FlexNet Embedded 2023.03

License Server Producer Guide

Legal Information

Book Name: FlexNet Embedded 2023.03 License Server Producer Guide

Part Number: FNE-2023.03-LSG00

Product Release Date: March 2023

Copyright Notice

Copyright © 2023 Flexera Software

This publication contains proprietary and confidential information and creative works owned by Flexera Software and its licensors, if any. Any use, copying, publication, distribution, display, modification, or transmission of such publication in whole or in part in any form or by any means without the prior express written permission of Flexera Software is strictly prohibited. Except where expressly provided by Flexera Software in writing, possession of this publication shall not be construed to confer any license or rights under any Flexera Software intellectual property rights, whether by estoppel, implication, or otherwise.

All copies of the technology and related information, if allowed by Flexera Software, must display this notice of copyright and ownership in full.

FlexNet Embedded incorporates software developed by others and redistributed according to license agreements. Copyright notices and licenses for these external libraries are provided in a supplementary document that accompanies this one.

Intellectual Property

For a list of trademarks and patents that are owned by Flexera Software, see https://www.revenera.com/legal/intellectual-property.html. All other brand and product names mentioned in Flexera Software products, product documentation, and marketing materials are the trademarks and registered trademarks of their respective owners.

Restricted Rights Legend

The Software is commercial computer software. If the user or licensee of the Software is an agency, department, or other entity of the United States Government, the use, duplication, reproduction, release, modification, disclosure, or transfer of the Software, or any related documentation of any kind, including technical data and manuals, is restricted by a license agreement or by the terms of this Agreement in accordance with Federal Acquisition Regulation 12.212 for civilian purposes and Defense Federal Acquisition Regulation Supplement 227.7202 for military purposes. The Software was developed fully at private expense. All other use is prohibited.

Contents

1 Introduction . . . 11

Local License Servers vs. License Servers Hosted in the Cloud . . . 11

What's in this Guide . . . 12

Product Support Resources . . . 13

Contact Us . . . 14

2 Getting Started . . . 15

About the License Server Software Package . . . 15

License Server Requirements . . . 15

Downloading the Software Package . . . 16

License Server Components . . . 16

Additional Components . . . 19

Preparing and Deploying the License Server: Producer Experience . . . 19

Enabling Secure Anchoring . . . 21

Requirement. . . . 21

Enable Secure Anchoring . . . 21

Determining Deliverables Needed to Enable Outgoing HTTPS . . . 21

Determining Administrative Security Policies . . . 22

Generating the Producer Settings File . . . 22

Policy Settings Reference . . . 22

Requirements. . . . 22

Generate the Policy Settings File . . . 23

Determining Administration Tools for the Enterprise . . . 23

Entitling the Enterprise Customer Account . . . 23

Delivering the License Server . . . 24

3 Installing and Running the License Server . . . 27

Installing and Starting the License Server . . . 27

Prerequisites . . . 28
Install and Start on Windows . . . 28
Install and Start on Linux . . . 35
Setting the Server Time Zone . . . 42
Logging Functionality . . . 42
Editing Local Settings Post-Installation . . . 43
Post-Installation Configuration on Windows . . . 43
Post-Installation Configuration on Linux . . . 45
Configuring the Cipher Choice Mechanism . . . 52
Understanding Hostids . . . 53
Hostid Types . . . 54
Specifying a Server Hostid . . . 54
Retrieving Server Hostids . . . 55
Activating the License Rights for the Server . . . 56
Using the REST API Option . . . 56
Using the Administration Tools . . . 57
Granting Client Requests . . . 57
Basic Capability-Request Processing . . . 57
Usage Capture . . . 58
Upgrading the License Server . . . 58
Uninstalling the License Server . . . 59
Uninstall the License Server Service . . . 59
Uninstall the License Server Running in Console Mode . . . 61

4 License Server REST APIs . . . 63

Accessing the REST APIs . . . 64
Base URLs . . . 64
Sample cURL Command . . . 65
Managing Large Amounts of Returned JSON Data . . . 66
Writing a REST API Client for a Secured License Server . . . 66
Using the Base URL . . . 66
Authorizing Access to Secured REST APIs . . . 67
Creating User Accounts . . . 67
APIs for Basic License Server Operations . . . 68
API Summary . . . 68
Viewing Instance Details . . . 69
Viewing Instance hostids . . . 70
Processing Requests from Clients . . . 72
Sending Requests to the Back Office . . . 72
Activating Licenses on the Server . . . 74
Suspending or Resuming Operations . . . 75
Initiating On-demand Synchronization . . . 75
APIs to Manage Clients . . . 76
Viewing All Clients . . . 77
Viewing All Current Clients and Their Features . . . 80

- Retrieving Client Data as NDJSON Objects . . . 83
- Viewing a Specified Client Record . . . 85
- Viewing Features for a Specified Client Record . . . 86
- Removing a Specified Client . . . 89
- Viewing Client Record History for Specified Hostid . . . 90
- Viewing Client Record History Including Features for a Specified hostid . . . 91

APIs to Obtain Feature Information . . . 93
- Viewing All Features . . . 94
- Viewing A Summary Of Features . . . 96
- Retrieving Feature Data as NDJSON Objects . . . 98
- Viewing Only Active Features . . . 100
- Viewing Features in an Expiration Grace Period . . . 100
- Viewing Number of Features . . . 102
- Viewing Specified Feature . . . 102
- Viewing Feature Usage . . . 104
- Listing Overages . . . 106

APIs to Manage Named License Pools . . . 107
- API Summary . . . 108
- Uploading the Model Definition . . . 109
- Viewing the Model Definition . . . 110
- Deleting the Model Definition . . . 111
- Viewing License Pools . . . 112
- Viewing a Specified License Pool By ID . . . 115
- JSON Response Details . . . 116

APIs to Manage Reservations . . . 117
- API Summary . . . 118
- Managing Reservation Groups . . . 119
- Managing a Specific Reservation Group . . . 122
- Managing All Reservations in a Specific Reservation Group . . . 124
- Managing a Specific Reservation in a Reservation Group . . . 127
- Managing a Specific Reservation Entry . . . 129

APIs to Show Statistics . . . 130
- Viewing Instance Version . . . 131
- Viewing Success of Database Connection . . . 132

APIs to Manage Instance Configuration . . . 133
- Managing Instance Policies . . . 134
- Viewing Configuration Metadata . . . 135

APIs to Perform Offline Binary Operations . . . 137
- API Summary . . . 137
- Generating an Offline Activation Request . . . 138
- Generating an Offline Capability Request . . . 139
- Processing an Offline Capability Response . . . 139
- Generating an Offline Sync Message . . . 141
- Processing an Offline Sync Acknowledgment . . . 141

API to View Binding-Break Status . . . 142

APIs to Manage Administrative Security 143
APIs to Authorize Access to Secured Endpoints 143
APIs to Manage User Accounts 145
API for Enabling JSON Security for the Cloud Monetization API 152
License Server Administration Security vs JSON Security 153
HTTPS Protocol Recommended 154
Implementing JSON Security for the Cloud Monetization API 154
“/rest_licensing_keys” API Details 157
Sample Code Implementations for Setting Up JSON Security 159
Online REST API Reference Documentation Available 162
Exception Handling 162
List of Exceptions 163

5 More About Basic License Server Functionality 169

Synchronization Operations 169
Notes about Policy Settings for Synchronization 170
Online Synchronization to the Back Office 171
On-Demand Synchronization 172
Offline Synchronization to the Back Office 172
Disabling Synchronization to the Back Office 177
Synchronization From the Back Office 178
Failover Using Synchronization to FlexNet Embedded 178
Allocating Licenses Using Named License Pools 183
Named License Pool Terminology 184
Introduction to the Model Definition 184
Model Definition Components 185
Server Behavior When Distributing Feature Counts to Named License Pools 195
Server Behavior when Assigning Features to Clients 198
Named License Pools vs. Reservations 200
Limitations of Named License Pools 203
License Reservations 203
Overview of Reservation Types 204
Reservation Hierarchy 206
Managing Reservations 207
License Allocation on the Server 211
Processing the Capability Request When Reservations Are Used 213
Reservation Considerations and Limitations 216

6 Advanced License Server Features 217

Producer-defined Hostids 217
Configuring Support for the Producer-defined Hostid 218
Providing the Shared Library for Hostid Retrieval 218
Using the Producer-defined Hostid 220
Secure Anchoring 221
Prerequisites 221

Enabling a Secure Profile . . . 221
Clone Detection Reporting . . . 221
About the Clone Detection Process . . . 222
Requirements. . . . 222
Generate the Clone Detection Report in FlexNet Operations . . . 223
Report Alternatives . . . 223
Considerations and Limitations . . . 224
Binding-Break Detection with Grace Period . . . 224
Enabling the Binding-Break Detection Feature . . . 226
Providing the Shared Library to Retrieve the Binding Elements . . . 226
How the Binding-Break Detection Works . . . 229
Repairing the Broken Binding . . . 233
Considerations and Limitations . . . 233
Trusted Storage Backup and Restoration . . . 233
Performance Implications . . . 234
Enabling Backups of Trusted Storage . . . 234
Overview of the Backup Process. . . . 234
Running a Trusted Storage Restoration . . . 234
Outgoing HTTPS. . . . 235
Default Server Configuration . . . 236
Enabling Server for HTTPS When Another Certificate Is Used . . . 236
Incoming HTTPS . . . 239
Step 1: Obtain Certificate . . . 240
Step 2: Enable Access to “server” Certificate . . . 240
Step 3: Define Scope of HTTPS Communications. . . . 242
Proxy Support for Communication with the Back Office . . . 242
Configuring the License Server for Proxy Support . . . 243
Obfuscating the Proxy Password . . . 244
Extended Hostids . . . 244
Scenario Using Extended Hostids . . . 245
Enabling Support for Extended Hostids . . . 245
Setting and Using Extended Hostids . . . 246
Trusted-Storage Reset . . . 247
Determining Whether Records Pending Synchronization Exist . . . 247
Resetting Trusted Storage. . . . 248
Enabling a Forced Trusted-Storage Reset . . . 249
License Checkout: Support for Special Capability-Request Options . . . 250
Incremental Capability-Request Processing . . . 250
Granting All Available Quantity for a Feature . . . 253
Feature-Selector Filtering . . . 254
Vendor Dictionary Data . . . 257
Capability Preview . . . 257
Usage Capture and Management . . . 263
Synchronization Required . . . 263
For More Information. . . . 263

