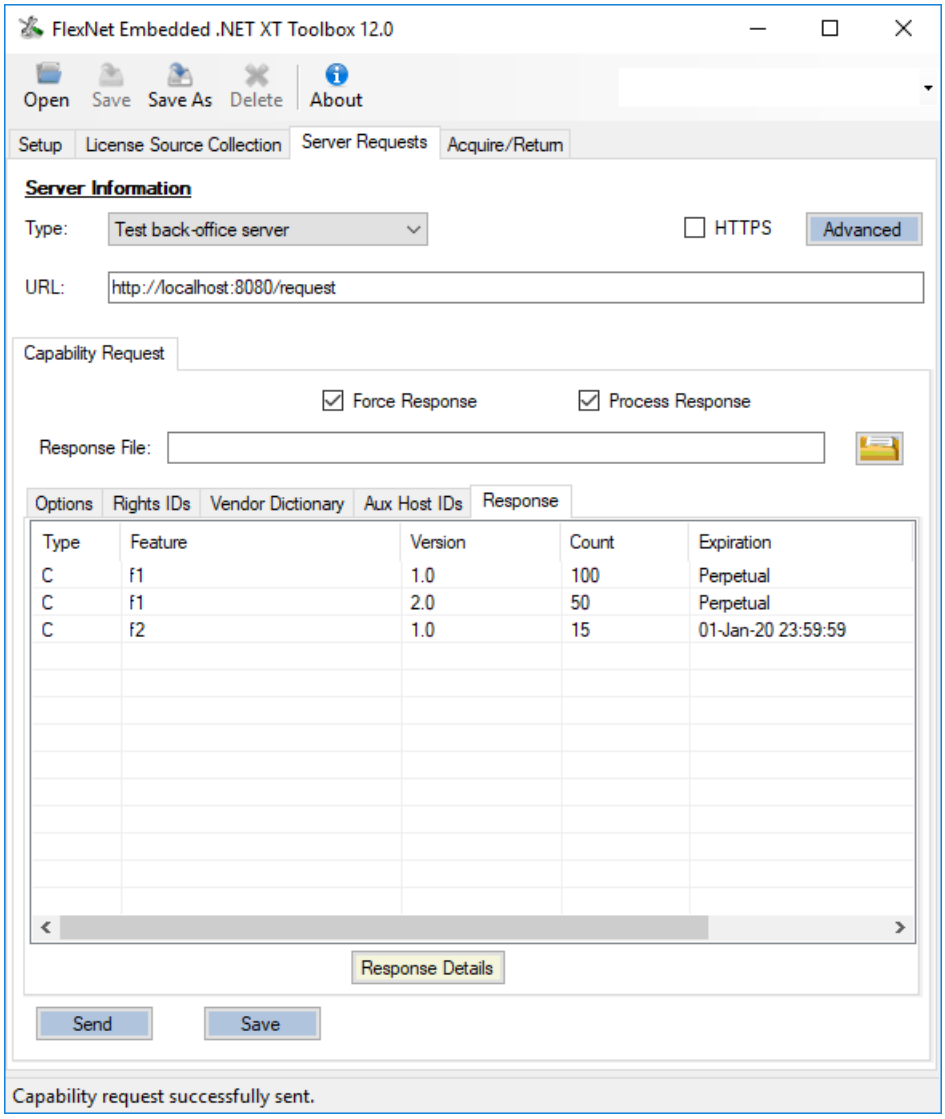


Figure 8-6: Viewing Features Contained in a Server Response

If you specified to process the server's response, the valid features loaded by the response will also be displayed on the **License Source Collection** tab (if **Add Trusted Storage** is selected).

License Acquisition

In the **Acquire/Return** tab, you can attempt to acquire licenses from the license source collection. The **Feature** list contains an entry that you can select for each feature in the license source collection, or you can enter a different name. You also specify the version and count of the feature you want to acquire.

When you have entered the characteristics of the feature you want to acquire, click **Acquire**. If the acquisition attempt succeeds, feature information is displayed in the **Acquired Licenses** table.

