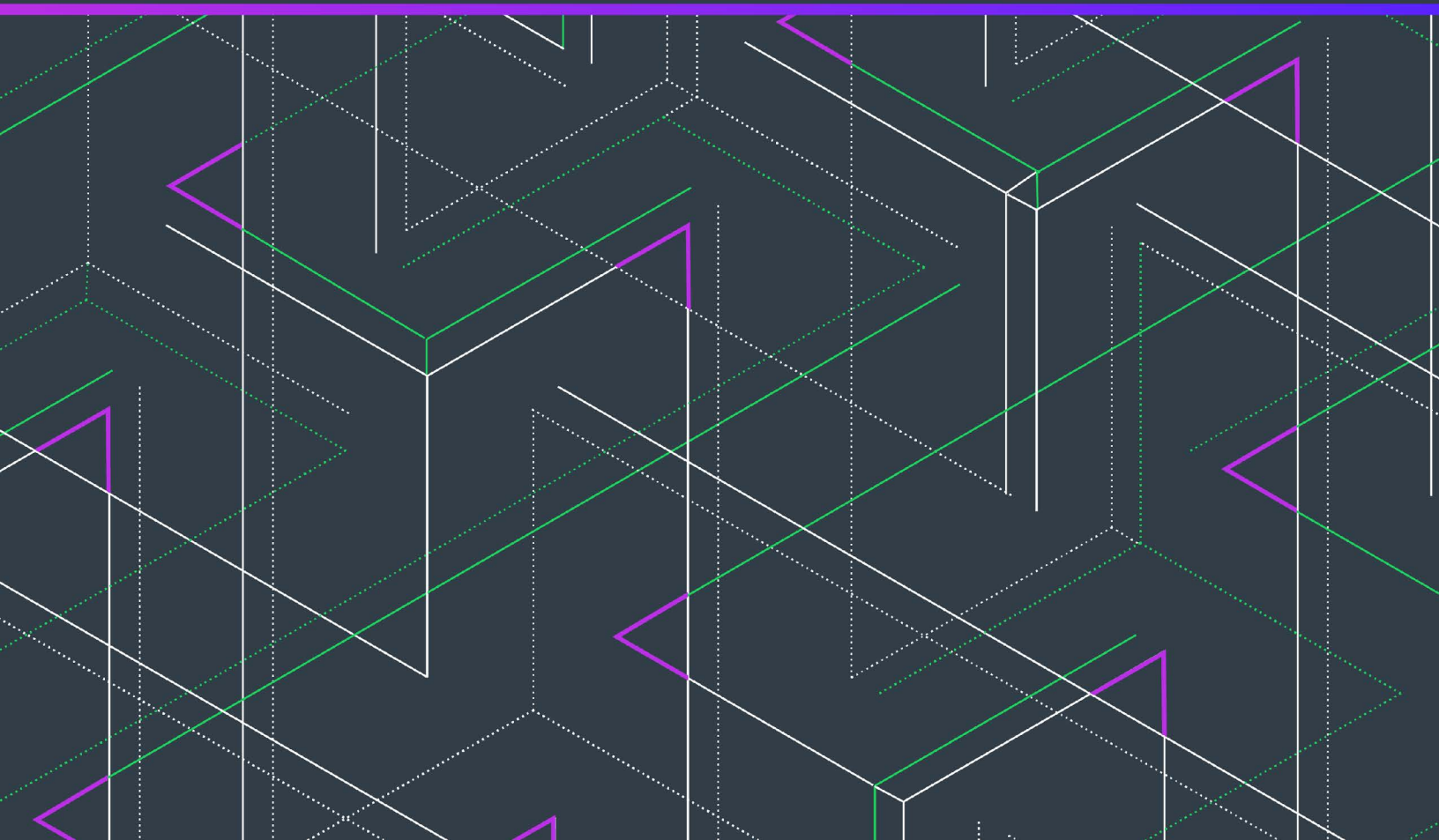
revenera

FlexNet Embedded Client 2024.11

Java XT SDK User Guide



Legal Information

Book Name:	FlexNet Embedded Client 2024.11 Java XT SDK User Guide
Part Number:	FNE2024.11-JXTSDK-UG00
Product Release Date:	November 2024
Documentation Last Updated:	June 26, 2024

Copyright Notice

Intellectual Property

Restricted Rights Legend

Contents

1

About the FlexNet Embedded Client Java XT Toolkit and this Guide

The FlexNet Embedded Client Java XT toolkit offers implementers a source of APIs and tools to instrument a secure licensing framework in which software producers can control the product features to which end users are entitled on client systems. This set of APIs and tools—that is, FlexNet Embedded—enables producers to offer different product configurations, enforce node-locked licensing, and enable hands-free activation, silent trials, and electronic (field) upgrades.

This chapter contains the following information:

User Guide Overview

The purpose of this user guide is to provide an overview of the FlexNet Embedded Client Java XT software product toolkit to help you start creating code. The guide includes the following chapters:.

Table 1-1 ▪ Overview of the *FlexNet Embedded Client Java XT SDK User Guide*

Topic	Content
Quick Start with the Java XT Toolkit	Provides an introductory walkthrough of preparing the FlexNet Embedded Client Java XT toolkit, and then building and running example code that performs basic licensing. The chapter is geared toward potential customers or current customers who want a simple “getting started” demonstration to see how the licensing works.

Table 1-1 ▪ Overview of the *FlexNet Embedded Client Java XT SDK User Guide* (cont.)

Topic	Content
Toolkit Overview	Provides an overview of the FlexNet Embedded Client Java XT toolkit, describing: ● Terminology used for FlexNet Embedded licensing services ● Toolkit contents ● Example projects included with the toolkit ● Build instructions for the toolkit examples
Overview of the Java XT APIs	Describes the groups of FlexNet Embedded and common APIs included in the toolkit.
Using the FlexNet Embedded APIs	Describes the primary objects included in the FlexNet Embedded Client Java 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
Utility Reference	Explains how to use the toolkit utilities and the test back-office server `capserverutil` (also called the Capability Server utility) to test and prepare your FlexNet Embedded client application for production.

Product Support Resources

The following resources are available to assist you:

- Revenera Product Documentation
- Revenera Community
- Revenera Learning Center
- Revenera Support

Revenera Product Documentation

You can find documentation for all Revenera products on the Revenera Product Documentation site:

https://docs.revenera.com

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 first logging into the Revenera Community and then making selections on the **Get Support** menu, including **Open New Case** and other options.

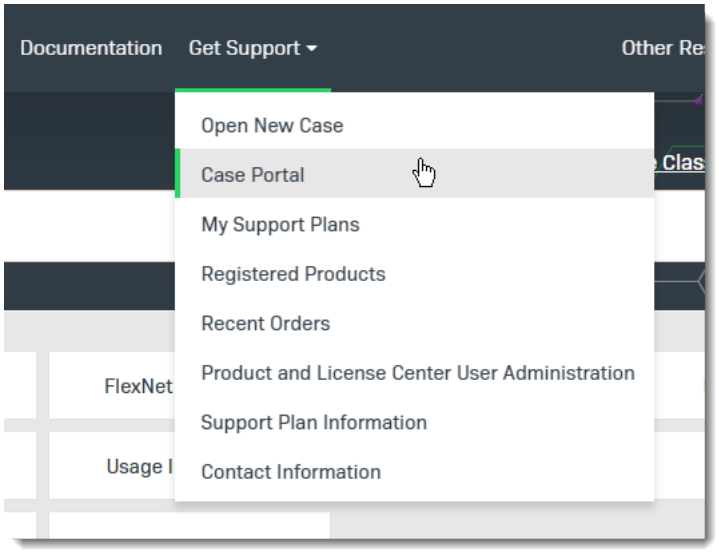


Figure 1-1: Get Support Menu of Revenera Community

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 Java XT Toolkit

This chapter describes quick basics to get started with the FlexNet Embedded Client Java XT toolkit:

- Toolkit Requirements
- Downloading the Toolkit
- Creating the Producer Identity
- Building and Running the “Basic Client” 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 Java 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` or `.tgz` 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 Java 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 Java 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 Java XT 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 rights sent to the client. You must then 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 Java 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 Java-compatible Identity Data
- Distributing Identity Data
- Updating Your Publisher Keys in Identity Data
- Multiple Signature Strengths for Standalone Devices

Creating the Identity Binary Files

To create the producer-identity binary files, you need the organization-specific production keys included in your email from Revenera and the `pubidutil` utility provided in the FlexNet Embedded Client Java XT toolkit.

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

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

- **Back-Office Identity File**—Location of your back-office identity file. Used to update existing identity data or to add a signature strength.
- 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.

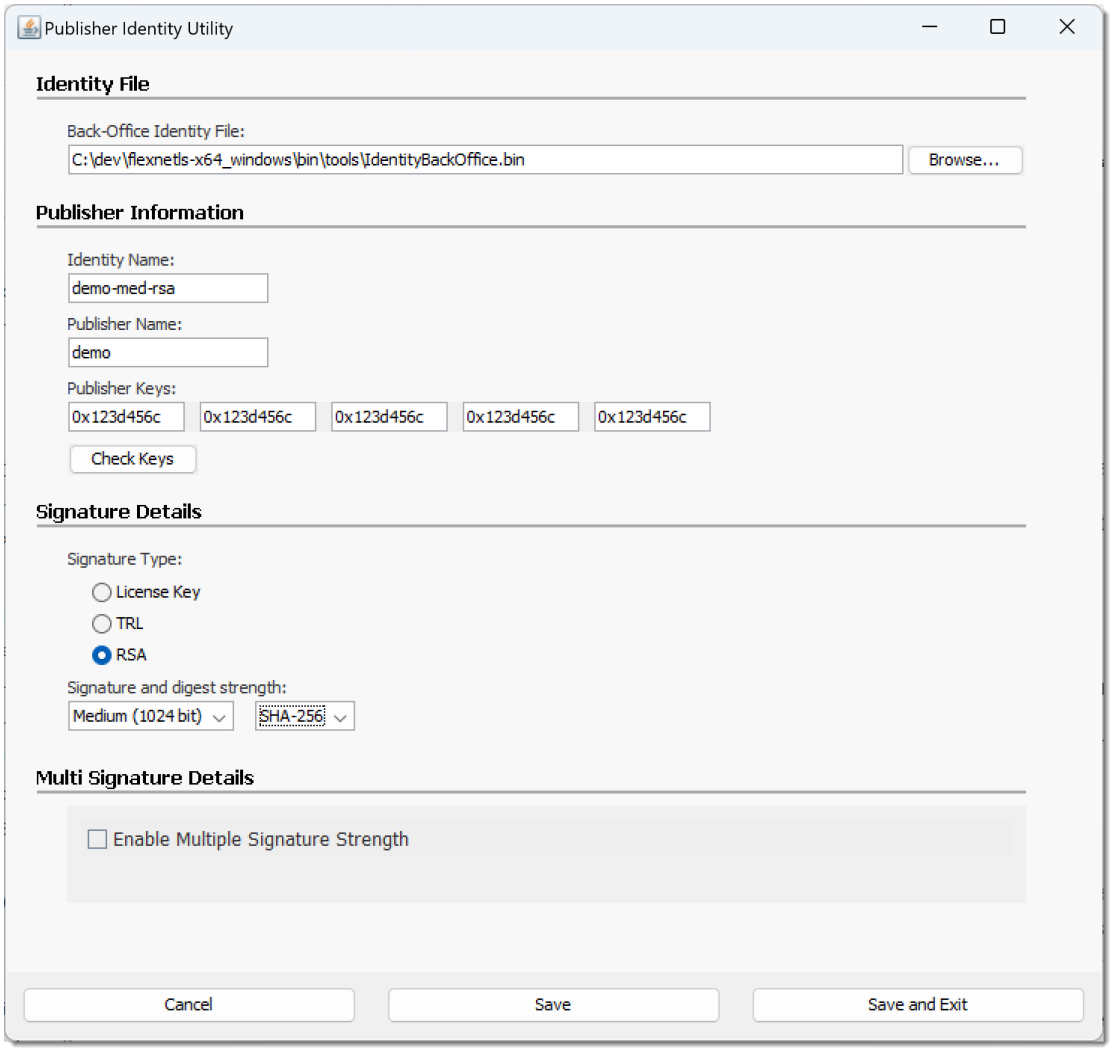


Figure 2-1: Entering Producer-specific Information in the Publisher Identity Utility

Click **Save** or **Save and Exit** when you have entered this information. `pubidutil` prompts you to specify a folder. Upon confirming the folder location, `pubidutil` creates a subfolder with a file name in the format `YYYY-MM-DDThh-mm-ss-sss` (for example, `2024-06-12T09-30-24-064`), which will hold the new set of identity files:

- 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).

- 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 Your Publisher Keys in Identity Data.

Generating Java-compatible Identity Data

The Publisher Identity utility generates your identity files in binary format. FlexNet Embedded Client Java XT 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 Java XT toolkit.