Administrative Security 264
Overview: Deploying a License Server with Security Enabled 264
About Administrative Security 265
About License Server Administrator Tools 269
Enabling Administrative Security on a Local License Server 269

7 Producer Tools **271**

Publisher Identity Utility 272
Purpose 272
Usage 272
Entering Your Identity Data 274
Updating Your Identity Data 274

Print Binary Utility 275
Viewing Contents 275
Viewing Contents and Validating Signatures 276
Displaying Binary in Compiler-Readable Format 276
Conversion to Base 64 Format in FlexNet Embedded 276
Additional printbin Switches 277

Identity Update Utility 277
Usage 278
Device Hostid Types Used to Restrict Hostid Detection 279
Example Identity Update 280

License Conversion Utility 280

Trial File Utility 281

Capability Server Utility 282
Considerations for Using the Utility 282
Usage 283
Starting and Stopping the Capability Server Utility 283
About License Templates 283
Endpoint for Sending Capability Requests to the Utility 285

Capability Request Utility 286

Capability Response Utility 289

Secure Profile Utility 291
Viewing Available Security Profiles 291
Enabling Secure Anchoring 292

License Server Configuration Utility 292
Policy Settings Reference 293
Generating the Policy Settings 293
Distributing this File 295
Updating the File After Server Installation 295

8 Troubleshooting Reference **297**

A Reference: Policy Settings for the License Server **299**

B Effects of Special Request Options on Use of Reservations 315
Scenario Assumptions 316
Incremental Option Enabled 316
Partial-Checkout Option Enabled 318
Both Incremental and Partial-Checkout Options Enabled 319

C SysV Alternative for Installation on Linux 323
Files Required for License Server Installation Using SysV 323
Prepare for Linux Installation 324
Install and Start as a Linux Service 325
Install and Start in Console Mode 326
Edit Local Settings Post-Installation 326
When Server Runs as a Service 326
When Server Runs Console Mode 329
Uninstall the License Server 329
Uninstall the License Server Service 329
Uninstall the License Server Running in Console Mode 330
Manage the License Server Service: Command Summary 330

D Model Definition Grammar for Named License Pools 333
Model Definition Grammar and Syntax—EBNF 333
Parser 333
Lexer 335
Use Case Examples and Their Model Definitions for Named License Pools 336
Background Information for Use Cases 336
Use Case: Simple Allow List 337
Use Case: Simple Block List 338
Use Case: Sharing Counts Between Business Units 338
Use Case: Assigning Extra Counts To Business Unit 339
Use Case: Exclusive Use of Feature Counts for Business Unit 339
Use Case: Exclusive Use of Feature Counts for Business Unit With Exception of Specific Clients 340
Use Case: Exclusive Use of Feature Counts for Business Unit and Specified Clients from other Business Units . . . 341
Use Case: Assigning Features Based on Combined hosttype and hostname Properties 341
Use Case: Assigning Features Based on Vendor String Property 342
Use Case: Device-specific Handling—Sharing Feature Counts Based On Hosttype 344
Use Case: Named License Pool Receiving Entire Remaining Feature Count 345
Use Case: Using Regular Expression to Allocate License Counts 345
Use Case: Making Feature Counts Available to Multiple Business Units 347
Use Case: Letting Server Specify Counts 348
Use Case: Accumulating Counts from Multiple Named License Pools (“Continue” Action) 349
Model Definition Examples for Reservations Converted to Named License Pools 350
Scenario: Counts Reserved By Hostid 350
Scenario: Server-specified Counts 350

E Logging Functionality on the Local License Server 353

Logging Style 353

Custom Log Configurations 354

Integration of License Server Logging With External Systems 354

Graylog 355
Elastic Stack 355
logz.io 356
Example Configurations 356

F Consuming Streaming Content 365

G Workflow Example for Producer-Defined Binding 367

Adding Binding 368
Simulating a Binding Break 371
Repairing a Binding Break 371

1

Introduction

The license server described in this guide provides functionality for serving and monitoring a counted pool of licenses for intelligent devices or software clients that use code that is license-enabled through either FlexNet Embedded language-based API implementations or a REST interface called the Cloud Monetization API. The license server is designed to administer and enforce a pool of licenses within a customer's enterprise, report license usage to the back office (FlexNet Operations), and provide served-license status information.

This book provides guidance on how to prepare and deploy the license server to your enterprise customers and how to manage the license server from both a producer and license-server administrator perspective. (Note that the *FlexNet Embedded License Server Administration Guide* provides similar information for the license server administrator only.)

For information about implementing client code that is licensed through FlexNet Embedded language-based API calls to the license server, see the *FlexNet Embedded Client C SDK User Guide* or the FlexNet Embedded client SDK user guide appropriate for your programming language. For information about client code that is licensed through REST API calls to the server, see the *Cloud Monetization API User Guide*.

For more information about using FlexNet Operations, see your FlexNet Operations documentation.

Local License Servers vs. License Servers Hosted in the Cloud

The license server can be deployed as a local license server at a customer site or as a Cloud Licensing Service (CLS) instance, hosted in the Revenera Cloud. Both the local license server and the CLS instance provide the same functionality.

Note that the preparation and deployment instructions in this book focus mainly on the local license server, basically because the setup and deployment of the CLS instance and some of its administration are performed in FlexNet Operations. However, this book directs you to the appropriate FlexNet Operations and FlexNet Embedded documentation for CLS instance information when needed.

Finally, because both license servers use the same REST endpoints to perform administrative operations, license server administration tools provided by the producer (for example, `flexnetlsadmin`) can be used to manage important features, such as named license pools, license reservations, security, and enterprise user accounts, for both deployment types. This book notes these areas where functionality and procedures apply to both server types.

For more information about deploying CLS license servers to your customers and administering the license server, refer to these resources:

- Your Revenera representative for information about purchasing Cloud Licensing Service privileges
- *FlexNet Operations Getting Started Guide for Cloud Licensing Service* in the *FlexNet Operations User Guide*
- *Managing a CLS Instance* and other appropriate chapters in the *FlexNet Embedded License Server Administration Guide*

What's in this Guide

The *FlexNet Embedded License Server Producer Guide* includes the following chapters:

Table 1-1 ▪ *FlexNet Embedded License Server Producer Guide*

Topic	Content
Introduction	Provides an overview of the book and list of conventions used in the book's contents.
Getting Started	Helps you get started with basic information about the FlexNet Embedded local license server software package and an overview the tasks needed to prepare the license server for distribution.
Installing and Running the License Server	Provides instructions on installing and getting the FlexNet Embedded local license server up and running.
License Server REST APIs	Describes the REST endpoints that you can access to administer the license server or to perform REST-driven licensing.
More About Basic License Server Functionality	Describes basic functionality that you can enable for the FlexNet Embedded local license server and that might require additional setup.
Advanced License Server Features	Describes special features that you can enable for the FlexNet Embedded local license server and that might require additional setup.
Producer Tools	Describes tools that help you test and prepare the license server for production.
Troubleshooting Reference	Lists some common problems you might encounter with the license server and provides workarounds.
Reference: Policy Settings for the License Server	Provides a reference to the various settings that you can include in your policy settings file, `producer-settings.xml`, for the license server.

Table 1-1 ▪ *FlexNet Embedded License Server Producer Guide* (cont.)

Topic	Content
Effects of Special Request Options on Use of Reservations	Describes how certain options you can include in a capability request affect how reservations are processed when satisfying feature requests.
SysV Alternative for Installation on Linux	Describes how to install the local license server as a Linux service using the SysV init runlevel system.
Model Definition Grammar for Named License Pools	Provides a full description of the model definition grammar that is used for creating named license pools. Also includes use cases and sample model definitions that help license administrators prepare the model definition to suit their requirements.
Logging Functionality on the Local License Server	Describes the logging functionality available for the local license server, including logging style, custom log configurations, and how to integrate license server logging with external systems.
Consuming Streaming Content	Describes the `/clients.stream` and `/features.stream` APIs which should be used for querying large tables.

Product Support Resources

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

- Revenera Community
- Revenera Learning Center
- Revenera Support

Revenera Community

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

https://community.revenera.com

Revenera Learning Center

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

https://learning.revenera.com

Revenera Support

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

https://community.revenera.com

Contact Us

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

http://www.revenera.com

You can also follow us on social media:

- Twitter
- Facebook
- LinkedIn
- YouTube
- Instagram

2

Getting Started

This chapter helps you get started with basic information about the FlexNet Embedded local license server software package and an overview of the tasks needed to prepare the license server for distribution. The chapter includes these sections:

- About the License Server Software Package
- Preparing and Deploying the License Server: Producer Experience
- Enabling Secure Anchoring
- Determining Deliverables Needed to Enable Outgoing HTTPS
- Generating the Producer Settings File
- Determining Administration Tools for the Enterprise
- Entitling the Enterprise Customer Account
- Delivering the License Server

About the License Server Software Package

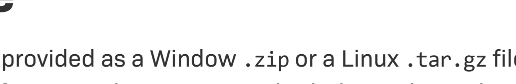
The following describes basic information about the software package:

- License Server Requirements
- Downloading the Software Package
- License Server Components
- Additional Components

License Server Requirements

The complete list of requirements for installing and running the license server is found in the FlexNet Embedded *License Server Release Notes* for the current release.

Important* ▪ *Due to potential incompatibilities with trusted storage and other stored data, the enterprise customer cannot downgrade their current license server to a previous version.



Downloading the Software Package

The FlexNet Embedded local license server software package is provided as a Window `.zip` or a Linux `.tar.gz` file that you download from the Product and License Center. The software package supports both the 32-bit and 64-bit architectures:

- flexnetls-i86_windows-*version*.zip
- flexnetls-x64_windows-*version*.zip
- flexnetls-i86_linux-*version*.tar.gz
- flexnetls-x64_linux-*version*.tar.gz

Task ***To download the license server software package***

1. From the Product and License Center, download the appropriate license server software package for your platform.
2. Unpack the files into an installation directory.

Throughout the remainder of this guide, the term *installation directory* refers to the location in which you unpacked the license server software package.

License Server Components

The following describes the license server components for your FlexNet Embedded local license server.

Table 2-1 ▪ Components in Producer's License Server Software Package

Directory/Files	Purpose
"bin\tools" directory	Directory containing tools used to test and prepare the license server for distribution, including the `flexnetlsconfig(.bat)` utility used to generate the `producer-settings.xml` file required to run the license server.
"enterprise" directory	Directory containing optional license-server administration tools that you might choose to distribute to the enterprise.
flexnetlsadmin.sh(.bat) flexnetlsadmin.jar	Files used to run the FlexNet License Server Administrator, a command-line tool to help the enterprise manage operations on the license server.