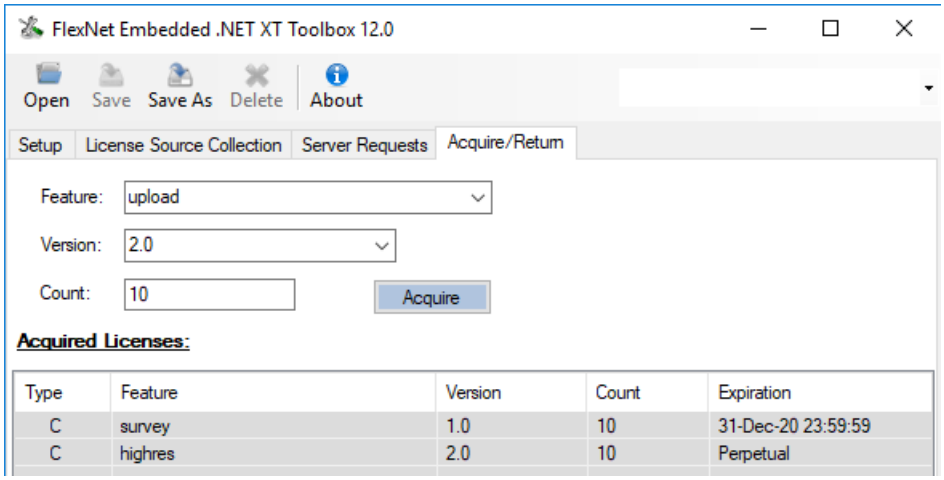


Figure 8-7: Acquiring a License from the License Source Collection

Click **Return Selected** or **Return All** (not pictured) to return licenses that you have acquired.

If an attempt to acquire a license fails, an icon is displayed next to the **Acquire** button, with a tooltip explaining the reason for the failure. The error message is also displayed in the status area at the bottom of the **Acquire/Return** window.

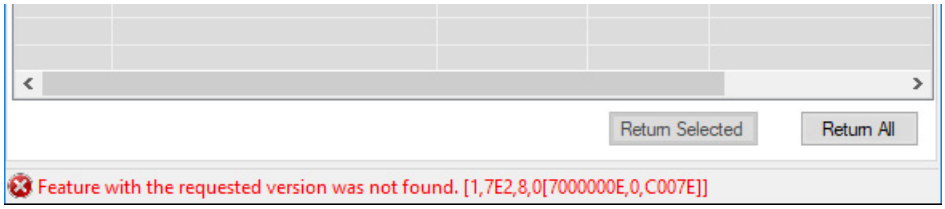


Figure 8-8: Error Message for a Failed Acquisition Attempt

A

Manifest File Contents for a Product Update

The Updates and Insights notification server generates a manifest file as a means to download the one or more files needed to install a single product update successfully. The manifest file contains the list of files (along with other relevant information such as file locations) that are downloaded and, if necessary, executed to complete the update installation.

The Update and Insights client differentiates a manifest file from other types of notification items by its content type value of 11.

The following describes a manifest file:

- Manifest File Format
- Manifest File Setup Rules
- Manifest File Processing Rules

Manifest File Format

A manifest file is a text file consisting of a header, followed by set of lines, each line an entry describing a file to download. The file can also include comment lines that are denoted a semicolon (;) or a # as the first character in the line. The text file is saved with a `.mfdl` extension.

The following shows the contents of a sample manifest file:

```
1.0
#Example of executable file where md5 hash is verified
https://download-uat.flexnetoperations.com/439215/ci_1000/947/7016947/
InstallThis.exe;10752;md5;b15120271715c4e97ca6b7f7f64fd94c;yes
#Example of non-executable file where md5 hash is verified
https://download-uat.flexnetoperations.com/439215/ci_1000/407/7016407/
success.jpg;49578;md5;70f92964866f86002f35a1956134d904;no
#Example of executable file where no hash is verified
https://download-uat.flexnetoperations.com/439215/ci_1000/407/7016427/
RunConfigUtil.exe;8192;;;yes;run_all
#Example of non-executable file where no hash is verified
https://download-uat.flexnetoperations.com/439215/ci_1000/407/7016400/config.txt;8192;;;;
```

Header Line

The first line in the manifest file must be the header line, which lists the format version of the manifest file. (Contact Revenera for the available manifest-file format versions.) This line is required and must be the first line in the file, separate from the other lines. For example, the following is the header line for the 1.0 format version:

```
1.0
```

File Entries

The header line is followed by one or more lines, or *manifest entries*. Each entry consists of a set of fields that describe a single file to be downloaded. The following shows the general format of an entry:

```
Filename;FileSize;HashCryptoType;HashValue;Execute;ExecuteCmdLine
```

The fields used in an entry are described in the following table. Only the *Filename* and *FileSize* are required fields.