To generate a Java array of bytes for use in Java code, run the following command:

```
printbin -java IdentityClient.bin -package flxexamples -o IdentityClient.java
```

Note that the -package flxexamples option applies only to the toolkit examples, which use the flxexamples package name.

The output should look similar to the following:

```
package flxexamples;

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

public class IdentityClient {
    public static final byte[] IDENTITY_DATA = {
        -104,   97,  112, -112,  121,   32,  108,  105,   99, -101,
        -110, -115, -105, -110,  103,  -33,   32,   45,  114, -111,
    ...
         -98,  101,  114, -116,  100,    0 };
}
```

Distributing Identity Data

To prepare the client-identity data for use in the FlexNet Embedded Client Java XT code, copy the file IdentityClient.java into a directory where it can be accessed by and compiled into your executable code. (To run the FlexNet Embedded Client Java XT examples, copy the identity data to the *install_dir*/examples/client_examples/src/flxexamples 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 the appropriate location. 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.

For information about using the FlexNet Operations Producer Portal to upload the back-office identity, refer to the Portal's online help system. For information about using capserverutil, see Capability Server Utility in the *Utility Reference* chapter.

Updating Your Publisher Keys in 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`.

GUI Mode

To update your publisher keys in identity data using `pubidutil` in UI mode, enter or browse to the existing back-office identity file in the **Back-Office Identity File** field. The Publisher Identity utility will automatically populate the remaining fields. Make sure that you use all original settings (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.

Click **Save** or **Save and Exit** and specify the folder that will hold the subfolder (with a file name in the format `YYYY-MM-DDThh-mm-ss-sss`, for example, `2024-06-12T09-30-24-064`) which will then contain the updated set of identity files.

Console Mode

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 Java-compatible Identity Data and Distributing Identity Data to update client and server components with the new identity data.

Multiple Signature Strengths for Standalone Devices

With technology and computational power evolving rapidly, algorithms and key sizes that are considered secure today may be considered vulnerable in the future. Therefore, software producers may decide that they would like to migrate to a different signature strength.

To enable migration to a different signature strength, the Publisher Identity utility allows you to add one additional signature strength to the already existing signature strength. Currently, only standalone devices support two signature strengths.

This topic is grouped into the following sections:

Client Behavior for Validating Messages

When you add an additional signature strength to an identity, existing clients will continue to validate any downstream messages, such as capability responses, using the original signature strength. Any new clients that you build using the updated identity will validate downstream messages using the added signature strength.

The following diagrams visualize the behavior.

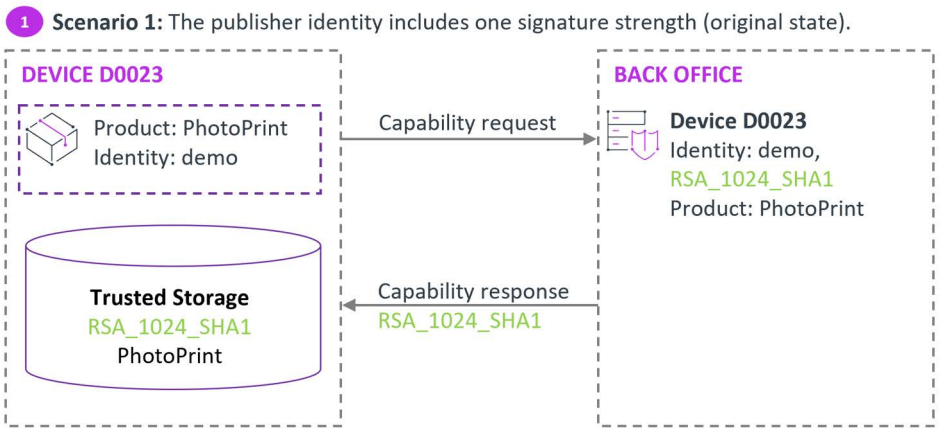


Figure 2-2: Initial scenario where the identity used for the client software and in the back office has only one signature strength.

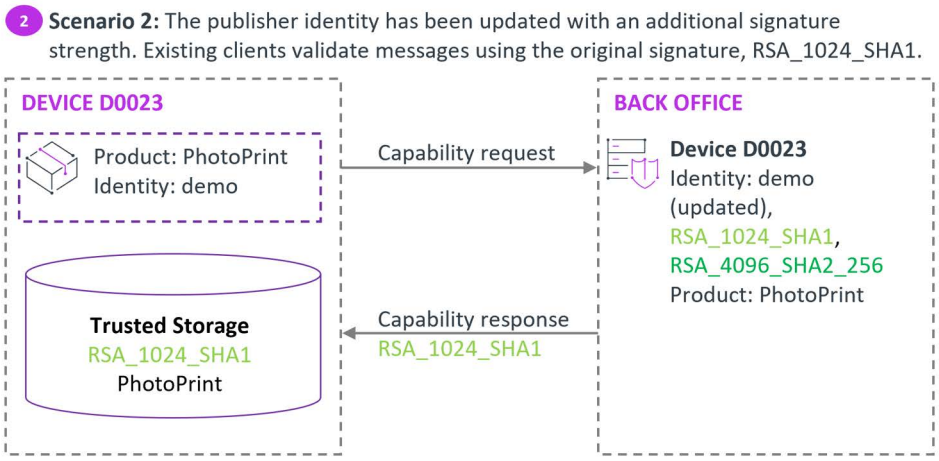


Figure 2-3: In scenario 2, the publisher identity in the back office has an additional signature strength. Existing devices running the client software that was built using the original, unchanged identity use the original signature strength to validate downstream messages.

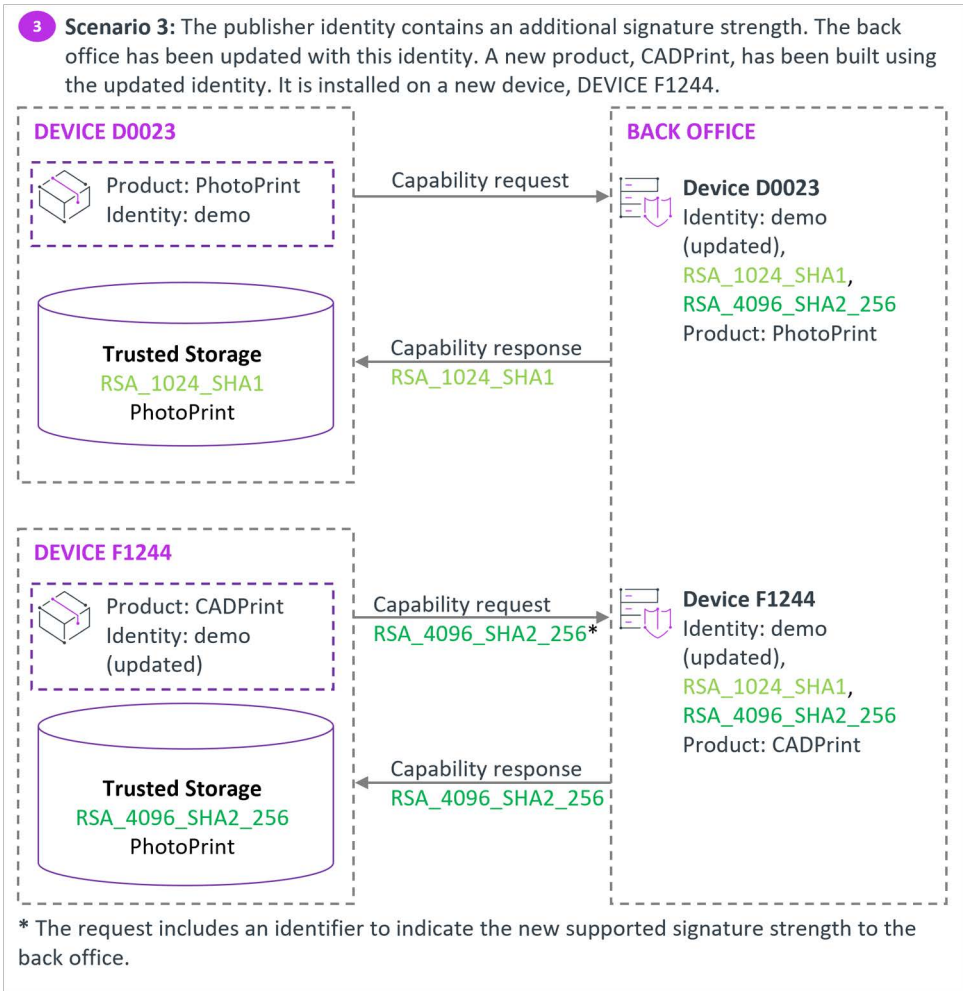


Figure 2-4: In scenario 3, the publisher has launched a new product which was built using the identity that includes the additional signature strength. This client will always use the added, higher signature strength to validate downstream messages.

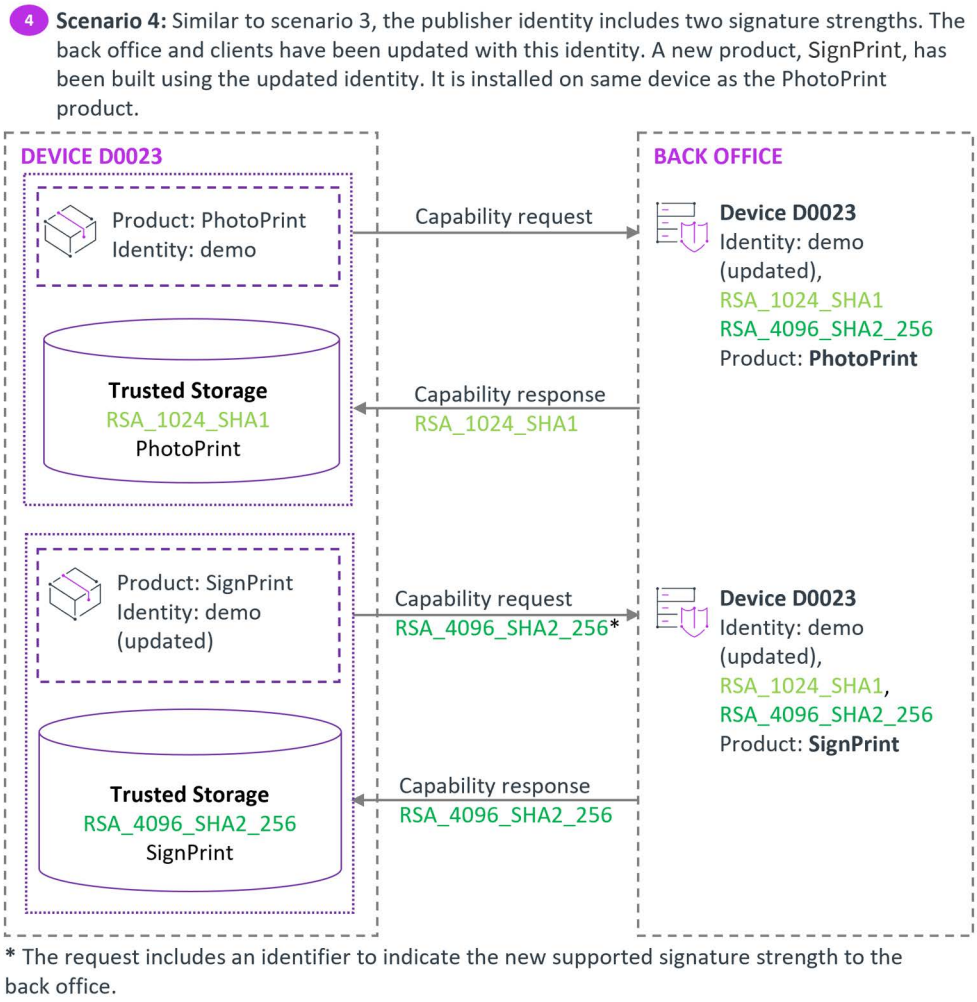


Figure 2-5: In scenario 4, the publisher has launched another new product which has also been built using the updated identity. It is installed on the same device as the PhotoPrint product. However, both clients use their own trusted storage. PhotoPrint will use the original signature strength, while SignPrint will use the higher signature strength to validate downstream messages.

Adding a Signature Strength

You can add an additional signature strength using Publisher Identity utility in GUI mode or in console mode.

Adding a Signature Strength in GUI Mode

This section describes how to add a signature strength in GUI mode.

Task ***To add a signature strength in GUI mode***

1. Open the Publisher Identity utility in GUI mode.
2. In the **Back-Office Identity File** field, enter the path and file name or browse to the back-office identity file to which you want to add a signature strength.

   The remaining fields will automatically be populated. Do not change any of the values.
3. In the section **Multi Signature Details**, click **Enable Multiple Signature Strengths** to display additional controls.
4. Click **Add Signature Strength**. The **Signature Details** window is displayed.
5. Select the desired signature type, and the signature and digest strength and click **OK**.
6. Click **Save** to save the updated set of identity files at a location of your choice or **Save and Exit** to save the updated identity files and close the Publisher Identity utility.