Table 2-1 ▪ Components in Producer's License Server Software Package (cont.)

Directory/Files	Purpose
serverofflinesynctool (.bat) backofficeonlinesynctool (.bat)	Tools that allow the license server to perform offline synchronization to the back office. (The license server supports offline synchronization by default, but you can choose not to provide this functionality.) These tools require the `flxPublicTools.jar`, `EccpressoAll.jar`, and `commons-codec-1.9.jar` files found in the `\lib` directory.
"examples" directory	Directory containing example code and other files to help you implement special license-server functionality, such as producer-defined hostids (see Producer-defined Hostids) and binding-break detection with (see Binding-Break Detection with Grace Period).
"lib" directory	Directory containing the libraries needed to run the producer-related tools that test and prepare the license server.
"server" directory	Directory containing the files needed to run the license server. You must include these files in the license server package distributed to the enterprise.
flexnetls.jar	FlexNet Embedded local license server Java executable.
flexnetls(.bat)	The Linux shell script or Windows batch file used to install, start, and control the license server service. You can use this as a sample to modify or write your own install script. (Your installer must place this file in the same directory as `flexnetls.jar`.) ***Note ▪*** *The Linux shell script version of "flexnetls" is used only when the license server is installed as a Linux service using a SysV init runlevel system. However, the standard service installation method described in this book uses systemd. If you want to provide a SysV installation for your customers, see Appendix C, SysV Alternative for Installation on Linux.*
regid.2009-06.com(...).swidtag	The software ID tag file for use with software asset management tools.
flexnetls.settings	(*Windows only*) The local settings file that license server administrator can edit to define the port number, log level, and other parameters used to run the license server as a service. (Your installer must place this file in the same directory as `flexnetls.jar`.)

Table 2-1 ▪ Components in Producer's License Server Software Package (cont.)

Directory/Files	Purpose
configure	(*Linux service with SysV installation only*) The configuration tool used by the license server administrator to generate and modify the local settings file /etc/default/flexnetls-*producerName*. This file defines the port number, log level, and other parameters used to run the license server. (Your installer must place this file in the same directory as flexnetls.jar.) ***Note ▪*** *This file is used only when the license server is installed as a Linux service using a SysV init runlevel system. However, the standard service installation method described in this book uses systemd. If you want to provide a SysV installation for your customers, see Appendix C, SysV Alternative for Installation on Linux.*
local-configuration.yaml	A file containing optional local settings available to the license server. It is used as follows: • On Windows, settings in this file override any values in flexnetls.settings that are not classed as system properties (system properties are passed to the Java Runtim system in -D*key*=*value* format). • On Linux systemd installs, this file is generated during installation and can be customized as needed to configure the server. Script command-line options are available to set common initial values. For SysV installs, the license server software package includes an example local-configuration.yaml file.
install-systemd.sh install-functions.sh	(*Linux only*) Files used to install and run the license server service. (Your installer must place these files in the same directory as flexnetls.jar.)
flexnetlsw.exe flexnetlsw.xml pre-install.vbs Remove-Permission.ps1 Set-Permission.ps1	(*Windows only*) Files used to run the license server as a Windows service. (Your installer must place these files in the same directory as flexnetls.jar.)
"service" directory	(*Linux only*) Directory containing the files needed to run FNLicensingService. This service retrieves the VM UUID for local license servers that run on certain virtual Linux machines and that use the VM UUID as their hostid.

Additional Components

You can purchase the following components separately from the license server software package. See your Revenera sales representative for information.

Back-Office Component

The FlexNet Embedded local license server supports only FlexNet Operations as a back office to activate the license pool on the server and with which to synchronize the server's license distribution and usage data. If you intend to run the license server with back-office support, you must purchase FlexNet Operations.

Support for Metered Licensing and Usage Capture

FlexNet Embedded supports metered licensing and usage capture. This support requires a separate purchase of the FlexNet Usage Capture and Management feature. (Usage capture is the FlexNet Embedded component of this feature; usage management is the FlexNet Operations component. Both components are required to use the feature.)

REST-driven Licensing

FlexNet Embedded REST-driven licensing uses a REST interface run against a CLS instance or local license server to manage licensing for your product. (This type of licensing is in contrast to incorporating FlexNet Embedded SDK licensing functionality in your licensed client code.) For a CLS instance, REST-driven licensing must be purchased as a separate component.

Important ▪ *Going forward, REST-driven licensing will be known as Cloud Monetization API. Future documentation will reflect this new terminology.*

Preparing and Deploying the License Server: Producer Experience

As the producer, you decide how you want to package and deliver the license server to your customers. The components will be installed to the selected install path on the enterprise system.

The following table summarizes the basic tasks you perform to prepare and deploy the license server to the enterprise. Each task description includes a link to further information in this book or a reference to the *FlexNet Embedded License Server Administration Guide*, which provides information for the license server administrator to monitor and manage the server at the enterprise site.

Table 2-2 ▪ Overview of the Software Producer Tasks

Task	Description
Determine the license server security profile for secure anchoring (optional)	You can enable functionality called *secure anchoring* to enhance the standard anchoring security normally provided for the license server. To enable this feature, you incorporate specific security-profile information in the license-server identity data. This step must occur before you generate your license server policy settings (the next task described). See Enabling Secure Anchoring in this chapter.
Determine deliverables needed to enable outgoing HTTPS (optional)	The FlexNet Embedded local license server is capable of HTTPS communication with FlexNet Operations. However, to enable this capability, you must ensure that certain files are included in the server installation and that the server is configured properly. See Determining Deliverables Needed to Enable Outgoing HTTPS in this chapter.
Determine administrative security policies (optional)	The license server provides a facility that secures access to the REST APIs used to administer the license server. This administrative security facility prevents unauthorized queries or operations on either the local license server or a CLS instance. For more information, see Determining Administrative Security Policies.
Generate the settings file defining license server policies	You need to generate a settings file that defines your policies for the various operations that the license server can perform, such as synchronization to and from the back office, licensing distribution, logging, license server failover, and others. You include this file with the license server installation to control the license server operations. See Generating the Producer Settings File in this chapter.
Determine the administration tools to provide the enterprise	The FlexNet Embedded local license server software package provides a server administration tool: the FlexNet License Server Administrator. You can choose to provide the enterprise customer with this tool or create an administration tool of your own. See Determining Administration Tools for the Enterprise in this chapter.
Entitle the enterprise account	Before an enterprise customer can activate licenses on the license server, you must set up the enterprise account on the back office. See Entitling the Enterprise Customer Account for more information.

Table 2-2 ▪ Overview of the Software Producer Tasks (cont.)

Task	Description
Deliver the license server to the enterprise customer	The final step is to package and deliver the license server to your enterprise customer. See Delivering the License Server for more information.

Enabling Secure Anchoring

The FlexNet Embedded local license server 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 devices that run the license server. While default anchoring stores the anchoring information in the anchor file, secure anchoring uses additional techniques to store anchor information on the system on which your license server runs.

Requirement

Enabling secure anchoring functionality requires the binary file containing your producer-identity data for the license server (for example, `ClientServerIdentity.bin`). This identity data is generated—along with your identity data files for the back office and the FlexNet Embedded client—by the back office itself or by the example `pubidutil` utility provided in your license server software package (see Publisher Identity Utility in Chapter 7, Producer Tools).

Enable Secure Anchoring

To enable secure anchoring, you use the Secure Profile utility to embed specific security-profile information into the identity data for the license server. For more information about secure anchoring and its enablement, see the following:

- Secure Anchoring in Chapter 6, Advanced License Server Features
- Secure Profile Utility in Chapter 7, Producer Tools

Determining Deliverables Needed to Enable Outgoing HTTPS

The FlexNet Embedded local license server is capable of HTTPS communication with FlexNet Operations. If you want to enable the license server for outgoing HTTPS, you need to determine whether to include a truststore file in the server installation and what configuration might be needed on the license server. For more information, see Outgoing HTTPS in Chapter 6, Advanced License Server Features.

Determining Administrative Security Policies

The FlexNet Embedded license server can control access to the REST APIs used to administer the license server, thus preventing unauthorized queries or administrative operations on the local license server or a CLS instance. For a local license server, you enable this administrative security by setting policies in the producer settings file (described briefly next). For a CLS instance, follow the procedure described in the FlexNet Operations documentation. For complete information about administrative security and the policies used to enable it, refer Administrative Security the in Chapter 6, Advanced License Server Features.

Generating the Producer Settings File

To run, the license server requires a `producer-settings.xml` file, which you create and distribute with the license server. The settings in this file define your policies for license server operations—for example, synchronization policies and virtual-environment policies, policies that control administrative security, and many others.

To generate the settings file, use the `flexnetlsconfig` utility provided in your license server software package. You can create multiple versions of this file to accommodate the different types of license server deployments at your enterprise customer sites. When you deploy the license server, the settings file must exist in the same directory as the license server executable. The license server administrator at an enterprise site is allowed to edit certain settings in this file as needed.

Policy Settings Reference

For a description of each policy setting available for inclusion in this file, the setting's default value, and whether it is editable by the enterprise license server administrator, see Appendix A, Reference: Policy Settings for the License Server. Note that some available settings are not included in the default version of the settings file. Therefore, to enable and control certain types of license server functionality, you need to add specific settings to this file. The chapters More About Basic License Server Functionality and Advanced License Server Features, which describe different types of functionality available with the license server, inform you which settings you need to include if you are enabling a specific function.

Requirements

Creating the settings file requires that you specify the binary file containing your producer-identity data for the license server (for example, `ClientServerIdentity.bin`). This binary file is generated—along with your identity data files for the back office and the FlexNet Embedded client—by the back office itself or by the example `pubidutil` utility provided in your license server software package (see Publisher Identity Utility in Chapter 7, Producer Tools).

If you intend to use secure anchoring functionality, it must be enabled before you generate policy settings. See the previous section Enabling Secure Anchoring for more information.

Generate the Policy Settings File

For complete instructions for generating a settings file with default policy settings or generating a customized version, see License Server Configuration Utility in Chapter 7, Producer Tools. As a reference, the section shows example contents of a policy settings file and a properties file used to update settings. The settings file is a mandatory component of the license server and must reside in the same directory as the `flexnetls` batch or shell file.