Table A-1 • Fields for Each Entry in the Manifest File

Field	Description
URL from which to download file	The URL for the file to be downloaded. This value must be either: • The explicit URL for the file. Supported URL types include `http://`, `https://`, `ftp://`, and `ftps://`. • The file and its path relative to a location determined by the producer or Revenera.
FileSize	File size in bytes.
HashCryptoType	(Optional) Cryptographic hash type. This value is either `sha1` or `md5`. If no value is provided, the default value md5 is used. See the following *HashValue* description for more information.
HashValue	(Required if a HashCryptoType value is provided) The MD5 or SHA1 hash in hexadecimal notation. This value is used to verify the file's integrity. The Updates and Insights client uses the general `FlxCrypto` library (located in *`install_dir`*`\lib`) with which your application is built to run a hash algorithm on the file downloaded for this entry. The resulting hash value is compared to the value specified here to verify the file's integrity.
Execute	(Optional) The `yes` value to indicate that the file is to be executed. By default, the file is not executed, so entering the value `no` is not necessary.
ExecuteCmdLine	(Optional) Execution command line options for this manifest entry. These take effect only if the execution field is set to `yes`.

Manifest File Setup Rules

The Updates and Insights notification server follows these rules when creating the manifest file:

- Lists files in the order in which the Updates and Insights client must process them.
- Ensures that the specified files are downloadable.
- Uses a semicolon (;) or # at the beginning of comment lines only. Lines that begin with these characters are ignored during processing.
- Saves the manifest file as a text file with the `.mfdl` extension.

Manifest File Processing Rules

The Updates and Insights client uses the following rules to process the manifest file:

- Ignores lines that start with a space or tab.
- Treats trailing semicolons on manifest file entries as optional.
- Processes entries in the manifest file in sequential order.
- Encodes manifest files in UTF-8.
- If the *Execute* field is set to **yes** and command-line options (`ExexcuteCmdLine`) are set in the entry, uses the command line options to execute the specified file.
- If the *Execute* field is set to **yes** and no command-line options (`ExexcuteCmdLine`) are set in the entry, but the notification item for the manifest file defines command-line options, uses the options in the notification item to execute the file.
- If the *Execute* field is not set (or set explicitly to **no**), ignores any command-line options (either in the manifest item or the notification item) when executing the file.
- Interprets a line in the manifest file that begins with a semicolon (;) or # as a comment line, and ignores the line.
- Generates an error if an entry does not point to a downloadable file. For example, an entry with a blank directory results in an error.
- Generates an error if an entry that contains incorrectly formatted file names. For example, embedded tabs in a file name entry cause an error.
- Downloads a file of a special type, such as a compressed file, as is. For example, if a manifest entry points to a `.zip` file, the file is downloaded as a `.zip` file in its entirety.

Index

A

acquisition ID, in capability request to send feature-usage data 106
anchoring in trusted storage 44
APIs in FlexNet Embedded .NET XT
  overview 65
  primary interfaces 61
  reference guide 62
APIs in FlexNet Embedded Client .NET XT
  error handling 63
  primary groups 63
APIs in Updates and Insights
  groups 62
  implementation walkthroughs 123

B

back-office identity
  using pubidutil or back-office server to generate 16, 138
back-office servers
  description 43, 76
Base 64 license format 142
basic_client example
  building 20, 33
BasicClient example
  API walkthrough 73
  overview 49
  running 22
binary signed license file, generating 74
binary-file contents
  validation with printbin 141
  viewing 141
buffer licenses implementation, see Client example

C

callback function (example) to monitor update download and execution progress 130
capability request
  creating 78
  example 76
  generating with .NET XT Toolbox 161
  generating with caprequestutil 151
  operation types 105
  processing response 81
  sending to back-office server 80
  used for usage capture 105
capability response
  APIs used to process 81
  generating with capresponseutil 154
  processing with .NET XT Toolbox 162
CapabilityRequest example
  API walkthrough 76
  overview 49
capabilityrequest example
  running 80
caprequestutil, using to generate capability request 151
capresponseutil, using to create capability response 154
certificate-based licensing, implementing functionality for
  about the signed certificate 117, 118
  acquiring features 118
  comparison with FlexNet Publisher certificate-licensing 119
  creating the license source 118
  overview 117
  preparing identity data 117
Client example
  overview 49
client identity data
  in usage-capture scenarios 107

using FlexNet Operations to generate 16, 107
using printbin to convert into C# code 18, 32, 66, 140
using pubidutil or back-office server to generate 16, 138
client-server identity, using pubidutil or back-office server to generate 16
client-server identity, using pubidutil to generate 138
clock-windback detection
APIs called 70
description 70
Cloud Licensing Service, used to monitor feature usage for metered licenses 108
collection (notification) object, creating 128
containerized environment, detecting 69
core Updates and Insights objects, creating
client object (IUAIClient) 125
product package object 126
correlation ID
assigning 106
used to undo a feature-usage capture 106, 111

D

device alias 80
diagnostic API 76
See also View example

E

examples
instrumentation 51, 53, 54
notification 23, 25, 50, 127
profile 51, 53, 54
register 51, 53, 54
expiration date in feature definition 42