The Publisher Identity utility displays the added signature type and strength in the section **Multi Signature Details**.

All three identity files—`IdentityBackOffice.bin`, `IdentityClient.bin`, and `IdentityClientServer.bin`—are updated to include the old signature strength as well as the additional signature strength. They are saved in a folder with a file name in the format `YYYY-MM-DDThh-mm-ss-sss` (for example, `2024-06-12T09-30-24-064`).

Optionally, use the print-binary utility `printbin` to see the added signature strength in `IdentityBackOffice.bin`. You can validate the updated identity by clicking the **Validate** button.

Adding a Signature Strength in Console Mode

This section describes how to add a signature strength in console mode.

Task ***To add a signature strength in console mode***

1. Run the `pubidutil` script in the `bin/tools` directory of the toolkit.
2. Run the following command:

   ```
   pubidutil -silent [-backOffice backofficeidfile.bin] [-addSignatureStrength sig-type sig-
       strength]
   ```

   where `backofficeidfile.bin` is your back-office identity file. Replace *`sig-type`* and *`sig-strength`* with the desired signature type and signature strength, respectively, that you want to add.

   Example: The following command adds the signature type `RSA` with signature strength `4096 bit` and digest strength `SHA-512`.

   ```
   pubidutil.bat -silent -backOffice IdentityBackOffice.bin -addSignatureStrength RSA 35
   ```

   Tip ▪ *For information about signature strength values, see Signature Strengths.*

All three identity files are updated to include the old signature strength as well as the additional signature strength. They are saved in the same folder as the existing `IdentityBackOffice.bin` file that you specified on the command line.

Deleting the Additional Signature Strength

You can delete the additional signature strength, but not the original signature strength from a publisher identity.

Caution ▪ *Deleting an additional signature strength should be treated with extreme care and should only be done in a test environment but never with released products. It is not possible to add the additional signature strength back to an identity once it has been deleted. If you issue licenses with the additional signature strength, delete that additional signature strength, and then update existing clients with that new identity data then they can no longer validate already issued licenses in trusted storage that contain that signature strength.*

Deleting a Signature Strength in GUI Mode

This section describes how to delete a signature strength in GUI mode.

Task ***To delete the additional signature strength in GUI mode***

1. Open the Publisher Identity utility in GUI mode.
2. In the **Back-Office Identity File** field, enter the path and file name or browse to the back-office identity file to which you want to add a signature strength.

   The remaining fields will automatically be populated. Do not change any of the values.
3. In the section **Multi Signature Details**, click **Delete Signature Strength**.
4. In the **Delete Index** window, add the index **0** and click **OK**.

Deleting a Signature Strength in Console Mode

This section describes how to delete a signature strength in console mode.

Task ***To delete the additional signature strength in console mode***

1. Run the `pubidutil` script in the `bin/tools` directory of the toolkit.
2. Run the following command:

   ```
   pubidutil -silent [-backOffice backofficeidfile.bin] [-delSignatureStrength sig-type sig-strength]
   ```

   where `backofficeidfile.bin` is your back-office identity file. Replace *`sig-type`* and *`sig-strength`* with the desired signature type and signature strength, respectively, that you want to delete.

   Example: The following command deletes the signature type `RSA` with signature strength `4096 bit` and digest strength `SHA-512`.

   ```
   pubidutil.bat -silent -backOffice IdentityBackOffice.bin -delSignatureStrength RSA 35
   ```

***Tip** ▪ For information about signature strength values, see Signature Strengths.*

The additional signature strength is removed from all three identity files. They are saved in the same folder as the existing `IdentityBackOffice.bin` file that you specified on the command line.

Signature Strengths

This section lists the strength value for different signature types available in FlexNet Embedded. The signature strength value is derived from the following elements:

- Signature type:
  - Tamper-Resistant Licenses (TRL): 113-bit, 163-bit, or 239-bit public key encryption
  - License Key (LK): 48-bit proprietary algorithm
  - RSA: 512-bit, 1024-bit, 2048-bit, 3072, or 4096-bit RSA encryption
- Signature strength (low, medium, high)
- Digest strength (SHA-1, SHA-256, SHA-512)

Refer to the table below when you add a signature strength to an identity using the command line, where you replace `sig-strength` with the required signature strength value:

```
pubidutil -silent [-backOffice backofficeidfile.bin] [-addSignatureStrength sig-type sig-strength]
```

Table 2-1 ▪ Signature strength values

Signature Type	Signature Strength	Digest Strength	Signature Type	Signature Strength Value
LICENSE_KEY	**NA**	N/a	LICENSE_KEY	0
TRL	**Low (113 bit)**	N/a	EC_DSA	0
	Medium (163 bit)	N/a	EC_DSA	1
	High (239 bit)	N/a	EC_DSA	2

Table 2-1 ▪ Signature strength values

Signature Type	Signature Strength	Digest Strength	Signature Type	Signature Strength Value
RSA	**Low (512 bit)**	SHA-1	RSA	0
		SHA-256	RSA	16
	Medium (1024 bit)	SHA-1	RSA	1
		SHA-256	RSA	17
		SHA-512	RSA	32
	High (2048 bit)	SHA-1	RSA	2
		SHA-256	RSA	18
		SHA-512	RSA	33
	RSA (3072 bit)	SHA-1	RSA	3
		SHA-256	RSA	19
		SHA-512	RSA	34
	RSA (4096 bit)	SHA-1	RSA	4
		SHA-256	RSA	20
		SHA-512	RSA	35

Tip ▪ *For a list of all RSA signature strengths and digests, use* ***`pubidutil -listRsaTypes`****.*

Next Steps

You need to distribute the updated identity files to take effect:

- Upload the updated `IdentityBackOffice.bin` to your back office. If you are using FlexNet Operations as your back office, see the topic Creating a Publisher Identity in the FlexNet Operations User Guide for more information.
- Update your client components with the new identity data.

Note ▪ *Even though the `IdentityClientServer.bin` file has also been updated to include the additional signature strength, you do not need to update your server component with this identity, because license servers do not currently support multiple signature strengths.*

Existing clients will continue to validate messages received from the back office using the original identity and signature strength. New clients that use the updated client-identity data in their FlexNet Embedded client code support the original and the additional signature strengths.

Building and Running the "Basic Client" Licensing Example

This section walks you through the process of building and running the **BasicClient** example found in the FlexNet Embedded Client Java XT toolkit.

BasicClient is the simplest example of code that uses FlexNet Embedded APIs to enable licensing. It 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 the "BasicClient" Example

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

Building the "BasicClient" Example

You can build the **BasicClient** example using either of these methods:

- Build "BasicClient" Using the Supplied Apache Ant Script
- Build "BasicClient" Manually

Build "BasicClient" Using the Supplied Apache Ant Script

The FlexNet Embedded Client Java XT toolkit provides an Apache Ant script that builds all the toolkit examples, including the **BasicClient** example. Follow these instructions if you want this Ant script to build the examples.

Task ***To build the toolkit examples using the supplied Ant script***

Run the following from the root directory of the FlexNet Embedded Client Java XT toolkit to build all the toolkit examples, including:

```
ant -f ./build/build_client_samples.xml
```

If successful, Ant should display a message such as:

```
[javac] Compiling 11 source files to...examples/client_samples/classes

BUILD SUCCESSFUL
```

The compiled class files should now be available in *install_dir*/examples/client_samples/classes/flxexamples.

Build "BasicClient" Manually

If you do not have Ant installed, you can build one or more the toolkit examples using your Java development environment or command-line tools.

Task ***To build "BasicClient" manually***

1. Using your development environment, create a Java project.
2. Add the following files (located in *install_dir*/examples/client_samples/src/flxexamples) to the project:
   - BasicClient.java
   - IdentityClient.java
3. Add the following libraries (located in *install_dir*/lib) to the build path:
   - flxBinary.jar
   - flxClient.jar
   - flxClientNative.jar

   For example, if you are a using command-line tool to compile **BasicClient**, you would enter a command similar to the following. This sample command assumes that the current directory is examples/client_samples/src/ and that the FlexNet Embedded Client Java XT libraries are in the default toolkit location.

   ```
   javac -d ../classes
         -classpath ../classes;../../../lib/flxClient.jar;
                    ../../../lib/flxClientNative.jar;
                    ../../../lib/flxBinary.jar
         flxexamples/BasicClient.java flxexamples/IdentityClient.java
                    flxexamples/BinaryMessage.java
   ```
4. Build the project. The FlexNet Embedded Client Java XT toolkit provides an empty directory, examples/client_samples/classes, in which to place the compiled class files. Because the toolkit examples use the package name flxexamples, the path to the compiled class files is examples/client_samples/classes/flxexamples.

In the Preparing to Run "BasicClient" section, you will prepare to run the **BasicClient** example by generating the license rights to be acquired by the example and by installing the native library.

Troubleshooting Compilation Errors

If compilation fails with package does not exist or cannot find symbol errors, verify that the flxBinary.jar, flxClient.jar, and flxClientNative.jar libraries are included in the compiler class path and that the IdentityClient and BinaryMessage classes are available.

Preparing to Run "BasicClient"

Before running the BasicClient executable, do the following:

- Generate License Rights

- Install the Native Library 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.lic` 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 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.lic 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 Java XT code can read it.

Install the Native Library for FlexNet Embedded

FlexNet Embedded Client Java XT makes use of a native library for some licensing functionality. The library is located in the `lib` subdirectory of your FlexNet Embedded Client Java XT toolkit. (The variable *`release`* used in certain library names represents the release version of the given library.)

- On Windows, the library is `FlxCore.dll` (for the 32-bit toolkits) and `FlxCore64.dll` (for the 64-bit toolkits).
- On non-Windows systems (except OS X), the library is `libFlxCore.so.`*`release`* (for 32-bit toolkits) or `libFlxCore64.so.`*`release`* (for the 64-bit toolkits).

- On OS X, the library is `libFlxCore.release.dylib`.

Before running the **BasicClient** example—or any other code enabled for FlexNet Embedded licensing, ensure that the `FlxCore` library is properly installed so that your executable can find it. Refer to the appropriate section below for installation instructions for a given platform. (If you have built the FlexNet Embedded examples, including **BasicClient**, with the Ant build script supplied with the toolkit, installation of the native library is performed automatically for you. See Building the "BasicClient" Example.)

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

On Windows

Copy `FlxCore.dll` (or `FlxCore64.dll`) into the directory from which executable will be launched, or add the directory containing this library to the `PATH` environment variable or to the Java system property, `java.library.path`.

On non-Windows Systems

Install `libFlxCore` (or `libFlxCore64`) in your system's standard library directory. Alternatively, you can install the library in a non-standard location; but make sure that the `LD_LIBRARY_PATH` environment variable or the Java system property, `java.library.path`, points to the absolute path of the directory containing the library.

Running the "BasicClient" Example

At this point, you can put the components together and test the **BasicClient** example:

- Execute the Example
- Output from Executing the "Basic Client" Example
- Troubleshooting Build Errors

Execute the Example

Use the following procedure to run the **BasicClient** example.

Task ***To run the "BasicClient" example***

1. Copy the binary license rights file `license.bin` you generated in the section Generate License Rights into the current directory.
2. Ensure that you have placed the compiled class files in `examples/client_samples/classes/flxexamples` (a task automatically done when you build the examples using the Ant script provided in the toolkit).
3. Launch the script `BasicClient` or `BasicClient.bat` from the `classes` directory, passing the location of `license.bin` as a command-line argument:

```
BasicClient.bat license.bin
```

On Linux or other supported (non-Windows) systems, the command would be similar:

```
.\BasicClient license.bin
```

Optionally, to launch **BasicClient** without using the script provided with the toolkit, make sure that the following FlexNet Embedded client libraries are specified on the Java run-time class path before using an appropriate launch command:

- `flxBinary.jar`
- `flxClient.jar`
- `flxClientNative.jar`
- `commons-codec-1.3.jar`

Note that, in a production environment, your installation process would normally place these FlexNet Embedded client libraries and the native `FlxCore` library in a location accessible by your application, and would launch the application using a script or shortcut.