Determining Administration Tools for the Enterprise

The enterprise license-server administrator needs a facility that uses the REST endpoints for the FlexNet Embedded local license server, as described in Chapter 4, License Server REST APIs, to manage the server and control license distribution and usage. You can include one or more of the following administrator facilities with the license server installation:

- **FlexNet License Server Administrator command-line tool**—A ready-to-use command-line tool provided with the FlexNet Embedded local license server software package that lets the server administrator manage the license server and its operations. For more information, see the *Using the FlexNet License Server Administrator Command-line Tool* chapter in the *FlexNet Embedded License Server Administration Guide*.
- **Your own administrator interface or tool**—A license-server administration interface or tool that you develop, based the REST endpoints for the FlexNet Embedded local license server, as described in Chapter 4, License Server REST APIs.

Entitling the Enterprise Customer Account

To prepare the back office, whether for license enforcement or usage tracking with your FlexNet Embedded local license server, you need to create an *entitlement* for the license server—that is, one or more line items that make up the licenses the server will administer. The activation ID values associated with the license server's line items will be used to register the license server with FlexNet Operations and activate licenses on the enterprise customer's license server. The entitlement email sent to the customer should include these activation ID values.

For information about creating the entitlement for the license server and providing the customer with the activation IDs, refer to the following sections in the *FlexNet Operations User Guide*:

- "*FlexNet Operations Getting Started Guide for FlexNet Embedded Licensing*"
- "*FlexNet Operations Getting Started Guide for Usage Management*" (if you are implementing usage capture and management for your products)

When running the license server, the license server administrator specifies one or more of the activation IDs to register the server and obtain the licenses managed by the license server. (In some configurations, the licenses are automatically activated when the license server starts or each time online synchronization occurs.) The REST interface of the license server supports methods for passing activation IDs in a JSON file, as described in Chapter 4, License Server REST APIs.

Likewise, the FlexNet License Server Administrator command-line tool provides a means for activating licenses. See Chapter 3, Installing and Running the License Server for more information.

Delivering the License Server

As the software producer, you decide how you want to package and deliver the license server to your customers. The previous License Server Components section provided a description of the license server components included in your license server software package.

The following sections list the components for which you have redistribution rights as well as the files which you must not redistribute to customers.

Redistributable Components

Use the following table as a guide to determine which of these components to include in the license server product you deliver to customers.

You have redistribution rights to the components listed in this table.

Table 2-3 ▪ License Server Components Delivered to Customers

Components	Location in Producer Package	Customer Deliverables for Windows	Customer Deliverables for Linux
Mandatory components	`server` directory	• `flexnetls.bat` • `flexnetls.jar` • `flexnetls.settings` • `flexnetlsw.exe` • `flexnetlsw.xml` • `pre-install.vbs` • `Set-Permission.ps1` • `Remove-Permission.ps1`	For service installation using systemd: • `install-systemd.sh` • `install-functions.sh` • `flexnetls.jar` or For service installation using SysV: • `flexnetls` • `flexnetls.jar` • `configure`
	producer-generated	`producer-settings.xml` (must reside in same location as `flexnetls.jar`)	`producer-settings.xml` (must reside in same location as `flexnetls.jar`)
Optional: FlexNet License Server Administrator command-line tool	`enterprise` directory	• `flexnetlsadmin.bat` • `flexnetlsadmin.jar`	• `flexnetlsadmin.sh` • `flexnetlsadmin.jar`

Table 2-3 • License Server Components Delivered to Customers (cont.)

Components	Location in Producer Package	Customer Deliverables for Windows	Customer Deliverables for Linux
Optional: Offline synchronization tools	`enterprise` directory	• `backofficeofflinesynctool.bat` • `serverofflinesynctool.bat`	• `backofficeofflinesynctool` • `serverofflinesynctool`
	`lib` directory	• `EccpressoAll.jar` • `commons-codec-1.x.jar` • `flxPublicTools.jar`	• `EccpressoAll.jar` • `commons-codec-1.x.jar` • `flxPublicTools.jar`
Optional: Outgoing HTTPS file	`server` directory	A certificate you provide (required if you are using the "on-premises" version of FlexNet Operations and the server certificate has been signed by an unknown (private) certificate authority)	A certificate you provide (required if you are using the "on-premises" version of FlexNet Operations and the server certificate has been signed by an unknown (private) certificate authority)
Optional: Service for obtaining VM UUID on certain Linux platforms	`service` directory	N/A	• `bin` directory • `install-vm-only.sh` • `Readme.txt`
Optional: Software ID tag file	`server` directory	`regid.2009-06.com(...).swidtag`	`regid.2009-06.com(...).swidtag`

The license server components are installed in the default or specified installation path on the enterprise customer's system. For a complete list of supported platforms and other requirements for installing the license server on the enterprise, refer to the *FlexNet Embedded License Server Release Notes* for your release.

After the license server is installed, the enterprise's license server administrator can initiate the appropriate activation process to install licenses on the license server. For an understanding of the administrator's experience in activating licenses and managing the license server in general, refer to *FlexNet Embedded License Server Administration Guide*.

Components not to be Redistributed

The following files that are included in the FlexNet Embedded local license server software package are meant for producers and must not be redistributed:

- `flexnetlsconfig.jar`
- `flxBinary.jar`
- `flxTools.jar`

3

Installing and Running the License Server

This chapter provide instructions on installing and getting the FlexNet Embedded local license server up and running. It includes the following sections:

- Installing and Starting the License Server
- Editing Local Settings Post-Installation
- Understanding Hostids
- Activating the License Rights for the Server
- Granting Client Requests
- Upgrading the License Server
- Uninstalling the License Server

About “licenseServer_baseURL” Used in Commands

Some the commands referenced in this chapter use the license server’s base URL, indicated by *licenseServer_baseURL* in the command. For more information about this URL, see Base URLs in Chapter 4, License Server REST APIs.

Installing and Starting the License Server

This section covers the following information:

- Prerequisites
- Install and Start on Windows
- Install and Start on Linux
- Setting the Server Time Zone
- Logging Functionality

Prerequisites

Before installing and running the license server, perform the following:

- Download and extract the FlexNet Embedded license server software package as described in Downloading the Software Package in Chapter 2, Getting Started.
- Create the `producer-settings.xml` file that defines your license server policies. See License Server Configuration Utility in Chapter 7, Producer Tools, for more information about creating this file.
- Set the `JAVA_HOME` (or `JRE_HOME`) environment variable on your system to the path for your default JDK (or JRE) installation.

Note ▪ *The license server requires only the JRE component. If JRE is your default Java installation, set the JRE_HOME environment variable; if JDK is your default installation, set JAVA_HOME. See the "FlexNet Embedded License Server Release Notes" for details.*

Install and Start on Windows

Use the instructions in these sections to install and start the license server as a service or in console mode.

- Prepare for Windows Installation
- Install and Start as a Windows Service
- Start in Console Mode on Windows
- Manage the License Server Service on Windows

Prepare for Windows Installation

Use the following steps to configure license server in preparation for installation.

Task ***To prepare for installation***

1. Copy the `producer-settings.xml` file that you created into the `server` directory (or in the same directory as the `flexnetls.jar` and `flexnetls.bat` files) in the installation directory for the license server software package. This file defines operational policies for your license server. See License Server Configuration Utility in Chapter 7, Producer Tools, for details about this file.
2. To update settings for your local environment, use a text editor to open the `flexnetls.settings` file, found in the `server` directory (or in the same directory as the `flexnetls.jar` and `flexnetls.bat` files). Edit or add information as needed; or leave the file as is to accept the default settings. (For example, you might want to change the `JAVA_HOME` value or uncomment and provide a value for `PORT`.)

   When you update any setting, these rules apply:

   - Any setting value that uses a space must be enclosed in quotations
   - Insert no spaces before or after the equal sign (=) in the setting syntax (for example, **`PORT=7071`**).

Refer to the following for a description of each setting:

Table 3-1 ▪ Local Settings for the License Server on Windows

Setting	Description
Required Settings	
FLEXNETJAR	The Java executable file for the FlexNet Embedded local license server (default is `flexnetls.jar`).
PUBSETTINGS	The license server configuration file generated by the producer (default is `producer-settings.xml`).
JAVA_HOME (or JRE_HOME)	The path for JDK or JRE installation that the license server should use. The `flexnetls` batch file uses this location to find the necessary `java.exe` and `jvm.dll` files. By default, the license server uses the value of your `JAVA_HOME` (or `JRE_HOME`) system environment variable to determine the Java installation location, as indicated by the `"%JAVA_HOME:"=%"` (or `"%JRE_HOME:"=%"`) value for this local setting. However, if you want the license server to use a different Java installation on your system, edit this local setting to override the server's use of the environment variable. (This override pertains to the license server only; your system continues to use the Java installation defined by the system environment variable.) To perform the override, replace the default value (for example, `"%JAVA_HOME:"=%"`) with the path for the desired Java installation. ***Note ▪*** *The "flexnetls.settings file" includes both a JAVA_HOME and a JRE_HOME setting. If both values are set, JAVA_HOME is used.*
Optional Settings	
PORT	The listening port used by the license server. This value is not set by default. (If no value is specified, the server uses 7070 by default.) If the device on which the license server is running uses multiple network interfaces, you can use the `PORT` option to specify the interface that you want the license server to use. Simply include the IP address for the interface in square brackets, as shown in this example: `PORT=[127.0.0.1].1443`

Table 3-1 ▪ Local Settings for the License Server on Windows (cont.)

Setting	Description
ACTIVE_HOSTID	The hostid to use for the license server. The hostid can only be set using a configuration file if no hostid has yet been specified for the license server. Once a hostid has been set, it can only be changed using the REST API `/hostids/selected` or using a license server administration tool such as FlexNet License Server Administrator. The syntax is *value/type* (for example, `7200014f5df0/ETHERNET`). If no value is specified for this setting and no active hostid has yet been set for the license server, the license server uses, by default, the first available Ethernet address on the machine. If using a dongle ID or a hostid other than the first available Ethernet address, the license server administrator can specify it here. ***Important ▪*** *It is not recommended to change the hostid of a license server that has licenses mapped in the back office (FlexNet Operations). If the hostid is changed to a value that is different to that specified for the license server in the back office, any existing licenses mapped to the license server that is locked to the old hostid in the back office will be orphaned.* *To prevent this from happening, it is best practice to return the license server in the back office. During the return operation, you as the producer can transfer the licenses to a different device; this can be the same machine with the desired hostid. After the transfer, wait for the license server to synchronize with the back office (server synchronization occurs based on synchronization policies or on-demand).* *For more information on returning the license server, refer to the FlexNet Operations User Guide, section “Returning a Device”.* ***Important ▪*** *If you specify a server hostid using ACTIVE_HOSTID, the `server.hostType.order` property (if set) is ignored.* For more information about hostids, see Understanding Hostids.
HTTPS_SERVER_CONFIG	The path to the HTTPS “server” configuration file used to support *incoming* HTTPS from client devices. ***Note ▪*** *The “server” configuration file is being deprecated and will be removed in a future release. For details on how the enterprise creates this configuration file and sets up the server, see Incoming HTTPS in Chapter 6, Advanced License Server Features.*

Table 3-1 ▪ Local Settings for the License Server on Windows (cont.)