F

feature definition syntax 41
feature-usage capture, implementing functionality for
about capability requests 105
about correlation IDs 106
capture data for capped usage 110
capture data for uncapped usage 109
creating trusted-storage license source 107
overview 104
preparing FlexNet Operations 107
using rights ID to register client on Cloud Licensing Service 108
FlexNet Embedded .NET XT API reference 62
FlexNet Embedded Client .NET XT toolkit
contents 46
overview 46
FlexNet Operations
configured to receive usage data for metered licenses 107
used to generate producer identity data 16
used to generate publisher identity data 140
used to generate publisher identity information 107
used to re-host licenses 101
frequency, clock windback 70

H

HOSTID keyword in feature definition 42
hostids, description 40

I

identity data
creating 16, 30
distributing 18, 32
updating 19, 32
identity files 138
IdentityBackOffice.bin 124, 138
using pubidutil or back-office server to generate 66
IdentityClient.bin 66, 124, 138
IdentityClientServer.bin 66, 138
INCREMENT syntax 41
instrumentation example
building 51, 54
running 53
IUAIClient (client object), creating 125

L

license conversion utility, see licensefileutil, using to generate license binary file
license source collection, creating and populating 75, 77
license-file contents
signature validation with printbin 141
viewing 141
licensefileutil, using to generate license binary file 74, 146
license-rights examination
creating diagnostic license source 113
overview 112
viewing details in feature collection 114
licenses
acquiring (implementing APIs) 75
acquiring (using .NET XT Toolbox) 163
generating binary signed 74
manually creating license file (text) 74
obtained from back-office server 76
providing access to 77
reading details 76
using on the device 74
Lmflex example
API walkthrough 117

running 118

M

manifest file
contents 165
downloading 130
metered licenses
capturing capped usage 110
capturing uncapped usage 109
license attributes 111
See also feature-usage capture, implementing functionality for

N

notification collection object, creating 128
notification example
overview 23, 50
running 25
walkthrough 127
notification items
definition 46
notifications items
obtaining (example code). See also notification example 128

O

Overdraft attribute for metered licenses 112

P

prerequisites for FlexNet Embedded Client .NET XT 13
print binary utility, see printbin, using to convert and display contents of binary data
printbin 141
printbin, using to convert and display contents of binary data
client identity data 18, 32, 66, 67, 140
license-file conversion to Base 64 142
producer identity data
using pubidutil or back-office server to generate 16
product package, associating to Updates and Insights client (IUAIClient) 126
product updates, downloading and executing (example code). See also notification example 130
profile example
building 51, 54
running 53
pubidutil, using to generate producer identity binary data
command and arguments 138
graphical UI 138
publisher identity data
using pubidutil or back-office server to generate 138
Publisher Identity utility
generating identity files 16, 30
Publisher Identity utility, see pubidutil, using to generate producer identity data

R

recalling metered features 111
register example
building 51, 54
running 53
rehosting licenses 99
request, see capability request
requestor ID, in capability request to send feature-usage data 106
response status items 81
rights ID 152
in capability request (.NET XT Toolbox) 162
in capability response 81
in usage-capture scenarios 107
runtime acquisition 82

S

secure re-hosting 99
secureprofileutil, using to enable secure anchoring 156
server, back-office vs. local 43
signatures, validation with printbin 141
START keyword in feature definition 43

T

test back-office server, see Test back-office server
tolerance, clock windback 70
trialfileutil, using to generate signed trial rights 146
Trials example
API walkthrough 97
overview 50
trials, implementing functionality for
acquiring license rights 99
creating license source 98
creating trial license-rights file 97
processing trial data into license source 98
trusted storage
anchoring 44
description 44
specifying location for 67
used to handle metered licenses 104
used to store licenses from back-office server or license server 76

U

uncounted features 76
Undo Interval attribute for metered licenses 112
undoing feature-usage capture 111
unsigned license file, creating 74, 77
updates (product), downloading and executing (example code).
    See also notification example 130
Updates and Insights
    API implementation walkthroughs 123
    running example projects 53
Updates and Insights client, definition 45
Updates and Insights notification server
    creating identity file for (IdentityBackOffice.bin) 16, 30
    sending identity to 18, 32
Updates and Insights objects, creating
    client object (IUAIClient) 125
    notification collection 128
    product package object 126
UsageCaptureClient example
    API walkthrough 104
    overview 50
    running 108, 109, 110
utilities included in FlexNet Embedded Client .NET XT toolkit 137

V

vendor dictionary 72
VENDOR_STRING keyword in feature definition 43
View example
    API walkthrough 112
    overview 50