Output from Executing the "Basic Client" Example

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

```
Reading data from license.bin.
Successfully acquired "survey", version 1.0, 1 count.
Successfully acquired "highres", version 1.0, 1 count.
```

Troubleshooting Build Errors

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

- This error might be generated: `License-enabled code requires client identity data, which you create with pubidutil and printbin -java or with FlexNet Operations`. Ensure that you have copied the Java-compatible identity file, `IdentityClient.java`, to the `examples/client_samples/src/flxexamples` directory. (This is the identity file you created with `pubidutil` and `printbin`.)
- If an error beginning `Failed to load FlxCore library` is displayed, verify that the native `FlxCore` library has been properly installed, as described in Install the Native Library for FlexNet Embedded.
- If an error message `Unable to find file license.bin` is displayed, verify that the `BasicClient` executable was unable to find the binary license rights file `license.bin`. The default location is always the current directory, but you can specify the path to the buffer license file as a command-line argument to `BasicClient`.
- Verify that both the client identity (`IdentityClient.java`) 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.
- (Linux only) If the example fails to execute with the error `.../lib/libFlxCore.so.`*`xxxx.xx`*`: cannot restore segment prot after reloc: Permission denied`, the issue is the SELinux security system blocking the launch.

To enable using the library in such an environment, run the following command, providing the correct version name for the library:

```
chcon -t textrel_shlib_t ../../../lib/libFlxCore.so.2018.05
```

- 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.
- The example can generate an `UnsatisfiedLinkError` exception if it uses 32-bit libraries but is attempting to run on a 64-bit Java virtual machine (or vice versa). Make sure that applications with 32-bit libraries run on 32-bit Java virtual machines and that applications with 64-bit libraries run on 64-bit Java virtual machines. (This exception usually does not occur on a OS X platform.)

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 Java 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 server.
- Using utilities provided with the toolkit.

3

Toolkit Overview

FlexNet Embedded Client Java XT provides a collection of libraries, example source code, and utilities used by a producer to create code enabled for FlexNet Embedded licensing.

This chapter provides basic information to help you get started with FlexNet Embedded Client Java XT toolkit:

- FlexNet Embedded Concepts
- FlexNet Embedded Client Java XT Toolkit Requirements
- Toolkit Contents
- About the Example Projects
- Building the Example Projects
- Running the Examples
- Toolkit Files to Distribute with Your Product

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 Java 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 3-1 ▪ 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

Table 3-1 ▪ Supported Hostid Types and their Keywords

Hostid Type	Keyword	Example	Case Sensitive
Internet (IPv4) address	INTERNET	HOSTID=INTERNET=11.22.33.44	No
Internet (IPv6) address	INTERNET6	HOSTID=INTERNET6=2001:0db8:0000:0000:0000: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

Information for Producers Supporting flexid9 or flexid10 Hostids

Producers who want to support flexid9 or flexid10 hostids for their applications should download the **FlexNet Embedded Accessories for Dongles (*YYYY.MM*)** zip file from the Revenera Product and License Center. This zip contains flexid9 and flexid10 dongle drivers and shared objects.

Task ***To download the flexid9 and flexid10 dongle libraries from the Product and License Center***

1. In the Revenera Product and License Center, in the Product List, navigate to **FlexNet Licensing** > **FlexNet Embedded**.
2. Locate and download the **FlexNet Embedded Accessories for Dongles (*YYYY.MM*)** artifact, where *YYYY* and *MM* stand for the year and month corresponding to the relevant FlexNet Embedded release.
3. Extract the file dongle--fne_*YYYY*.*MM*.zip from the artifact.

The file dongle--fne_*YYYY*.*MM*.zip contains the following files:

Table 3-2 ▪ Files in dongle--fne_*YYYY*.*MM*.zip

Platform Folder	FLEXID	Installer	Shared Object
x64_windows	**flexid9**	FLEXID9_Windows_v*_*_x64.zip	haspsrm_win64.dll
	flexid10	FLEXID10_Windows_v*_*_x64.zip	N/A
i86_windows	**flexid9**	FLEXID9_Windows_v*_*_i686.zip	haspsrm_win32.dll
	flexid10	FLEXID10_Windows_v*_*_i686.zip	N/A

Table 3-2 ▪ Files in dongle--fne_*YYYY*.*MM*.zip

Platform Folder	FLEXID	Installer	Shared Object
x64_mac10	**flexid9**	FLEXID9_OSX_V*_*.dmg	hasp_darwin.dylib
	flexid10	FLEXID10_OSX_*.dmg	N/A
x64_linux	**flexid9**	aksusbd-redhat-suse-*.tar.gz aksusbd-ubuntu-*.tar.gz	libhasp_linux_x86_64.so libhasp_linux_x86_64.so
	flexid10	WkRt-Lin-*.x86_64.rpm	N/A
i86_linux	**flexid9**	aksusbd-redhat-suse-*.tar	libhasp_linux_i686.so
	flexid10	WkRt-Lin-*.i386.rpm	N/A

The following section, flexid9, explains the additional steps required for applications supporting flexid9 hostids. There are no additional steps required for flexid10 hostids.

flexid9

From release 2023.09 onwards, the dongle libraries required for flexid9 hostids are no longer statically linked in the FlexNet Embedded Client Java XT toolkit. Instead, if you want your application to support the flexid9 hostid, you must include the relevant dll/dylib/so files (included as a separate library file in the relevant <platform_dir> folder in the dongle--fne_*YYYY*.*MM*.zip, see previous section) in your application's installer.

The following table lists the names of the user-space dongle dynamic libraries (shared objects) that must be loaded at run-time by FlexNet Embedded client applications that need to detect flexid9 hostids:

Platform	File Name
x64_windows	haspsrm_win64.dll
i86_windows	haspsrm_win32.dll
x64_mac10	hasp_darwin.dylib
x64_linux	libhasp_linux_x86_64.so
i86_linux	libhasp_linux_i686.so

Task ***To include the dongle DLL/shared object for flexid9 hostids in your client application's installer:***

- On **Windows**, do one of the following:
  - Copy the dongle DLLs, haspsrm_win32.dll and haspsrm_win64.dll for 32-bit and 64-bit platforms respectively, to the following folders:
    - On a 64-bit system, install haspsrm_win64.dll to %windir%/System32.

    - On a 32-bit system, install `haspsrm_win32.dll` to `%windir%/System32`.
  - Use the environment variable `FLEXID_LIBRARY_PATH` to specify the location of the dongle DLL, `haspsrm_win32.dll` or `haspsrm_win64.dll` (depending on your architecture). Best practice is to set the path to the same folder as your application. Communicate the location to your application's end users. Before running your application, end users must set the environment variable `FLEXID_LIBRARY_PATH` to the location you defined.
- On **macOS**, do one of the following:
  - Define the environment variable `FLEXID_LIBRARY_PATH` to point to the location of the shared object, `hasp_darwin.dylib`.
  - Define the system environment variable `DYLD_LIBRARY_PATH` to point to the location of the shared object, `hasp_darwin.dylib`.

  Communicate the location to your application's end users. Before running your application, end users must set either the environment variable `FLEXID_LIBRARY_PATH` or the system environment variable `DYLD_LIBRARY_PATH` to the location you defined.
- On **Linux**, do one of the following:
  - Define the environment variable `FLEXID_LIBRARY_PATH` to point to the location of the shared object, `libhasp_linux_i686.so` or `libhasp_linux_x86_64.so` (depending on your architecture).
  - Define the system environment variable `LD_LIBRARY_PATH` to point to the location of the shared object, `libhasp_linux_i686.so` or `libhasp_linux_x86_64.so` (depending on your architecture).

  Communicate the location to your application's end users. Before running your application, end users must set either the environment variable `FLEXID_LIBRARY_PATH` or the system environment variable `LD_LIBRARY_PATH` to the location you defined.

Important ▪ *The value of FLEXID_LIBRARY_PATH should not exceed 255 characters.*

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 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. 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 the 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 the 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 FlexNet Embedded Client Java XT toolkit provides tools—including the `licensefileutil` utility 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 FlexNet Embedded Client Java XT 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 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 the *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 in the *FlexNet Embedded License Server Producer Guide*, and the CLS (Cloud Licensing Service) license server, described in the *FlexNet Operations Getting Started Guide for Cloud Licensing Service* 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 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. 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.

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.

FlexNet Embedded Client Java XT 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*.

Toolkit Contents

The FlexNet Embedded Client Java XT toolkit `.zip` file or `.tgz` file 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:

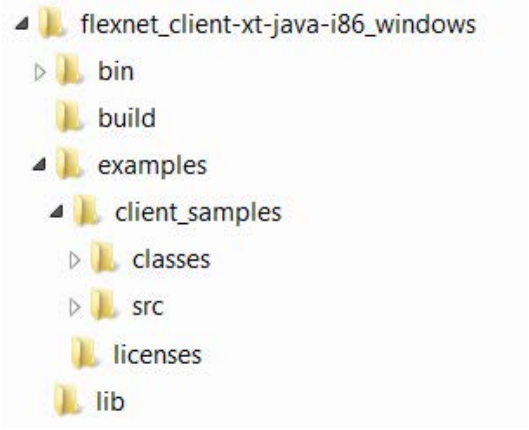


Figure 3-1: Directories in the FlexNet Embedded Client Java XT Toolkit

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

Table 3-3 ▪ FlexNet Embedded Client Java XT Toolkit Directories

Directory	Contents
bin	The `bin` directory containing the FlexNet Embedded Client development and testing utilities. These various command-line tools create binary producer identity data, generate different types of license rights, and so forth.
build	Ant script for building the example projects.
examples	Source code and other files needed to build the examples in the FlexNet Embedded Client toolkit. The files are organized in these subdirectories: • `client_samples`: Source code for various example license-enabled projects and for example projects enabled for Tamper Resistance for Applications (TRA) technology. • `licenses`: Example unsigned license files.
lib	The native and core FlexNet Embedded Client Java XT libraries used in license-enabled code.

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 Java XT Toolkit chapter, 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 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/src directory.)

- 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.

  The example 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** 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.

Building the Example Projects

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

- Obtaining Producer Identity Data
- Building the Examples

Obtaining Producer Identity Data

Each producer is identified by a unique producer name and producer keys. To enable your FlexNet Embedded Client Java 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.

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.java` file to the *install_dir*`/examples/client_samples/src/flxexamples` directory.) Then use the following procedures to build the example projects.

Building the Examples

To build the examples of the FlexNet Embedded Client Java XT toolkit, follow these steps.

Build Examples Using Ant

Using Apache Ant, run the following command from the root of the FlexNet Embedded Client Java XT toolkit to build the FlexNet Embedded examples:

```
ant -f ./build/build_client_samples.xml
```

Build Examples Manually

To build the examples manually, without using Ant repeat the following for each example:

1. Using your development environment, create a Java project. Add the following files from the `examples/client_samples/src/flxexamples` subdirectory to your project:
   - `BasicClient.java`
   - `IdentityClient.java`
   - `BinaryMessage.java`
2. Add the following libraries from the `lib` subdirectory to the build path:
   - `flxBinary.jar`
   - `flxClient.jar`
   - `flxClientNative.jar`
   - `commons-codec-1.9.jar`
3. Build the project.

   The FlexNet Embedded Client Java XT toolkit provides an empty directory, `examples/client_samples/classes/flxexamples`, in which to place your compiled class files. (Note that `flxexamples` is the package name of the examples.)

   For example, if you are compiling the example using command-line tools, you would enter a command similar to the following. This sample command assumes that the current directory is `examples/client_samples/src/` and that the FlexNet Embedded Client Java XT libraries are in the default toolkit location.