Setting	Description
HTTPS_CLIENT_CONFIG	The path to the HTTPS “client” configuration file used to support *outgoing* HTTPS communication to FlexNet Operations. ***Note ▪*** *The “client” configuration file is being deprecated and will be removed in a future release. For more information about the configuration file and HTTPS setup on the license server, see Outgoing HTTPS in Chapter 6, Advanced License Server Features.*
EXTENDED_SUFFIX	The suffix used for the extended hostid feature. See Extended Hostids in Chapter 6, Advanced License Server Features for details.
SERVER_ALIAS	A user-defined name (sometimes called *host name*) for the license server. This name is added to server’s capability requests to the back office, where it is then saved and used to identify the license server in the FlexNet Operations Producer and End User portals. One important use for this setting is that the alias can be included in the initial capability request sent at server registration, providing a helpful name by which users can identify the new server in the portals. If no alias is sent at registration, the server is identified by its hostid. For more information about hostids, see Understanding Hostids.

EXTRA_SYSPROPERTIES	One or more system properties (each in -D*key=value* format) that are passed to the Java Runtime system. The license server depends on the Java Runtime Environment to support certain network functionality such as specifying the HTTP proxy.
	
	For example, if you plan to have the license server communicate with the back office through an HTTP proxy, use this setting to identify the proxy parameters needed to configure the server. (For details, see Proxy Support for Communication with the Back Office in Chapter 6, Advanced License Server Features.) The following shows example proxy parameters listed as -D system properties for this setting:
	
	`EXTRA_SYSPROPERTIES="-Dhttp.proxyHost=10.90.3.133`
	`-Dhttp.proxyPort=3128 -Dhttp.proxyUser=user1a`
	`-Dhttp.proxyPassword=user1apwd35"`

Table 3-1 ▪ Local Settings for the License Server on Windows (cont.)

Setting	Description
BACKUP_SERVER_HOSTID	The hostid of the back-up license server in a failover configuration with the current license server (as the main server), if the back-up server is “unknown”—that is, not registered—in FlexNet Operations. This setting adds the back-up server’s hostid to the capability request sent by the current license server to the back office. The back office then automatically registers the back-up server in a failover configuration with the main server. This process saves the extra step of having to manually register the failover pair in FlexNet Operations. For more information, see Failover Using Synchronization to FlexNet Embedded in Chapter 5, More About Basic License Server Functionality. Make sure the back-up server’s hostid is the same type (for example, “ETHERNET”) as the hostid type of the current (main) license server. You might need to add this setting manually if it is not included as an available option in the settings file. For more information about hostids, see Understanding Hostids.

***Important ▪** Another file, “flexnetlsw.xml” file, also contains settings used to run the Windows service. However, do not edit this file. All information in this file is generated and maintained internally and should never be updated through an external means. To update service settings, edit only the “flexnetls.settings” file.*

Install and Start as a Windows Service

Use these steps to install and start the license server as a service on your Windows device.

***Important ▪** If you are upgrading your license server, you must uninstall the old license server service before installing and starting the service for the new license server. For instructions on uninstalling the license server service on Windows, see Uninstall the Windows Service.*

Task ***To install and start the license server as a service on Windows***

1. As an Administrator, open a command window and navigate to the `server` directory (or in the same directory as the `flexnetls.jar` and `flexnetls.bat` files) in the license server installation.
2. Execute `flexnetls.bat -install` to install the license server as a service.
3. Execute `flexnetls.bat -start` to start the service.

***Note** ▪ Depending on the security policies defined on your device, a warning message might be displayed, warning you to only run scripts that you trust. If such as warning is displayed, confirm that you want to allow each script to be executed.*

4. Confirm that the service is running by doing one of the following:

    - Execute `flexnetls.bat -status`. (The output should be `Service running`.)
    - In the Windows Services window (`services.msc`), check that the service **FlexNet License Server - *producer_name*** has started. (Note that in the Services tab of the Task Manager, the service's display name is **FNLS-*producer_name***.)

    (If you want to stop the service, run `flexnetls.bat -stop`.)

5. To view the license server log, navigate to the server's logging directory (by default, `C:\Windows\ServiceProfiles\NetworkService\flexnetls\`*`producer_name`*`\logs`), and review the contents of the appropriate `.log` file.

***Important** ▪ Because there is no verification that the service has started when executing the "flexnetls.bat -start" command, best practice is that you always check the status of the service after starting it.*

***Note** ▪ Once you install the license server as service, you cannot use any of the "flexnetls" non-service command-line options, such as "console" or "install".*

Start in Console Mode on Windows

Use these steps to start the license server in console mode on your Windows device.

***Note** ▪ Run the license server in console mode for testing and development purposes only. It is recommended that the enterprise customer run the license server only as a service.*

Task ***To start the license server in console mode on Windows***

1. As an Administrator, open a command window and navigate to the `server` directory (or in the same directory as the `flexnetls.jar` and `flexnetls.bat` files) in the license server installation.
2. Run `flexnetls.bat` to start the license server in console mode.
3. Confirm that the license server is running by doing one of the following:

    - Check the status of the license server using the FlexNet License Server Administrator command-line tool, included in your license-server software package (in the `enterprise` directory). Enter the following:

      ```
      flexnetlsadmin.bat -server licenseServer_baseURL -status
      ```

- View the license server log by navigating to the server's logging directory. By default, this directory is `C:\Users\`*`user_name`*`\flexnetls\`*`producer_name`*`\logs` when the server runs in console mode. Review the contents of the appropriate `.log` file.

Manage the License Server Service on Windows

The following provides a summary of the command options available with `flexnetls.bat` to manage the license server service on Windows.

The general command usage is the following:

```
flexnetls.bat [-command_option]
```

Table 3-2 ▪ Options Available with "flexnetls.bat" on Windows

Option	Description
-help	Lists the syntax of all `flexnetls.bat` commands.
-start	Starts the license server service.
-stop	Stops the license server service.
-install	Installs the license server as a service.
-uninstall	Uninstalls the license server service. (You must stop the service before using this option.)
-update	Stops the license server service and applies any changes made to `flexnetls.settings`. (You then need to restart the service.)
-status	Shows the status of the license server service.
-logging-threshold	Defines the logging threshold for the license server service to the value you specify: **`FATAL`**, **`ERROR`**, **`WARN`**, **`INFO`**, **`LICENSING`**, **`POLICY`**, or **`DEBUG`**. This value supersedes the current `logging.threshold` policy defined in `producer-settings.xml` for the license server. This threshold is the lowest level of log-message granularity to record. For example, if **`WARN`** is set, warning, debug, and error messages are recorded. However, if **`ERROR`** is set, only error messages are recorded. Setting this option from the `flexnetls` command line enables you to debug start-up problems with the license server. In general, however, you should use the license server's administration tools to control the logging threshold. For information about individual logging levels, see Logging Policies in Appendix A, Reference: Policy Settings for the License Server.

Table 3-2 ▪ Options Available with "flexnetls.bat" on Windows (cont.)

Option	Description
-reset-database	Resets trusted storage on the license server. This option is rarely used except for testing purposes or deploying extended hostids. The enterprise license server administrator should not use this option unless you provide a directive to do so in specific situations.
-restore-database	Restores trusted storage from a backup copy stored on the license server, when the license server is running in console mode. See Trusted Storage Backup and Restoration.
-restore-service-database	Restores trusted storage from a backup copy stored on the license server, when the license server is running as a service. See Trusted Storage Backup and Restoration.

Install and Start on Linux

The following instructions describe how to use systemd to configure, install, and run the local license server as a Linux service or in console mode:

- Producer Tasks
- Administrator Tasks
- Verify systemd-Enablement on Linux System
- Additional Requirement for Certain Linux Platforms in Virtual Environments
- Run the Install Script
- Manage the License Server Service on Linux

Alternatively, you can configure, install, and run the license server as a Linux service using a SysV init runlevel system. To do so, follow the instructions in Appendix B, SysV Alternative for Installation on Linux instead of the instructions in this chapter.

The content in this section assumes familiarity with the use of the systemd system for managing Linux services. You can find information about systemd at the following site:

> https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/System_Administrators_Guide/chap-Managing_Services_with_systemd.htm

Producer Tasks

As producer, distribute the following script files with the license server to support the running of the license server service under systemd at the enterprise customer's site:

- `install-systemd.sh`
- `install-functions.sh`

These files must reside in the same directory as `flexnetls.jar`. (In your FlexNet Embedded license server software package, all three files are located in the *install_dir*/`server` directory.) The scripts work together to generate files required to install and start the license server service in a systemd configuration.

Also ensure that you have generated the `producer-settings.xml` file and that is installed in the same directory as `flexnetls.jar` and the script files. (See License Server Configuration Utility in Chapter 7, Producer Tools.)

Optionally, you can create and distribute an `rpm` or `deb` wrapper that copies these required files to the proper location on the enterprise device and then installs the license server service.

For a list of all files that you can distribute with the license server, see Delivering the License Server in Chapter 2, Getting Started.

Administrator Tasks

The following is an overview of tasks that the enterprise's license server administrator performs to install and run the license server as a Linux service under systemd:

- **Task 1:** Verify that the Linux system on which the license server is being installed is systemd-enabled (see Verify systemd-Enablement on Linux System).
- **Task 2:** Run the `install-systemd.sh` installer (or your own wrapper), updating certain default configuration settings in the command line as needed (see Run the Install Script).
- **Task 3:** Manage the service (see Manage the License Server Service on Linux).
- **Task 4:** (Optional) Edit configuration settings post-installation as needed (see Post-Installation Configuration on Linux).

Verify systemd-Enablement on Linux System

Before proceeding with the installation, the license server administrator can run the following command to determine whether the Linux system on which the license server is being installed is systemd-enabled:

```
systemctl
```

If the system is systemd-enabled, the command runs successfully and lists all active units.

Additional Requirement for Certain Linux Platforms in Virtual Environments

Certain Linux platforms running on virtual machines do not provide an appropriate mechanism for retrieving the VM UUID for local license servers that use this ID as the hostid. To enable VM UUID retrieval on these platforms, the license server software package provides a special service, called `FNLicensingService`, which must be manually installed before the license server—whether installed as a service or in console mode—is started. For more information about hostids, see Understanding Hostids.

The installer for `FNLicensingService`, called `install-vm-only.sh`, is located in the `service` directory of the license server software package. This same directory also contains a `Readme` file that provides the following important information:

- Specific conditions under which a license server requires `FNLicensingService`

- Prerequisites for running this service
- Instructions for installing the service

If an enterprise customer requiring this service, include the contents of the `server` directory in the license server deliverable and provide any instructions necessary for installing the service.

Run the Install Script

The license server administrator runs `install-systemd.sh` to create and start the license server service using either the default configuration for the service or a configuration customized through command-line options, as described in these next sections:

- About the Install Script
- Install and Start the Service with the Default Configuration
- Install and Start the Service with a Modified Configuration
- Configuration Values Editable from the Command Line
- Start and Stop in Console Mode on Linux

The commands described in these next sections are run from the directory in which the systemd script files (see About the Install Script) and `flexnetls.jar` reside.

Prerequisite

Only root or `sudo` users can run the install script.

About the Install Script

When run, the install script `install-systemd.sh` performs the following:

- Generates the service unit file called `/etc/systemd/system/flexnetls-`*`producer_name`*`.service`.
- Generates the configuration unit file called `/etc/systemd/system/flexnetls-`*`producer_name`*`.service.d/flexnetls.conf`. This file contains the configuration settings needed to start the service. It is created with default values, but the license server administrator can include options in the install-script command line to generate this file with custom values. See Install and Start the Service with a Modified Configuration.

  The administrator can also edit this configuration file post-installation as needed, as described in Post-Installation Configuration on Linux.

- Generates a `local-configuration.yaml` file in the `/opt/flexnetls/`*`producer`* directory. This file contains optional settings specific to the local service environment. Generally, these settings are initially disabled; the license server administrator can edit or enable them as needed once the service is installed. See Edit “local-configuration.yaml” Settings for details.

  Note ▪ *The generated local-configuration.yaml is auto-populated with the port numbers, hostids, logging settings and TLS (HTTPS) settings as specified by (optional) command-line parameters. Note that these settings are only auto-populated if there isn’t a YAML file at the destination, or if `--overwrite` is specified.*

- By default, enables the standard Syslog logging process for the license server service.
- Starts the license server service.

Install and Start the Service with the Default Configuration

The following command installs and starts the license server service using the default configuration. For more about the default values used to configure the service, see Configuration Values Editable from the Command Line.

Task ***To install and start the license server service using the default configuration***

Run the following:

```
sudo ./install-systemd.sh
```

Install and Start the Service with a Modified Configuration

The following command installs and starts the license server service using a configuration modified through one or more specified command-line options. For a description of the configuration values that can be updated from the install-script command line, see the next section, Configuration Values Editable from the Command Line.

Task ***To install and start the license server service with updates to the default configuration***

Run the install script, specifying one or more command-line options to change specific configuration settings (for example, the `user` value, as shown here):

```
sudo ./install-systemd.sh --user flexnetls01
```

Configuration Values Editable from the Command Line

The following options are used with the `./install-systemd.sh` command to edit current configuration settings for the license server service. (Running `./install-systemd.sh --help` also lists these command-line options.)

Table 3-3 ▪ Configuration Values Editable from the Command Line

Setting	Description
--program-dir *dir*	The installation location of the license server service (default is `/opt/flexnetls/`*`producer_name`*).
--data-dir *dir*	The location of trusted storage (default is `/var/opt/flexnetls/`*`producer_name`*). This overrides the `server.trustedStorageDir` value in the `producer-settings.xml`.
--user *user_name*	The user name under which the service runs (default is `flexnetls`).
--group *group_name*	The group name under which the service runs (default is `flexnetls`).

Table 3-3 ▪ Configuration Values Editable from the Command Line (cont.)

Setting	Description
--java_home *path* or **--jre_home *path***	The path for JDK or JRE installation that the license server should use. By default, the license server uses the value of the `JAVA_HOME` or `JRE_HOME` system environment variable (whichever is defined on the device on which the server is installed) as the Java installation to use. However, to have the server use a different Java installation, the license server administrator must provide the explicit path for the installation as either the `java_home` or `jre_home` value. (This override pertains to the license server only; the enterprise device in general continues to use the Java installation defined by its system environment variable.)
The following two settings—"port" and "logging-threshold"—do not exist in the `.conf` file. However, if either setting is updated using the command-line option, the new value is captured in the `local-configuration.yaml` file during its creation and is subsequently used by the license server service. If neither setting is updated, the service uses the default value (or the value specified in `producer-settings.xml`) for the setting.	
--port *port*	The listening port used by the license server service (default is `7070`). If the device on which the license server is running uses multiple network interfaces, the license server administrator can use the `port` option to specify the interface that the license server is to use. The IP address for the interface is included in square brackets, as shown in this example: `--port [127.0.0.1].1443`
--logging-threshold *level*	The lowest level of log-message granularity to record—`FATAL`, `ERROR`, `WARN`, `INFO`, `LICENSING`, `POLICY`, or `DEBUG`. For example, if `FATAL` is set, only messages about fatal events are recorded. However, if `WARN` is set, fatal-event, error, and warning messages are recorded. (Default is `INFO`.) The `POLICY` threshold records information about the checkout process when feature selectors are used to filter features. For information about individual logging levels, see Logging Policies in Appendix A, Reference: Policy Settings for the License Server.
--tls-port *port*	The HTTPS listening port used by the license server service.
--tls-cert *path*	Path to the "server" certificate file. ***Note ▪** The "server" configuration file is being deprecated and will be removed in a future release.*
--tls-cert-password *password*	The password for the certificate file, if provided. The password will be automatically obfuscated.

Start and Stop in Console Mode on Linux

Use these steps to start and stop the license server in console mode on your Linux device.

***Note** ▪ Run the license server in console mode for testing and development purposes only. It is recommended that the enterprise customer run the license server only as a service.*

***Task** To install and start the license server in console mode on Linux*

1. After you have installed and started the license server service, stop the service:

```
sudo systemctl stop flexnetls-producer_name
```

2. Go to the `opt/flexnetls/demo` directory, and issue this command:

```
sudo ./console.sh
```

To shut down console mode, press Ctrl + C.

Note that the license server running in console mode and as a service use the same trusted-storage and log locations.

Manage the License Server Service on Linux

Once the license server service is installed, you can manage the service using the systemd command `systemctl`. The following describes basic management functions for the service:

- Obtain the Service Status
- Stop the Service
- Start the Service
- Restart the Service
- Re-read All systemd Unit Files

Prerequisite

You must perform `systemctl` tasks as a root or `sudo` user.

Obtain the Service Status

The following command obtains the “active” or “not active” status of license server service.

***Task** To obtain the status of the license server service*

Run the following command:

```
sudo systemctl -l status flexnetls-producer_name
```

The `-l` switch disables truncation of lines in the output.

The following shows example status output. The current service status is highlighted in this excerpt:

```
flexnetls-fnedemo.service - FlexnetLS Local License Server.
Loaded: loaded (/etc/systemd/system/flexnetls-fnedemo.service; enabled; vendor preset: disabled)
Drop-In: /etc/systemd/system/flexnetls-fnedemo.service.d
└─flexnetls.conf
Active: active (running) since Tue 2018-11-22 14:28:25 GMT; 18h ago
...
```

Stop the Service

The following command stops the license server service.

Task ***To stop the license server service***

Use this command:

```
sudo systemctl stop flexnetls-producer_name
```

Start the Service

The following command starts the license server service.

Task ***To start the license server service***

Use this command:

```
sudo systemctl start flexnetls-producer_name
```

Restart the Service

The following command stops and then restarts the license server service without re-reading all the systemd unit files related to all services. This command is useful if you have made changes to the `local-configuration.yaml` file (see Edit “local-configuration.yaml” Settings).

Task ***To restart the license server service***

Run the following command:

```
sudo systemctl restart flexnetls-producer_name
```

Re-read All systemd Unit Files

The following command is used to re-read all systemd unit files related to all services. This command is useful when you have made changes manually to the `flexnetls.conf` file and want to apply the changes before restarting the service (see Edit the .conf File Manually).

Task ***To re-read all systemd unit files for all services***

Run the following command:

```
sudo systemctl daemon-reload
```

Setting the Server Time Zone

The license server can either use its default time zone or Coordinated Universal Time (UTC) for determining a feature's expiry date, start date, and issue date. You configure the time zone using the `licensing.defaultTimeZone` setting. Valid values:

- **UTC**— If `UTC` is set, a feature's start date is the start of the specified day in Coordinated Universal Time (UTC). Equally, a feature will expire at the end of the day of the configured expiry date in UTC time. This is the default value.
- **SERVER**—If `SERVER` is set, a feature's start date is the start of the specified day in the server's default time zone. Equally, a feature will expire at the end of the day of the configured expiry date in the server's default time zone.

You can set the time zone in the following ways:

- On a CLS instance, you can modify the `licensing.defaultTimeZone` setting at instance level using the `/configuration` API. See APIs to Manage Instance Configuration.
- On a local license server, edit the `licensing.defaultTimeZone` setting in the `producer-settings.xml` file, which you generate and provide with the license server. Alternatively, you can set this at instance level using the `/configuration` API.

Note ▪ *To change this setting at a global level for your tenant, contact Revenera support.*

Changes to the time zone setting affect features as they are being provisioned onto the server. It does not affect features that are already on the server.

For a description of license server policies, see Appendix A, Reference: Policy Settings for the License Server.

Logging Functionality

The logging functionality that is available for the local license server is described in detail in Appendix E, Logging Functionality on the Local License Server. The appendix covers how to configure the logging style and how to integrate license server logging with external systems.

Editing Local Settings Post-Installation

The license server package provides (or provides a means to generate) a local settings file that configures the license server for your specific environment. You can use the file as is with its default settings, or you can modify the settings as needed to reflect your environment. The previous section Installing and Starting the License Server describes how to update these settings before server installation. Use the following instructions to edit the settings any time *after* installation.

Post-Installation Configuration on Windows

If you need to modify the license server's local settings (defined in the `flexnetls.settings` file) *after* you have installed the server as a service on a Windows platform, use the appropriate procedure.