   ```
   javac -d ../classes
         -classpath ../classes;../../../lib/flxClient.jar;
               ../../../lib/flxClientNative.jar;
               ../../../lib/flxBinary.jar
         flxexamples/BasicClient.java flxexamples/IdentityClient.java
   ```

Running the Examples

The toolkit examples are console executables. To display help information for any example, launch the executable with the -h or -help switch, as shown here for the client executable:

```
Client -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:

- Generate Example License Rights
- Run the Example “Client” Project
- Output for a Successful “Client” Execution

Generate 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-jan-2016
INCREMENT special demo 1.0 permanent uncounted HOSTID=ID_STRING=1234567890 START=1-jan-2016
INCREMENT updates demo 1.0 permanent uncounted HOSTID=ID_STRING=1234567890 START=1-jan-2016
INCREMENT sdchannel demo 1.0 permanent 100 HOSTID=ID_STRING=1234567890 START=1-jan-2016 \
   VENDOR_STRING="Standard Definition Channels"
INCREMENT hdchannel demo 1.0 permanent 10 HOSTID=ID_STRING=1234567890 START=1-jan-2016 \
   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.

Run the Example "Client" Project

The toolkit examples are console applications. Use the following procedure to run the **Client** example.

Task ***To run the "Client" example***

1. Ensure that the native `FlxCore` library has been properly installed. For more information, see Install the Native Library for FlexNet Embedded.
2. Copy the binary license rights file `demo.bin` you generated in the section Generate Example License Rights into the current directory.
3. Ensure that you have placed the compiled class files in `examples/client_samples/classes/flxexamples` (a task automatically done when you build the examples using the Ant script provided in the toolkit).
4. Launch the script `Client` or `Client.bat` from the `classes` directory, passing the location of `demo.bin` as a command-line argument:

   ```
   Client.bat demo.bin
   ```

   On Linux or other supported (non-Windows) systems, the command would be similar:

   ```
   ./Client demo.bin
   ```

   Optionally, to launch **Client** without using the script provided with the toolkit, make sure that the following FlexNet Embedded client libraries are specified on the Java run-time class path before using an appropriate launch command:

   - `flxBinary.jar`
   - `flxClient.jar`
   - `flxClientNative.jar`
   - `commons-codec-1.3.jar`

Output for a Successful "Client" Execution

When you successfully the Client example, the console displays output similar to the following:

```
Reading data from demo.bin.
Number of features loaded from buffer: 8.
Number of features loaded from trusted storage: 0.
Number of features loaded from trial storage: 0.
Successfully acquired "survey", version 1.0, 1 count.
Successfully acquired "highres", version 2.0, 1 count.
Unable to acquire lowres : Requested feature was not found.
Successfully acquired "download", version 2.0, 1 count.
Successfully acquired "upload", version 2.0, 1 count.
Successfully acquired "updates", version 1.0, 1 count.
Successfully acquired "special", version 1.0, 1 count.
```

```
Successfully acquired "sdchannel", version 1.0, 100 count.
Successfully acquired "hdchannel", version 1.0, 10 count.
```

The messages indicate whether a license for a given feature in the binary license file could be acquired. (For example, the output mentions that the feature “lowres” is not present in the license rights.)

In later chapters, you will see how to experiment with different representations of license rights and processes using the other examples and additional toolkit utilities.

Toolkit Files to Distribute with Your Product

When you distribute a product that has been leveraged with FlexNet Embedded licensing, you need to distribute certain toolkit files along with the product. The following table lists these files, all of which are found in the `lib` folder of your toolkit installation. (Ship the files listed for both **Common** and **FlexNet Embedded** in the table.)

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

Table 3-4 ▪ Toolkit Deliverables to Customers

Client Functionality	Files on Windows	Files on Linux	Files on OS X
Common	• `commons-codec-1.9.jar` • `EccpressoAll.jar` • `flxBinary.jar`	• `commons-codec-1.9.jar` • `EccpressoAll.jar` • `flxBinary.jar`	• `commons-codec-1.9.jar` • `EccpressoAll.jar` • `flxBinary.jar`
FlexNet Embedded	• `FlxCore.dll` (or `FlxCore64.dll`) • `flxClient.jar` • `flxClientNative.jar` • `tra-run.jar*`	• `libFlxCore.so.`*`version`* (or `libFlxCore64.so.`*`version`*) • `flxClient.jar` • `flxClientNative.jar` • `tra-run.jar*`	• `libFlxCore.`*`version`*`.dylib` • `flxClient.jar` • `flxClientNative.jar` • `tra-run.jar*`

* Required only if your product incorporates FlexNet Embedded Tamper Resistance for Applications (TRA) functionality (see the *FlexNet Embedded TRA for Java XT Getting Started Guide*).

4

Overview of the Java XT APIs

This chapter provides an overview of the Java functionality included in the FlexNet Embedded Client Java XT toolkit. The functionality is organized into API.

- FlexNet Embedded API Interfaces—Used to perform licensing-related operations.
- FlexNet Common API Interfaces—Used by FlexNet Embedded to handle errors and communications with servers.

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

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

FlexNet Embedded API Interfaces

The following are the primary interfaces defined by the FlexNet Embedded Client Java XT 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 customize binary capability requests that are communicated to a back-office server or a license server. The server processes the request and generates a capability response. Depending on the type of server that will receive the request, the request may include 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.
- **ICapabilityResponseData:** 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.
- **Comm:** This interface provides support for sending binary FlexNet Embedded messages to a back-office server or license server and receiving responses via HTTP, SSL, and proxy-sever communications.

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

FlexNet Common API Interfaces

The following lists the groups of FlexNet Embedded Client Java XT APIs 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.*

- **IComm:** This interface provides support for sending binary FlexNet Embedded messages to a back-office server or a license server and receiving the response via HTTP, SSL, and proxy-server communications. It also provides support to download update or synchronization data from the content delivery host.
- **IStatusInformation:** This interface provides information on the error code, status information and message of FlxException. It also allows access to error information supplied by the back-office server and is available (if supplied by the back office server) through exception classes derived from FlxException.

The following enumerations are used to describe server errors or errors in execution.

- **Server Error Codes:** The list of error codes identifying errors that the FlexNet Embedded client might encounter when FlexNet Embedded talks with the back-office server or license server.
- **Common Error Codes:** The list of error codes that might be generated when FlexNet Embedded functionality is executed on the client.

Conventions for Retrieving Exception Information

The FlexNet Java Client XT API methods in some cases 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 about specific methods and their possible exceptions.

Note ▪ *Note that "FlxException" constructors are not intended for public use. It is recommended that your code not throw built-in* FlexNet Embedded Client Java XT *exceptions, but instead throws your own exceptions. Following this convention will help to reduce confusion when debugging your application code.*

The following are conventional methods in `FlxException` used to retrieve information about FlexNet Embedded client exceptions:

- The `getMessage` method retrieves an error or warning string corresponding to a particular exception.
- The `getDiagnosticMessage` method retrieves other details about the exception, such as an error code.
- The `getLocalizedMessage` method reformats an error message for the current locale.

These methods (in the `LicensingExceptions` class) obtain exception information useful to Revenera in resolving issues:

- The `getSystemCode` retrieves the error code from the operating system.
- The `getError` retrieves the FlexNet Embedded error code.
- The `getErrorCategory` retrieves the error category.
- The `getErrorLocation` retrieves the location in the code where the error occurred.

The following implementation is an example of how to retrieve information for an exception. The `to.string` method is used to provide a character representation of the retrieved information.

```
catch (LicensingException e) {
    int location = e.getLocation();
    ErrorCode errCode = e.getError();
    ErrorCategory errCat = e.getErrorCategory();
    System.out.println("Error: " + errCode.toString() + ", Category: " +
      errCat.toString() + ", SystemError=" + systemError + ", Location=" + location);
}
```

5

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 Java XT toolkit, which is a family of Java interfaces and methods for processing license rights, acquiring licenses, querying license data, and processing communications with the back-office server or a license server.

The following sections describe 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 (and in many cases sample license rights) discussed in this chapter are located in the directory *install_dir*`/examples/client_samples/src/flxexamples`.

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

The scenarios described here are:

- 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 Obtaining Producer Identity Data.

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

All of the example code uses the Java package name `flxexamples`:

```
package flxexamples;
```

In practice your code can use any package name. Keep in mind that the client identity data file `IdentityClient.java` that you created with printbin uses the `flxexamples` package, but you can change this in the source file or with the `-package` switch to `pubidutil`.

For more information about generating your identity information and distributing it appropriately, see Creating the Producer Identity.

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:

```
ILicensing      licensing      = null;
ILicenseManager licenseManager = null;

// Initialize ILicensing interface with identity data using file-based trusted storage in
// user's home directory and hard-coded string hostid "1234567890"
licensing = LicensingFactory.getLicensing(
   IdentityClient.IDENTITY_DATA, System.getProperty("user.home"), "1234567890", "examplename");
// Get ILicenseManager interface, the primary interface for licensing-related functionality
licenseManager = licensing.getLicenseManager( );
```

Note ▪ *In the* FlexNet Embedded Client Java XT *examples, most licensing operations are wrapped in a "try-catch-finally" 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.IDENTITY_DATA` expression passed as the first argument to `LicensingFactory.getLicensing`—used in this code sample can be found in `IdentityClient.java` 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 Java code.*

Caution ▪ *In multi-threaded code using the same "LicenseManager" object, method calls on that object must be synchronized. For example, calls from the creation of the "ILicensing" object to the end of calls on the "LicenseManager" object can be placed in a "synchronized" block.*

Specifying the Trusted Storage Location

The second argument to `getLicensing` is the 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 writable directory on the target system. (Your installation program or instructions should adjust directory permissions, as appropriate.) The examples use the value of the Java system property `user.home` as the location to store trusted storage.

Passing `null` in this argument causes FlexNet Embedded Client Java XT 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 often appropriate.

Note that `getLicensing` throws an exception if the identity data uses evaluation keys that have expired, do not enable Java support, or are otherwise invalid.

After calling `LicenseFactory.getLicensing` to initialize your `ILicensing` object, call `getLicenseManager` to create 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 other client system identification. For testing, however, you can specify a string to use as a hard-coded hostid value. The example code in the FlexNet Embedded Client Java XT toolkit uses hard-coded 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-sensitivity, 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 types of system identification. When not using the string hostid override in `getLicensing`, use the `setHostId` method of the `ILicenseManager` interface to set the desired hostid. The method signature is:

```
void setHostId(SharedConstants.HostIdType type, String id);
```

This function uses the following arguments:

- *type* is the hostid type to use. Supported types include:
  - `SharedConstants.HostIdType.ETHERNET` for an Ethernet address
  - `SharedConstants.HostIdType.INTERNET` for an IPv4 address
  - `SharedConstants.HostIdType.INTERNET6` for an IPv6 Internet address
  - `SharedConstants.HostIdType.FLEXID9` for an Aladdin dongle
  - `SharedConstants.HostIdType.FLEXID10` for a Wibu-Systems dongle

- `SharedConstants.HostIdType.VM_UUID` for a supported virtual machine's UUID
- `SharedConstants.HostIdType.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 machine at run time, call the `getHostIds` method of the `ILicenseManager` interface. Its signature is:

```
Map<SharedConstants.HostIdType, List<String>> getHostIds( )
```

In practice, client code will typically read the available hostid types and values with `getHostIds`, and then call `setHostId` with the desired value. (The `setHostId` method throws an exception if the specified hostid is not present on the system. Modifying the collection returned by `getHostIds` 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
Map<HostIdType, List<String>> hostIds = licenseManager.getHostIds( );

// select only the Ethernet addresses
List<String> hosts = hostIds.get(HostIdType.ETHERNET);

// use the first Ethernet address (index 0) in the list
licenseManager.setHostId(HostIdType.ETHERNET, hosts.get(0));
```

It might be necessary to import the `java.util.Map` and `java.util.List` packages to compile the code. 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, 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 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 `getHostIds` method of the `iLicenseManager` interface to return and read the available hostid types. In containerized environments, calling `getHostIds` 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 the `getCloneSuspect` method (in the `ICapabilityResponseData` interface) in your client code to retrieve this status directly from the response and use it for your own purposes.

The following shows an example implementation of this method:

```
// check if clone suspect
ICapabilityResponseData response = licMgr.getResponseDatails(responseBuffer);
If (response.getCloneSuspect()) {
        // 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 `enableWindbackDetection` of your `ILicenseManager` object. (You can disable clock-windback detection using `disableWindbackDetection`.) In the following method signature, the arguments are for the windback tolerance and frequency:

```
enableWindbackDetection(int tolerance, int 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 `windbackDetected`. Calling the method `enableWindbackDetection` 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 the 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`.

Use the `setRequestorId` method 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 Java 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 Java 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.

Methods Used to Retrieve Expiration Information

The following methods in the `IFeature` interface retrieve the expiration information:

- **getExpiration (in IFeature 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).

- **getFinalExpiration (in IFeature interface)**—Obtains the final expiration date for a feature.
- **getEntitlementExpiration (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. The same custom data can be returned in the response if requested by the client using the `-includeDictionary`, `-includeDictionaryKey`, or `-attr` flag in the capability request. 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, call `getVendorDictionary` to retrieve vendor dictionary items from the response.

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

```
// Optional vendor dictionary
options.addVendorDictionaryItem("key1", "one");
options.addVendorDictionaryItem("key2", 2);
```

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 Obtaining Producer Identity Data, 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 Java-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 License Checkout from the 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.

Additional Configuration

Secure anchoring leverages a number of operating-system mechanisms. Sometimes a desired mechanism is not available to secure anchoring—that is, either the mechanism is not in use on the operating system, or it *is* in use, but secure anchoring does not have access to it. Secure anchoring is then forced to rely on a back-up mechanism or use an alternate strategy altogether.

For the most effective use of secure anchoring on a given client machine, your application deliverable might need to perform extra steps to ensure that technologies currently in use on the operating system are accessible to secure anchoring.

For example, to take advantage of secure anchoring on a Linux machine, each end user of your license-enabled product on this machine must be a member of the FUSE group. You can add users to this group using the command `usermod -a -G fuse username`.

The following shows one method for inserting this command into a script, where *usergroup* is a group of users expected to use FUSE:

```
for u in $(lid -g -n usergroup); do usermod -a -G fuse $u; done
```

Alternatively, an installer might use the `fuse.conf` file to specify the privileges required to use FUSE. For complete details, refer to the FUSE Sourceforge website (http://sourceforgeJava/projects/fuse).

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, as are the hostid values specified in the `HOSTID` keyword.

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 that will run 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 `Client.java`, located in the `examples/client_samples/src/flxexamples` directory.

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

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

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 `BinaryMessage.readData`), which is then loaded as a buffer license source.

```
// Add buffer license source
if (inputFile != null) {
   System.out.println("Reading data from " + inputFile);
   licenseManager.addBufferLicenseSource(BinaryMessage.readData(inputFile));

   // Get valid features from buffer license source
   List<IFeature> bufferFeatures =
      licenseManager.getFeaturesFromBuffer(BinaryMessage.readData(inputFile), false);
   System.out.println("Number of features loaded from buffer: " + bufferFeatures.size( ));
}
else {
   System.out.println("No license file specified.");
}
```

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 Java "InputStream".*

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.

```
ILicense license = null;

try {
   license = licenseManager.acquire("survey", "1.0", 1);
   System.out.println("Successfully acquired \"survey\" license.");
}
catch (FlxException e) {
   System.out.println("Unable to acquire \"survey\" license: " + e.getMessage( ));
}

try {
   // return license
   licenseManager.returnLicense(license);
}
catch (FlxException e) {
   System.out.println("Unable to release \"survey\" license: " + e.getMessage( ));
```

```
}
```

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 `availableAcquireCount` 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, "getFeaturesFromBuffer", "getFeaturesFromTrustedStorage", and "getFeaturesFromTrials" will list features in a license source without validating signatures or expiration dates; and "getAcquisitionStatus" 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 calling accessor methods on the `ILicense` interface obtained with the `acquire` method:

```
System.out.println("Successfully acquired \"" + license.getName( ) + "\", version " +
      license.getVersion( ) +  ", " + license.getCount( ) + " count.");
```

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:

- FlexNet Operations as "Back-Office Server"

- 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 toolkit source code in the file *`install_dir`*`/examples/client_samples/src/flxexamples/CapabilityRequest.java` for an example implementation.

Running the **CapabilityRequest** example 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 it creates a trusted storage license source.

```
// Add trusted storage license source
licenseManager.addTrustedStorageLicenseSource( );
```

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 = licenseManager.createCapabilityRequestOptions( );

// Force back-office server to send a response
options.forceResponse( );

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

// Generate the request
requestData = licenseManager.generateCapabilityRequest(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 desired features as partial in the capability request, provide this basic flow in the code:

1. Create an `IRightsIdOptions` object using the `licenseManager.createRightsIdOptions` method, and set the `Partial` flag to `true`.
2. For each rights ID you want to mark as "partial", use the `AddRightsId` method that takes a rights ID options object.

Example Implementation

The following shows a sample implementation that requests three rights IDs—two marked as "partial" and one (`li3`) not marked with this attribute:

```
// Get request options
ICapabilityRequestOptions  options = licenseManager.createCapabilityRequestOptions();
// Create rights ID options
IRightsIdOptions rOptions = licenseManager.createRightsIdOptions();
// Enable partial
rOptions.setPartial(true);
// Add rights ID
options.addRightsId("li1", 10, rOptions);
options.addRightsId("li2", 5, rOptions);
options.addRightsId("li3", 5);
```

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 calls `setHostName` and `setHostType` from the `ILicenseManager` interface.

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 of `ICapabilityRequestOptions` 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 `talkToServer` 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 methods used for the different types of communications.*

The example's `talkToServer` implementation uses the `sendBinaryMessage` method of the `Comm` interface to send the binary capability request to a server over HTTP and receive the response. Any HTTP errors encountered are wrapped in a `FlxException`. (See the API reference for method details.)

However, it is not a requirement to use the `Comm` 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 HTTP 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.

Commands Used to Send the Request in the Example

For the **CapabilityRequest** example, enter a command similar to one of the following to send the request to the appropriate back-office server target.

To the Test Back-office Server

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
```

To FlexNet Operations

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
```

To a Proxy Server

To send the request to a proxy server (which has been set up to communicate with the back-office server), first update the example code to set the proxy-server information. You define this information explicitly in the `Comm.getHttpInstance` API (found in the `talkToServer` helper function):

```
Comm conn = Comm.getHttpInstance(uri, http://109.224.11.671, 8660, null, null);
```

where:

- `uri` will be set with the server name you provide in the command to run the example.
- `http://109.22.11.671` is an example name of a proxy server.

- `8660` is an example port for the proxy server.
- `null, null` are the parameters specifying an omitted user ID and password. For authentication purposes, you can set these parameters to the user ID and password needed to access the proxy server. Otherwise, set them to `null`.

Then run the appropriate command with the URL for the actual back-office server. (In this sample command, FlexNet Operations is the back-office server.)

```
CapabilityRequest -server http://servername:8888/flexnet/deviceservices
```

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 following shows the code that processes the response into trusted storage:

```
byte[] responseBytes = null;
ICapabilityResponseData responseDetails = null;

responseBytes = talkToServer(requestData, /* URL specified by command-line arguments */);

responseDetails = licenseManager.processCapabilityResponse(responseBytes);
```

Retrieving Response Contents

If the response contains any vendor-dictionary items sent by the server, the code can call `getVendorDictionary` 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, call `getResponseStatus`, followed by iterating over the list of items, calling methods of the `IResponseStatus` class 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 the `getConfirmationRequestNeeded`

method in the `ICapabilityResponseData` interface. 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. (An exception is when the in-memory trusted storage implementation is used.)

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. 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 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
- Register the Client with the Cloud Licensing Service
- 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 a 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.

Specify the URL for the CLS license server in the capability request, as described in the next section.

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 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.

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("f1", "1.0", 1);
options.addDesiredFeature("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 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 `setIncremental` method 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”:

```
// Get request options
ICapabilityRequestOptions  options = licenseManager.createCapabilityRequestOptions();
// Set incremental
```

```
options.setIncremental(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 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. Define a feature-options object using the `createDesiredFeatureOptions` method in the `IDesiredFeatureOptions` interface.
2. Use `setPartial` to enable the "partial" attribute as feature option in the object.
3. For each desired feature you want to mark as "partial", use the `AddDesiredFeature` method with a pointer to the feature-options object.

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

```
// Get request options
ICapabilityRequestOptions  options = licenseManager.createCapabilityRequestOptions();
// Create desired feature option
IDesiredFeatureOptions dfOptions = licenseManager.createDesiredFeatureOptions();
// Enable partial
dfOptions.setPartial(true);
// Add desired feature
options.addDesiredFeature("survey", "1.0", 10, dfOptions);
options.addDesiredFeature("highres", "1.0", 5, dfOptions);
options.addDesiredFeature("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.)

```
// Get request options
ICapabilityRequestOptions  options = licenseManager.createCapabilityRequestOptions();
// Add feature selectors
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:

```
// Get request options
ICapabilityRequestOptions  options = licenseManager.createCapabilityRequestOptions();
// Add auxiliary hostids
options.addAuxiliaryHostId(HostIdType.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 of `ICapabilityRequestOptions`, 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 `setBorrowInterval` method in the `ICapabilityRequestOptions` interface.

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

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 a 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 = licenseManager.createCapabilityRequestOptions();
options.setRequestOperation(RequestOperation.PREVIEW);
```

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

- `IDesiredFeatureOptions.setPartial` set to `true`
- `ICapabilityRequestOptions.setCorrelationId`
- `ICapabilityRequestOptions.setIncremental` 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 = licenseManager.createCapabilityRequestOptions();
options.setRequestOperation(RequestOperation.PREVIEW);
options.addDesiredFeature("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` method 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` method (set to `true`) in the `ICapablityRequestOptions` interface, as shown in the following sample implementations:

```
ICapabilityRequestOptions options = licenseManager.createCapabilityRequestOptions();
options.setRequestOperation(RequestOperation.PREVIEW);
options.requestAllFeatures();
```

or

```
ICapabilityRequestOptions options = licenseManager.createCapabilityRequestOptions();
options.setRequestOperation(RequestOperation.PREVIEW);
options.requestAllFeatures(true);
```

Note the following restrictions when using the `requestAllFeatures` method (set to `true`) in a preview capability request:

- Do not use this method with the `ICapablityRequestOptions.addDesiredFeature` method in the same preview capability request. You can either request all features or add desired features, not both.

- This method 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 uses the `ICapablityRequestOptions.addDesiredFeature` or the `ICapablityRequestOptions.requestAllFeatures` method.

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

Processing a 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` method 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

Your code can use the toolkit methods (and custom methods) to create a feature collection from the preview capability response, iterate through the collection, extract details for each preview feature, including its count and maximum count, and display this information as output.

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

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

This code shows a sample implementation for examining the preview capability response:

```
private static void examinePreviewResponse(ICapabilityResponseData response) {
       if (response.isPreview()) {
```

```
                // is a preview response, look at features
                System.out.println("Preview capability response");
                try {
                      for (IFeature feature : response.getFeatures()) {
                            // print out feature info
                            System.out.println(feature.getName() + " " +
                                        feature.getVersion() + " " +
                                        "TYPE=preview" + " " +
                                        "COUNT=" + feature.getCount() + " " +
                                        "MAXCOUNT=" + feature.getMaxCount());
                       }
                }
                catch (FlxException e) {
                      // handle error here
                }
          }
}
```

Display the Preview Features

The output from the preview capability exchange can include the following information:

- A statement 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). For the **CapabilityRequest** example, this line for the single preview feature might include 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` method to preview all desired features, your code needs to set this method to "false" (and build the desired features from scratch) when reusing the object:

```
ICapabilityRequestOptions options = licensing.LicenseManager.CreateCapabilityRequestOptions();
options.requestAllFeatures(false);
options.setRequestOperation(RequestOperation.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. They are useful in situations where the producer allows users to try out a product or certain product features before making a purchase. 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 Java XT toolkit provides a `trialfileutil` utility for creating signed binary trial license rights, based on an unsigned text license file. Using `trialfileutil`, perform these steps:

1. 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
   ```

2. To create the binary trial file, run the following command (which, in this example, creates 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/src/flxexamples` 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 is created.

```
// Add trial license source
licenseManager.addTrialLicenseSource( );
```

If trials storage has been previously used to store rights, 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 (inside `BinaryMessage.readData`). Note that the `BinaryMessage` class is not part of the standard API, and is used solely for demonstration purposes. Trial information can instead be embedded in the executable binary, by converting the binary trial file into a Java-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 one of the previously created license sources. Currently only trials license sources are available for storing trial license rights.

```
// Read new trial data
System.out.println("Reading trial data from " + trialFile);
trialBuffer = BinaryMessage.readData(trialFile);

// See if it has already been loaded
Long expirationInSeconds = licenseManager.trialIsLoaded(trialBuffer);
if (expirationInSeconds != null) {
   // Trial already loaded
   if (expirationInSeconds.longValue( ) == 0)
      System.out.println("Trial has already been loaded and has expired.");
   else
      System.out.println("Trial expires in: " + expirationInSeconds.longValue( ));
}
else {
   // Load the new trial
   licenseManager.processTrial(trialBuffer);