When Server Runs as a Service

Use this procedure to update local settings in `flexnetls.settings` when the license server is running as a service on Windows. This file is located in the `server` directory (or in the same directory as the `flexnetls.jar` and `flexnetls.bat` files) in the license server installation.

***Important ▪** Never edit the "flexnetlsw.xml" file, which also contains settings used to run the Windows service. All information in this file is generated and maintained internally and should never be updated through an external means. To update service settings, edit only the "flexnetls.settings" file.*

Task ***To modify local settings after installing the license server as a service on Windows***

1. As an Administrator, open a command prompt window, and navigate to the `server` directory in license server installation.
2. Open the `flexnetls.settings` file in a text editor, and edit the settings as needed. (For example, you can uncomment settings or change existing values.) See the previous section Prepare for Windows Installation for details about the editing process.
3. Save the file.
4. Execute **`flexnetls.bat -update`** to apply the setting changes.
5. Execute **`flexnetls.bat -start`** to start the license server service, or restart it from the Windows Services window.

   ***Note ▪** Depending on the security policies defined on your device, a warning message might be displayed, warning you to only run scripts that you trust. If such as warning is displayed, confirm that you want to allow each script to be executed.*

Add a Setting to Control the License Server Load

By default, the local license server has no limit for the number of capability requests it will accept at a given time. However, if a license server experiences a high load of capability requests, a degradation in response time can result. A setting called `rate-limit` is available that sets a maximum on the number of capability requests per second the license server will allow.

This option is intended for use only in situations where this type of exceptional load is expected, as the rate limit helps to enable consistent response times for accepted requests. A number of factors affect the rate at which the server is capable of handling requests, including the specification of the server hardware and the complexity of the requests. Therefore, the `rate-limit` configuration should be determined only through load-testing on a mirror of the production environment using a representative client load.

You define this setting in the `local-configuration.yaml` file, another settings file located in the `server` directory (or in the same directory as the `flexnetls.jar` and `flexnetls.bat` files).

Task ***To define the rate-limiting setting***

1. As an Administrator, open a command prompt window, and navigate to the `server` directory in your license server installation.
2. In a text editor, open the `local-configuration.yaml`.
3. Insert (or update) the following line in the file:

   ```
   rate-limit: value
   ```

   where *value* is the maximum number of capability requests per second that you want the license server to handle.
4. Save the file.
5. Execute **`flexnetls.bat -update`** to update the settings.
6. Execute **`flexnetls.bat -start`** to start the license server service, or restart it from the Windows Services window.

   Note *▪ Depending on the security policies defined on your device, a warning message might be displayed, warning you to only run scripts that you trust. If such as warning is displayed, confirm that you want to allow each script to be executed.*

When Server Runs in Console Mode

Use this procedure to update local settings in `flexnetls.settings` when the license server is running in console mode on Windows. This file is located in the `server` directory (or in the same directory as the `flexnetls.jar` and `flexnetls.bat` files) in the license server installation.

Task ***To modify local settings after installing the license server in console mode on Windows***

1. In the command window running the license server, press Ctrl + c to close the window and stop the server.
2. From the `server` directory, open the `flexnetls.settings` file in a text editor, and edit the settings as needed. (For example, you can uncomment settings or change existing values.) See the previous section Prepare for Windows Installation for details about the editing process.
3. Save the file.
4. Execute **`flexnetls.bat`** to run the server in console mode. (The server now uses the updated settings.)

Post-Installation Configuration on Linux

If you need to modify the license server's local settings *after* you have installed the license server on a Linux platform, use these procedures:

- Edit Settings in "flexnetls.conf"
- Edit the DEFINES Setting
- Edit "local-configuration.yaml" Settings
- Define Command-Line Options for HTTPS Configuration Files

Edit Settings in "flexnetls.conf"

The license server administrator can use either of the following methods to update configuration settings in the `/etc/systemd/system/flexnetls-`*`producer_name`*`.service.d/flexnetls.conf` unit file:

- Edit the .conf File Manually
- Re-install the Service with Command-line Options to Update Settings

Edit the .conf File Manually

The following procedure manually updates settings in the `flexnetls.conf` unit file.

Task ***To edit the flexnetls.conf file manually***

1. Stop the license server service:

   ```
   sudo systemctl stop flexnetls-producer_name
   ```

2. As a root or `sudo` user, edit the `/etc/systemd/system/flexnetls-`*`producer_name`*`.service.d/flexnetls.conf` file in a text editor (for example, GNU nano or gedit).
3. Once the edits are complete, run the following command to re-read all the systemd unit files to capture the configuration changes:

   ```
   sudo systemctl daemon-reload
   ```

4. Start the license server service:

```
sudo systemctl start flexnetls-producer_name
```

Re-install the Service with Command-line Options to Update Settings

The following command reinstalls the license server service, using command-line options to update specific settings in the `flexnetls.conf` file. For a description of the configuration settings that can be updated from the install-script command line, see Configuration Values Editable from the Command Line.

```
sudo ./install-systemd.sh  --user flexnentls1 --group flexnetls1
```

If the `--overwrite` option is included in the command, the re-installation generates a new `.conf` file and a new `.yaml` file with the specified updates; any changes previously made to these files are lost.

Edit the DEFINES Setting

The `DEFINES` setting in the `flexnetls.conf` unit file is the same as `EXTRA_SYSPROPERITES` in the `flexnetls.settings` file used in the standard (SysV init) license-server service installations. By design, this setting has no corresponding command-line option to update its value; therefore, the license server administrator needs to edit the setting manually in the configuration file. For instructions on manually editing this file, see Edit the .conf File Manually.

This setting is used to define one or more system properties (each in -D*key*=*value* format) that the license server needs to pass to the Java Runtime system to support certain network functionality, such as an HTTP proxy.

For example, if the license server is to communicate with the back office through an HTTP proxy, the administrator can use this setting to identify the proxy parameters needed to configure the server. (For details, see Proxy Support for Communication with the Back Office in Chapter 6, Advanced License Server Features.) The following shows example proxy parameters listed as -D system properties for this setting:

```
DEFINES="-Dhttp.proxyHost=10.90.3.133 -Dhttp.proxyPort=3128 -Dhttp.proxyUser=user1a
-Dhttp.proxyPassword=user1apwd35"
```

By default, this setting has no properties defined.

Edit “local-configuration.yaml” Settings

Use the following procedure to update the `local-configuration.yaml` file generated in the `/opt/flexnetls/`*`producer`* directory during installation. This file contains optional settings for the license server service that are, by default, disabled.

***Note** ▪ For systemd installs, the install script generates the `local-configuration.yaml` file. For SysV installs, the license server software package includes an example `local-configuration.yaml` file.*

Edit Settings in the “local-configuration.yaml” File

The following procedure enables, edits, or disables settings in `local-configuration.yaml` file, located in the same directory as `flexnetls.jar`.

Task ***To edit settings in the “local-configuration.yaml’ file***

1. As a root or sudo user, edit the local-configuration.yaml file in an editor. Uncomment (or comment out) lines and edit setting values as needed. For a description of the settings, see the next section.
2. Run the following command to stop and restart the service to apply the configuration changes:

```
sudo systemctl restart flexnetls-producer_name
```

Settings in the “local-configuration.yaml” File

The following settings can be edited in the local-configuration.yaml file:

Table 3-4 ▪ Settings in the “local-configuration.yaml” File

Setting	Description
port: ***port***	The override for the listening port used by the license server, as described in Configuration Values Editable from the Command Line. The license server administrator can edit this override value as needed. If no override is specified (that is, the setting is commented out), the service uses the default port 7070 (or the value defined in producer-settings.xml).

Table 3-4 ▪ Settings in the “local-configuration.yaml” File (cont.)

Setting	Description
active-hostid: *value/type*	The hostid to use for the license server. The hostid can only be set using a configuration file if no hostid has yet been specified for the license server. Once a hostid has been set, it can only be changed using the REST API `/hostids/selected` or using a server administration tool, such as FlexNet License Server Administrator. The syntax is *value/type* (for example, `7200014f5df0/ETHERNET`). If no value is specified for this setting and no active hostid has yet been set for the license server, the license server uses, by default, the first available Ethernet address on the machine. If using a dongle ID or a hostid other than the first available Ethernet address, the license server administrator can specify it here. ***Important*** ▪ *It is not recommended to change the hostid of a license server that has licenses mapped in the back office (FlexNet Operations). If the hostid is changed to a value that is different to that specified for the license server in the back office, any existing licenses mapped to the license server that is locked to the old hostid in the back office will be orphaned.* *To prevent this from happening, it is best practice to return the license server in the back office. During the return operation, you as the producer can transfer the licenses to a different device; this can be the same machine with the desired hostid. After the transfer, wait for the license server to synchronize with the back office (server synchronization occurs based on synchronization policies or on-demand).* *For more information on returning the license server, refer to the FlexNet Operations User Guide, section “Returning a Device”.* ***Important*** ▪ *If you specify a server hostid using `active-hostid`, the `server.hostType.order` property (if set) is ignored.* For more information about hostids, see Understanding Hostids.

Table 3-4 ▪ Settings in the "local-configuration.yaml" File (cont.)

Setting	Description
backup-hostid: ***value/ type***	The hostid of the back-up license server in a failover configuration with the current license server (as the main server), if the back-up server is "unknown"—that is, not registered—in FlexNet Operations. The syntax is `value/type` (for example, `7200014f5df0/ETHERNET`). For more information about hostids, see Understanding Hostids. This setting adds the back-up server's hostid to the capability request sent by the current license server to the back office. The back office then automatically registers the back-up server in a failover configuration with the main server. This process saves the extra step of having to manually register the failover pair in FlexNet Operations. For more information, see Failover Using Synchronization to FlexNet Embedded in Chapter 5, More About Basic License Server Functionality. Make sure the back-up server's hostid is the same type (for example, "ETHERNET") as the hostid type of the current (main) license server.
extended-suffix: ***suffix***	(Producers only) The suffix used for the extended hostid feature. See Extended Hostids in Chapter 6, Advanced License Server Features, for details.
server-alias: ***alias***	A user-defined name (sometimes called *host name*) for the license server. This name is added to server's capability requests to the back office, where it is then saved and used to identify the license server in the FlexNet Operations Producer and End User portals. One important use for this setting is that the alias can be included in the initial capability request sent at server registration, providing a helpful name by which users can identify the new server in the portals. If no alias is sent at registration, the server is identified by its hostid.
jni-helper-directory: ***path***	If the default location for the JNI shared library (`/tmp`) is not suitable (for example, because execute permission is required), use this setting to supply an alternative directory path. The directory has to exist and be writable by the server.
logging-threshold: ***level***	The override for the lowest level of log-message granularity to record—`FATAL`, `ERROR`, `WARN`, `INFO`, `LICENSING`, `POLICY`, or `DEBUG`, as described in Configuration Values Editable from the Command Line. The license server administrator can edit this override value as needed. If no override is specified (that is, the setting is commented out), the service uses the default level `INFO` (or the value defined in `producer-settings.xml`). For information about individual logging levels, see Logging Policies in Appendix A, Reference: Policy Settings for the License Server.