   // Get features from trial license source
   trialFeatures = licenseManager.getFeaturesFromTrials( );
   System.out.println("Number of features loaded from trials is: " + trialFeatures.size( ));
}
```

Different forms of the `processTrial` method accept the trial byte array, a stream, or a file path.

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. 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 `processTrial` 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 `Trials.java` source code in *`install_dir`*`/examples/client_samples/src/flxexamples` 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.

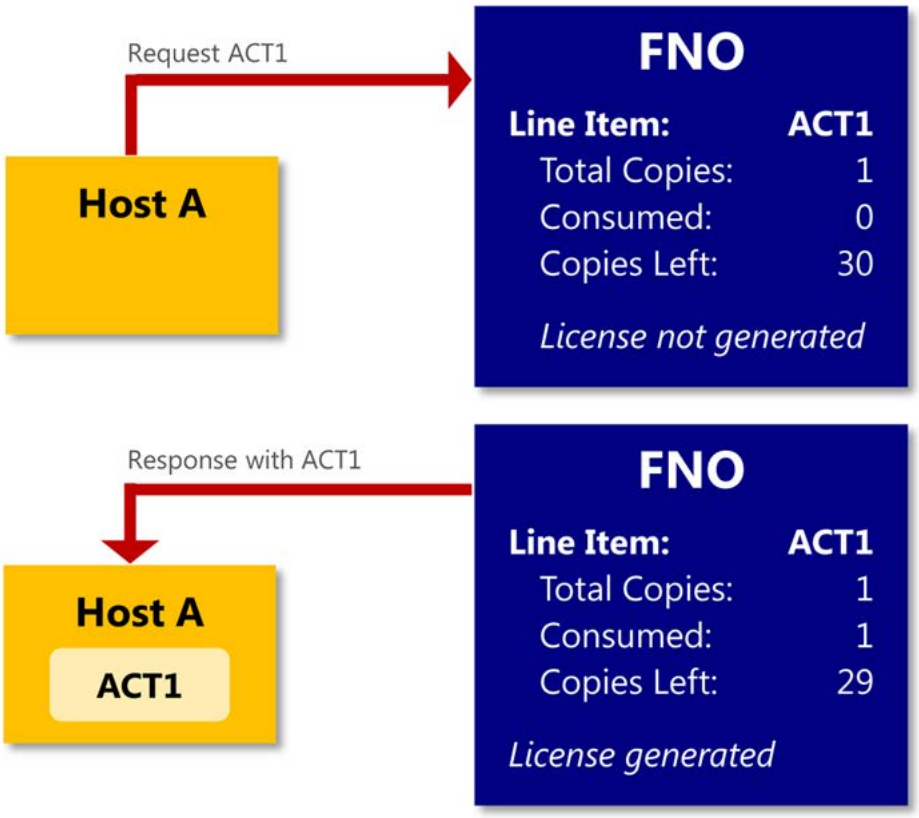
After the trusted storage license source is created, the license-enabled code can verify that functionality targeted for re-host is available on Host A. This can be done by acquiring licenses and reading license details from the trusted storage license source. See the corresponding steps in the previous 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

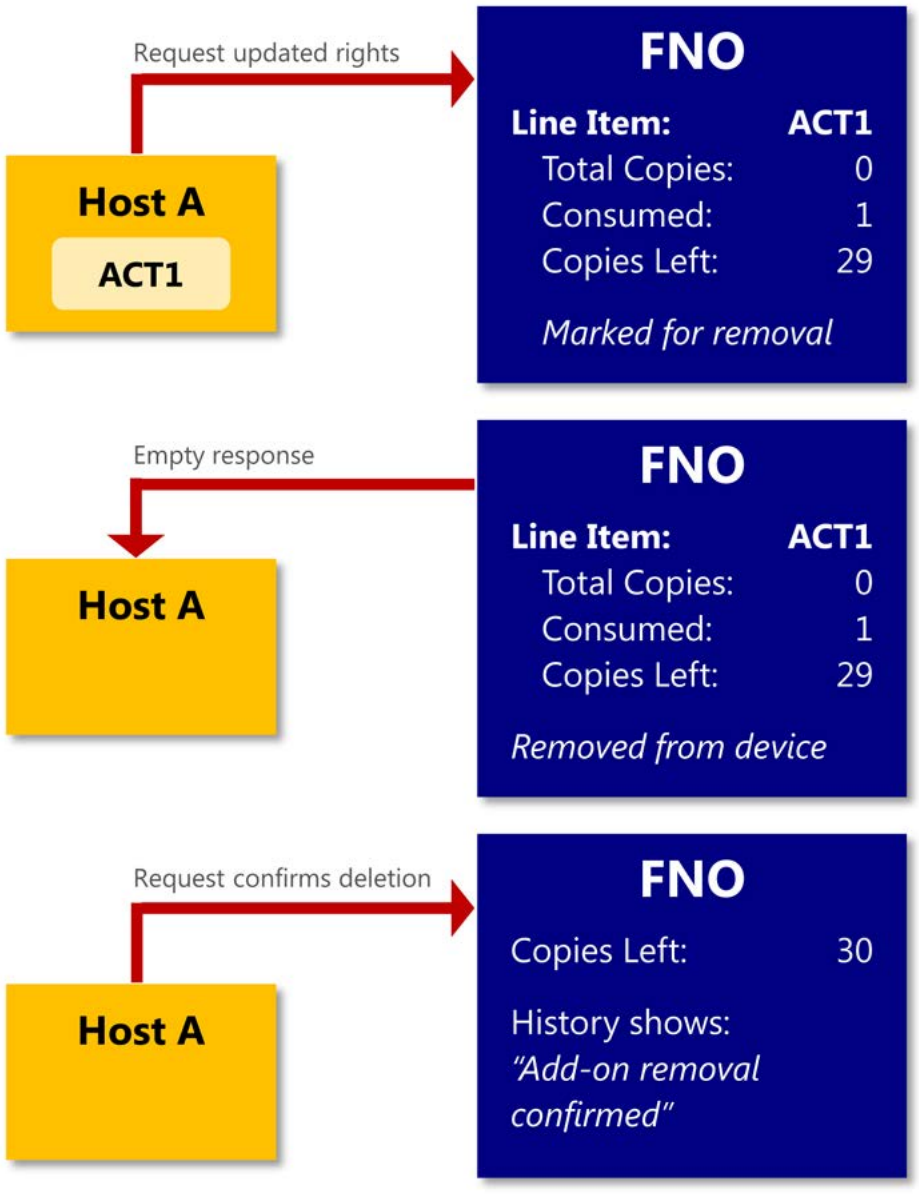
This step is similar to Step 2: Submit Capability Request from Host A to the Back-Office Server.

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/src/flxexamples`, in the source file `UsageCaptureClient.java`.

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
- Preparing 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, a "request" operation is assumed.

- **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 the `setRequestOperation` method in the `ICapabilityRequestOptions` interface 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 features to a capability request, call implement the `addDesiredFeature` method in the `ICapabilityRequestOptions` interface.

Usage-capture scenarios might also require an initial capability request that sends a rights ID value in order to register the client with the CLS server. The `addRightsId` method is used to add the rights ID value to the capability request. (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.)

Preparing 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 *FlexNet Operations Getting Started Guide for Usage Management*.

As described in the *FlexNet Operations Getting Started Guide for Usage Management*, create your producer identity data in FlexNet Operations, and then download the header file for the client identity to a location where your FlexNet Embedded Client Java XT toolkit can access it. (For the **UsageCaptureClient** example, download the file `IdentityClient.java` in the toolkit directory *`install_dir`*`/examples/client_samples/src/flxexamples`.) You must build the example with this identity.

The *FlexNet Operations Getting Started Guide for 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 Post-Usage-Capture: Managing Usage Data 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 Post-Usage-Capture: Managing Usage Data 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 Post-Usage-Capture: Managing Usage Data 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 section in this guide 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.

```
// Add trusted storage license source
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 "delete(IAdministration.AdminDeleteSourceOption.DELETE_TRUSTED_STORAGE)" method to delete 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 implementation uses the method `addRightsId` to add the specified rights ID to the capability request.

```
    // Get request options
    ICapabilityRequestOptions options = licenseManager.createCapabilityRequestOptions();

            {
            // Rights ID
            if (rightsId != null) {
               options.addRightsId(rightsId,  1);
            }
}
else if (operation == RequestOperation.REPORT && rightsId != null) {
   System.out.println("Report operation is not compatible with rightsid option.");
   return false;
```

```
        }
    return true;
    }
```

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 license server's instance ID on the Cloud Licensing Service. Note 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.

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 uses the `setRequestSetOperation` method with the `REPORT` option to generate the capability request. (In capped-usage scenarios, where a capability response is expected to return feature information to be processed into trusted storage, the `REQUEST` option is used instead.)

```
// Generate a capability request
if(generateRequest) {
   // Get request options
   ICapabilityRequestOptions options = licenseManager.createCapabilityRequestOptions();
   // Desired feature
   if (operation != RequestOperation.UNDO) {
      options.addDesiredFeature("survey", "1.0", 1);
   }
   // Request operation
   options.setRequestOperation(REPORT);
   // Generate request
   requestData = licenseManager.generateCapabilityRequest(options);
   if(storeRequest) {
      BinaryMessage.writeData(outputFile, requestData);
      System.out.println("Successfully generated request to " + outputFile + ".");
   }
if(processResponse && (responseData != null) && (responseData.length > 0)) {
```

```
// Process response
ICapabilityResponseData responseDetails =
       licenseManager.processCapabilityResponse(responseData);
```

Additionally, the **UsageCaptureClient** example uses the same `talkToServer` helper function to communicate with the CLS license server or the FlexNet Embedded local license server that the **CapabilityRequest** example uses to communicate with the back-office server. See Licenses Obtained from the Back-Office Server for details.

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 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.

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 Preparing FlexNet Operations and Post-Usage-Capture: Managing Usage Data for details.

To implement this type of usage capture, the code calls `setRequestSetOperation` with the `REQUEST` option to request a response from the license server. 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.

```
// Generate a capability request
if(generateRequest) {
   // Get request options
   ICapabilityRequestOptions options = licenseManager.createCapabilityRequestOptions();
   // Desired feature
   if (operation != RequestOperation.UNDO) {
      options.addDesiredFeature("survey", "1.0", 1);
   }
   // Request operation
   options.setRequestOperation(REQUEST);
   // Generate request
   requestData = licenseManager.generateCapabilityRequest(options);
   if(storeRequest) {
      BinaryMessage.writeData(outputFile, requestData);
      System.out.println("Successfully generated request to " + outputFile + ".");
```

```
    }
```

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.

```
if(processResponse && (responseData != null) && (responseData.length > 0)) {
   // Process response
   ICapabilityResponseData responseDetails =
       licenseManager.processCapabilityResponse(responseData);
   System.out.println("Successfully processed response.");
   if(responseDetails != null) {
      // Get response details
      getResponseDetails(responseDetails);
   }
```

The example then attempts to acquire the `survey` feature to determine whether the usage quota has been exceeded.

The **UsageCaptureClient** example uses the same `talkToServer` helper function to communicate with the CLS license server or the FlexNet Embedded local license server that the **CapabilityRequest** example uses to communicate with the back-office server. See Licenses Obtained from the Back-Office Server for details.

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 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.

When using the FlexNet Embedded local license server, specify the URL for the license server:

```
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 implementing the `setRequestSetOperation` method with the `UNDO` option in the `ICapabilityRequestOptions` interface. (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 `setCorrelationId(`*`correlationId`*`)` in the `ICapabilityRequestOptions` interface 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.

For “request” operations, in which the response containing features is processed into trusted storage, the last correlation ID used can be read using the `getCorrelationId` method in the `ICapabilityResponseData` interface.

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 *FlexNet Operations Getting Started Guide for Usage Management* and the Producer Portal help system 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 *FlexNet Operations Getting Started Guide for Usage Management* 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 using the `releaseLicense` method in the `ILicenseManager` interface.

The FlexNet Embedded API provides methods for reading attributes related to metered licenses and features. For example, the `isMetered` and `isMeteredReusable` methods in the `ILicense` interface respectively detect if an acquired license is metered or has the reusable attribute. (Similar `IFeature` methods indicate if a feature in a feature collection uses those attributes.) Related methods can give information about a license’s or feature’s “undo interval”. The **View** example illustrates how to use these methods 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. Refer to the source code in the *install_dir*`/examples/client_samples/src/flxexamples` directory, in the source file `View.java`.

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.

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 methods `getFeaturesFromBuffer` and `getFeaturesFromTrustedStorage`, 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.)

```
// Buffer feature info if present
if(licenseFile != null) {
   System.out.println("Reading data from " + licenseFile + ".");
   List<IFeature> bufferFeatures = licenseManager.getFeaturesFromBuffer(licenseFile, true);
   System.out.println("================================================");
   System.out.println("Features found in " + licenseFile);
   displayFeatures(bufferFeatures);
   System.out.println("================================================");
}

// Trusted storage feature info
List<IFeature> tsFeatures = licenseManager.getFeaturesFromTrustedStorage(true);
System.out.println("================================================");
System.out.println("Features found in trusted storage");
displayFeatures(tsFeatures);
System.out.println("================================================");

// Trial storage feature info
// Empty value for argument indicates diagnostic version is not used
```

```
List<IFeature> trialFeatures = licenseManager.getFeaturesFromTrials();
System.out.println("============================================");
System.out.println("Features found in trial storage");
displayFeatures(trialFeatures);
System.out.println("============================================");
```

Step 2: Examine Features in the Feature Collection

This step loops over the features in the collection and examines the attributes of each feature.

```
for(IFeature feature: features) {
   info = new StringBuilder();
   // Name
   info.append(feature.getName() + " ");
   // Version
   info.append(feature.getVersion() + " ");
   // Expiration
   Date expiration = feature.getExpiration();
   if(expiration != null) {
      info.append(format.format(expiration));
   } else {
      info.append("permanent");
   }
   // Count
   if(LicensingConstants.UNCOUNTED == feature.getCount()) {
      info.append(" uncounted");
   } else {
      info.append(" " + Long.toString(feature.getCount()));
   }
   // Hostid(s)
   Map<HostIdType, List<String>> hostIds = feature.getHostIds();
   if(hostIds.size() > 0) {
       int numOfHostIds = 0;
     String hostIdString = "";
      for(Map.Entry<HostIdType, List<String>> type : hostIds.entrySet()) {
              if(type.getKey() != HostIdType.ANY) {
                    hostIdString += type.getKey().toString() + "=";
              }
             boolean first = true;
             String value = "";
             int numOfType = 0;
             for(String id : type.getValue()) {
               if(first) {
                  first = false;
                  value = id;
               }
               else {
                 value += "," + id;
               }
               numOfType++;
               numOfHostIds++;
           }
           if(numOfType > 1) {
               hostIdString += "(" + value + ")";
           }
```

```
        else {
          hostIdString += value;
        }
               hostIdString += " ";
             }
               if(numOfHostIds == 1) {
                  info.append(" Hostid=" + hostIdString.trim());
             }
             else {
               info.append(" Hostids=[" + hostIdString.trim() + "]");
             }
}

// Notice
if(feature.getNotice() != null) {
   info.append(" NOTICE=\"" + feature.getNotice() + "\"");
}
// Issuer
if(feature.getIssuer() != null) {
   info.append(" ISSUER=\"" + feature.getIssuer() + "\"");
}
// Vendor string
if(feature.getVendorString() != null) {
   info.append(" VENDOR_STRING=\"" + feature.getVendorString() + "\"");
}
// Serial number
if(feature.getSerialNumber() != null) {
   info.append(" SN=\"" + feature.getSerialNumber()  + "\"");
}
// Issued date
Date issued = feature.getIssued();
if(issued != null) {
   info.append(" ISSUED=\"" + format.format(issued)  + "\"");
}
// Start date
Date startDate = feature.getStartDate();
if(startDate != null) {
   info.append(" START=\"" + format.format(startDate) + "\"");
}
// Metered REUSABLE
if (feature.isMeteredReusable()) {
   info.append(" MODEL=metered REUSABLE");
} // Metered
else if (feature.isMetered()) {
   if (feature.getMeteredUndoInterval() > 0) {
      info.append(" MODEL=metered UNDO_INTERVAL=" + feature.getMeteredUndoInterval());
   }
   else {
      info.append(" MODEL=metered");
   }
}
// Acquisition status
AcquisitionStatus status = feature.getAcquisitionStatus();
if(status != null && status == AcquisitionStatus.VALID) {
   if (feature.isMetered()) {
      long available = feature.getAvailableAcquisitionCount();
```

```
            if (available == 0) {
               info.append(", Entire count consumed");
            }
            else if (available == LicensingConstants.UNCOUNTED) {
               info.append(", Valid for acquisition");
            }
            else {
               info.append(", Available for acquisition: " + available);
            }
         }
         else {
            info.append(", Valid for acquisition");
         }
      }
      else {
         info.append(", Not valid for acquisition: " + getAcquisitionStatusString(status));
      }

      System.out.println(info.toString());
```

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, 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 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 FlexNet Embedded license source. In particular, FlexNet Embedded supports unserved, uncounted certificates, with a restricted set of keywords and hostid types.

The following sections describe the 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 Java XT 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 FlexNet Embedded 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 `-java` switch to generate the `IdentityClient.java` file that you will compile into your FlexNet Embedded code. (An `IdentityClient.java` 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, FlexNet Embedded Client Java XT includes the **Lmflex** example. The `lmflex` 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="..."
```

For implementation details, refer to the source code in the *`install_dir`*`/examples/client_samples/src/flxexamples` directory, in the source file `Lmflex.java`. The identity data (`IdentityClient.java`) 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
licenseManager.addCertificateLicenseSource(legacyCertificateFile);
```

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:

```
Reading certificate from legacy.lic.
Successfully acquired "f1", version 1.0, 1 count.
Successfully acquired "f2", version 1.0, 1 count.
```

Exceptions thrown by the `acquireLicense` 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—a FlexNet Embedded 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.)

```
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 `SERVER_5` (server instance 5):

```
// Add default trusted storage license source
licenseManager.addTrustedStorageLicenseSource();
```

```
// Add trusted storage license source for SERVER_5
licenseManager.addTrustedStorageLicenseSource(LicenseServerInstance.SERVER_5);
```

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, use the following method in the `ICapabilityRequestOptions` interface to define the ID as an option:

```
licenseManager.createCapabilityRequestOptions(LicenseServerInstance.serverInstanceID)
```

The following shows a sample implementation of this functionality (in this case, to specify server instance 5):

```
// Create request option for SERVER_5
ICapabilityRequestOptions capOptionsServer5 =
        licenseManager.createCapabilityRequestOptions(LicenseServerInstance.SERVER_5);
```

The following code sample identifies which server instance ID is associated with a given set of capability request options:

```
// Get instance associated with this capability request option
                    LicenseServerInstance instance = capOptionsServer5.getServerInstance();
```

Processing the Response from a Server Instance

Use the `licenseManager.processCapabilityResponse(response, instance)` method in the `ILicenseManager` interface to process the capability response from a given server—identified by the server instance ID used in the method—into the trusted-storage location and license source with that same ID.

The following shows a sample implementation of this functionality, including how to retrieve the server instance ID from the capability response for use in response-processing:

```
// Set options, generate request, send request, and get response
// ...
byte[] response = null;     // Get response

// Process response, determine what instance
ICapabilityResponseData responseData = licenseManager.getResponseDetails(response);
```

```
instance = responseData.getServerInstance();
// Process response to appropriate instance
licenseManager.processCapabilityResponse(response, instance);
```

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 `licenseManager.processCapabilityResponse(`*`response`*`, `*`instance`*`)`.

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 Client Java 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.

6

Utility Reference

The following utilities are included in the FlexNet Embedded Client Java XT toolkit. You use these utilities (located in the toolkit's `bin/tools` directory) to prepare your toolkit for production use, to create and process license rights, and to test various implementation scenarios.

The 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 code enabled with FlexNet Embedded functionality.

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 the back office for digitally signing license rights; the client–server identity is used for served licenses; and the client-identity data is used by FlexNet Embedded to validate license rights and to perform other validation operations.

Purpose

The Publisher Identity utility enables producers to create producer-specific, API-compatible, identification data in binary format. This binary data will be passed as a buffer to the FlexNet Embedded Client Java XT method, `LicensingFactory.getLicensing`, for initializing the main `ILicensing` interface and for validation and security purposes.

The Publisher Identity utility is provided to enable each producer to create unique identity files that are used to sign FlexNet Embedded license files, trial rights, and capability responses. 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 a user interface. The command takes optional arguments that determine where the identity files will be written, whether the utility runs in console mode (no GUI), and other specifications.

```
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. The following command (with no arguments) presents a user interface to generate the identity files:

```
pubidutil
```

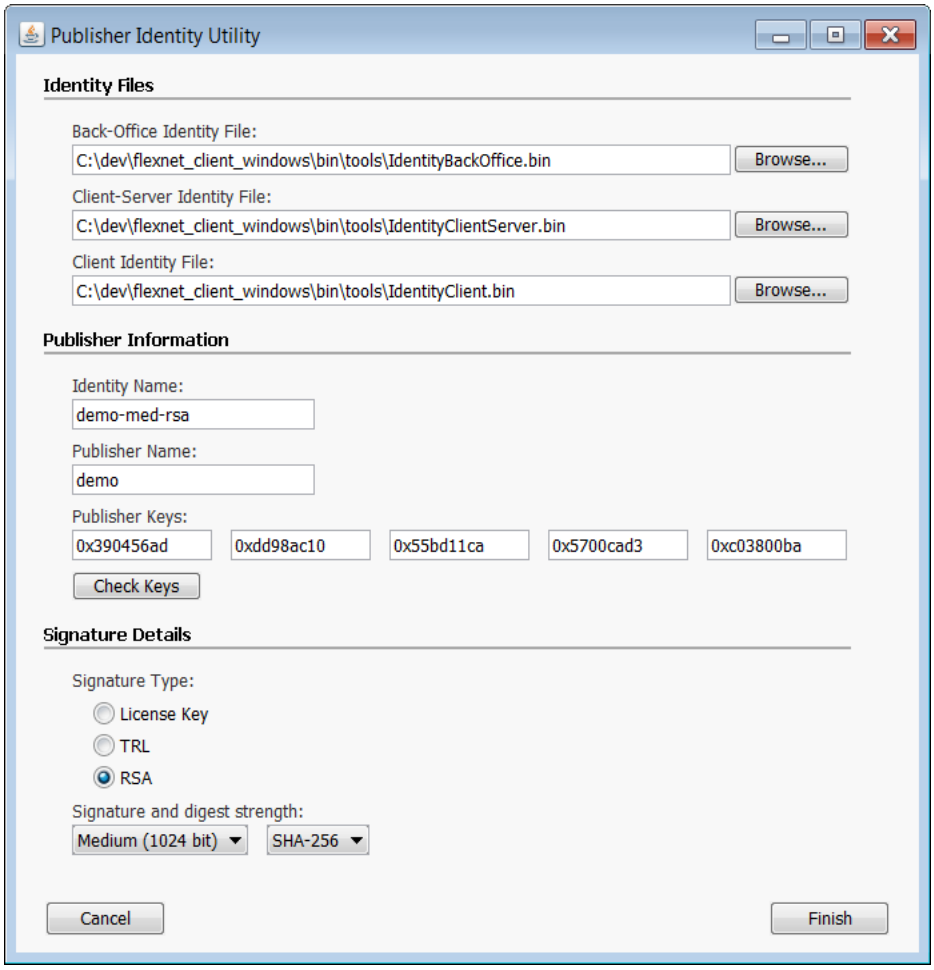


Figure 6-1: Pubidutil in GUI Mode

To avoid the use of the GUI, 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, you will most likely need to update the client-identity buffer data in your license-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 Java 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 as well as capability and response envelopes.
- **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, the client–server identity is used with served licenses, and the client identity file is used in license-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 in the next section, converts the binary client-identity data into Java 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 FlexNet Embedded binary license file, capability request, capability response, or trial file created with `licensefileutil`, `caprequestutil`, `capresponseutil`, or `trialfileutil`, respectively.

The print-binary utility also displays the contents of a binary identity file created with `pubidutil`, and optionally converts it to Java-compatible format for use in your compiled license-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 a binary license file's contents and validate any signatures contained in it, 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 Java code (typically as a hard-coded array to be passed to FlexNet Embedded functions that require the binary data), use the `-java` switch (and optionally the `-package` switch, for a Java package name):

```
printbin -java IdentityClient.bin -package flxexamples
```

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 should look similar to the following:

```
package flxexamples;

import java.util.Base64;
/*
    Publisher Identity Version = 0
    Identity Name = demo-rsa
    Publisher Name = demo
    Publisher Key = ...
    Signature Type = RSA
    Signature Strength = 2
*/
public class IdentityClient {

    public static byte[] IDENTITY_DATA = null;
```

```
    static {
            String identity_data_base64 =

                "AAAZiAAAQnf/////AAAACwByAAAAAAAAAAAMAIACZGVtbwAAAAAMAG8CZGVtbwAAAAALAG4AndzQ"+
                "gAAAAAsAbgCZqoMJAAAACwBuADOZWJYAAAALAG4AhpcTIAAAAAsAbgALFl3LAAAACwBsAAAAAAMA"+
                "AAALAFMAAAAAAgAAC6MAcwvr0UcCNd4BJl9HR2SfPI4E4+0CayEOYQMyPDoDCMLOBGQTq17kQeIB"+
                "/ezLIQXwZQM2AJVDCCtpBAyOXMLFH0gA0QzJmBRdmALHAnX6d3TaANCXzn8kyzgCBvD4BODHZgJs"+
                "MtFKHoqtAVccHGqBg3YBMlpgmuMP/wDwUv0MvNOjA/4Zw1outy4Ev6JdhipXyQKvXfHijkDnAa7V"+
                ...
                "FpgysJoFdUAvwrrU00UKL8uYO05Ar9LtHHoy+mJS0mEPa9Iy6OA="+
                "";

                IDENTITY_DATA = Base64.getDecoder().decode( identity_data_base64 );
            }

    }
```

You can then compile the client-identity data in your license-enabled code.

You can use this technique if your version of FlexNet Operations does not directly export Java-compatible identity data. First, download your binary client-identity data `IdentityClient.bin`, and then run the following:

```
printbin -java IdentityClient.bin -package flxexamples -o IdentityClient.java
```

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 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 `IDENTITY_DATA` when using the `-java` 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).

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 6-1 ▪ 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.
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`).

Table 6-1 ▪ Values for Hostid Types Used to Restrict Hostid Detection

Value for Hostid Type Restriction	Detects...
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=00020000501010002f403
```

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.

The following show its usage:

```
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, run `capserverutil`, using a 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 FlexNet Embedded Client Java XT *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]
    [-activate 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 listed in the capability request is always considered the main hostid. Each subsequent hostid is considered secondary. When license reservations are used, the first secondary hostid is used in the reservation search. (For more information, see in the Secondary Hostids section of the Using the 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 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 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.
  - The acquisition ID identifies the resource that was acquired.
  - The enterprise ID identifies the end-user account on behalf of the acquisition performed.

  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

    Note 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.
- *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 millliseconds. (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 secure profile utility `secureprofileutil` configures existing client-identity 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 Identifying the Device User for more information.) Before using this utility, you must obtain your producer client-identity data file (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 the 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 that you obtained from FlexNet Operations or that you 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 file 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.)

Index

D

E

F

H

I

L

M

O

P

R

S

T

U

V