Table 3-4 ▪ Settings in the “local-configuration.yaml” File (cont.)

Setting	Description
loggingStyle: ***style***	The logging style determines rollover, JSON formatting and timestamp behavior. If no logging style is specified, the default DAILY_ROLLOVER is used. **Logging style values:** **DAILY_ROLLOVER**—The log file is closed at midnight local time, compressed with gzip, and a new log file is started. Timestamp values use the local time zone. This is the default if no logging style is specified. **DAILY_ROLLOVER_UTC**—Same as DAILY_ROLLOVER, but with UTC timestamp. **CONTINUOUS**—Legacy value. Rollover is handled as DAILY_ROLLOVER. **CONTINUOUS_UTC**—Legacy value. Rollover is handled as DAILY_ROLLOVER_UTC. **JSON_ROLLOVER**—Logs are formatted using JSON. The log file is closed at midnight local time, compressed with gzip (suitable for filebeat), and a new log file is started. Always uses ISO 8601 UTC timestamps. **JSON**—Logs are emitted as JSON to stdout only (suitable for Docker). Always uses ISO 8601 UTC timestamps. ***Note ▪*** *The Docker-friendly JSON logging style is not supported when the local license server is run as a Windows or Linux service (because it only writes to stdout). If set, the style will be changed to DAILY_ROLLOVER.*
https-in: ***parameters***	Parameters used to access the “server” certificate from a certificate authority (CA) to support *incoming* HTTPS from client devices. For details, see Incoming HTTPS in Chapter 6, Advanced License Server Features. Parameters also control the enabled cipher list. For more information, see Configuring the Cipher Choice Mechanism. These parameters replace the “server” configuration file used for HTTPS in previous FlexNet Embedded license server versions. To use a “server” configuration file from a previous license server version, the license server administrator needs to set up a command-line option to specify the file path and name. For more information, see Define Command-Line Options for HTTPS Configuration Files. ***Note ▪*** *The “server” configuration file is being deprecated and will be removed in a future release.*

Table 3-4 ▪ Settings in the "local-configuration.yaml" File (cont.)

Setting	Description
https-out: ***parameters***	Parameters to access the truststore containing root and intermediate certificates to validate the license server's *outgoing* HTTPS communication to FlexNet Operations. This setting is not required for Revenera-hosted FlexNet Operations instances. If you have the "on-premises" version of FlexNet Operations and the HTTPS server certificate has been signed by an unknown (private) certificate authority, you must define this setting. These parameters replace the "client" configuration file used for HTTPS in previous FlexNet Embedded license server versions. To use a "client" configuration file from a previous license server version (instead of these parameters), the administrator needs to set up a command-line option to specify the file path and name. For more information, see Define Command-Line Options for HTTPS Configuration Files. ***Note ▪*** *The "client" configuration file is being deprecated and will be removed in a future release.*
rate-limit: ***value***	The maximum number of capability requests per second that the license server will accept. By default, the local license server has no limit for the number of capability requests it will accept at a given time. However, if a license server experiences a high load of capability requests, a degradation in response time can result. This option is intended for use only in situations where this type of exceptional load is expected, as such a limit helps to enable consistent response times for accepted requests. A number of factors affect the rate at which the server is capable of handling requests, including the specification of the server hardware and the complexity of the requests. Therefore, the `rate-limit` value should be determined only through load-testing on a mirror of the production environment using a representative client load. If this setting is currently not included in the `.yaml` file, you can manually add it to the file to define the necessary rate limit.

Define Command-Line Options for HTTPS Configuration Files

The `local-configuration.yaml` includes the `https-in` and `https-out` options used to define the parameters for the license server's HTTPS communications with FlexNet Embedded clients and FlexNet Operations. These two options replace the "server" and "client" HTTPS configuration files used in previous license server versions. However, the license server administrator can still use these configuration files (instead of the `https-in` and `https-out` options) by providing command-line options in the `flexnetls.conf` unit file to identify the paths to these files.

Note ▪ The “server” configuration file is being deprecated and will be removed in a future release.

Task ***To identify the path for an HTTPS configuration file***

Use the instructions in Edit the .conf File Manually to edit (or add) the `Environment="OPTIONS="` line in the `flexnetls.conf` unit file to specify one or both command-line options:

- To specify the command-line option that identifies the path and name for the “server” configuration file, enter the following:

  `Environment="OPTIONS=--https-server-configuration filePathAndName"`

- To specify the command-line option that identifies the path and name for the “client” configuration file, enter the following:

  `Environment="OPTIONS=--https-client-configuration filePathAndName"`

- To specify both command-line options, enter the following:

  `Environment="OPTIONS=--https-server-configuration filePathAndName --https-client-configuration filePathAndName"`

Configuring the Cipher Choice Mechanism

When enabling HTTPS on a local license server, license server administrators have to be careful about the choice of protocols and TLS ciphers. Failure to do this will leave the license server vulnerable to attacks.

Older TLS protocol versions (SSLv3, TLSv1.1) should not be active, not even for protocol re-negotiation.

Older ciphers (RC4, RSA for key exchange) and export-grade ciphers should not be enabled. AES should use GCM mode rather than CBC in order to avoid padding attacks, and AEAD (Authenticated Encryption with Associated Data) should be used for integrity checking wherever possible.

The enabled protocols in the local license server are TLSv1.2 and TLSv1.3. The latter requires Java 11, or versions of Java 8 greater than 8u261 (this version may differ slightly for OpenJDK releases). The local license server will automatically enable TLSv1.3 if it's available.

All known vulnerable ciphers are disabled in the default configuration. The enabled cipher list can be controlled through the configuration property `tlsCipherSuites` in the `https-in` section of `local-configuration.yaml`:

```
# Choice of TLS cipher suites. One of MODERN, COMPATIBLE or WEAK.
tlsCipherSuites: COMPATIBLE
```

The following table describes the cipher suites that are available:

Cipher Suite	Available Ciphers	Notes
MODERN TLS 1.3	"TLS_AES_128_GCM_SHA256", "TLS_AES_256_GCM_SHA384", "TLS_CHACHA20_POLY1305_SHA256"	ChaCha20 requires a LTS release of Java 11 (tested with OpenJDK Runtime Environment Microsoft-27990 (build 11.0.13+8-LTS)). There is a high chance of TLS handshake failures with MODERN unless all clients served by the local license server support TLS 1.3.
COMPATIBLE	"TLS_AES_128_GCM_SHA256", "TLS_AES_256_GCM_SHA384", "TLS_CHACHA20_POLY1305_SHA256", "ECDHE-ECDSA-AES128-GCM-SHA256", "ECDHE-RSA-AES128-GCM-SHA256", "ECDHE-ECDSA-AES256-GCM-SHA384", "ECDHE-RSA-AES256-GCM-SHA384", "ECDHE-ECDSA-CHACHA20-POLY1305", "ECDHE-RSA-CHACHA20-POLY1305", "DHE-RSA-AES128-GCM-SHA256", "DHE-RSA-AES256-GCM-SHA384"	Always use AES in GCM mode rather than CBC (to avoid padding attacks). Avoid deprecated and unsafe ciphers. This is the default setting.
WEAK	N/A	No constraints. Vulnerable ciphers are disabled at the JSSE level if at all. This choice would only be used when there is a need to communicate with old clients that get TLS handshake failures with COMPATIBLE.

Understanding Hostids

FlexNet Embedded uses system identifiers, called *hostids*, for identification. When the FlexNet Embedded local license server communicates with the back office, FlexNet Operations, to obtain its pool of licenses, the back office uses the server's hostid to identify that particular server. Similarly, when the client requests a license 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 focuses on license server hostids. For more information about client hostids, refer to the user guide for the respective client kit.

Hostid Types

FlexNet Embedded supports the following standard hostid types (also sometimes referred to as *host types*). If no hostid is specified, the license server looks up the hostids in the native DLL component of the local license server, in the order in which they are listed in the following table:

Table 3-5 ▪ License server hostid types

Hostid	Description
PUBLISHER_DEFINED	In addition to the standard hostid types (`ETHERNET`, `FLEXID9`, `FLEXID10`, and `VM_UUID`), you can optionally specify a producer-defined hostid value to identify the server to the back office. You define this hostid in `producer_settings.xml`. For more information about producer-defined hostids, see Chapter 6, Advanced License Server Features, section Producer-defined Hostids.
ETHERNET	The unique identifier assigned to each network interface controller (NIC).
FLEXID9	The hostid of the Aladdin dongle attached to the server.
FLEXID10	The hostid of the Wibu-Systems dongle attached to the server.
VM_UUID	Available if virtual machine detection is enabled.

Specifying a Server Hostid

If the license server is run with back-office support, you as the producer must specify a hostid in the back office during license server creation. The workflow for specifying a server hostid depends on whether a local license server or a CLS instance is used. In implementations with a local license server, the license server administrator is required to identify the hostid and communicate it to you. When using a CLS instance, the server hostid is based on the server instance ID (generated when the instance is created). During synchronization from the back office (which occurs based on synchronization policies or on-demand), the back office sends the hostid along with other information to the license server.

Instead of waiting for the synchronization to occur, the administrator of a local license server can also specify the license server's hostid manually. It is vital that the hostid that is set on the local license server matches the hostid that is specified in the back office. If the hostid on the license server is changed to a value that is different to that specified in the back office, any licenses already mapped to the license server that is locked to the old hostid in the back office will be orphaned.

If the server administrator needs to change the local license server's hostid when it already has licenses mapped in the back office, it is best practice to return the license server in the back office. During the return operation, you can transfer the licenses to a different device; this can be the same machine with the desired hostid. After the transfer, wait for the license server to synchronize with the back office. For more information about returning the license server, refer to the *FlexNet Operations User Guide*, section “Returning a Device”.

Selecting a Hostid

Specifying a server hostid is optional. If no hostid is set, the license server looks up the hostids in the native DLL component of the local license server, in the order in which they are listed in the table in Hostid Types. Alternatively, you as the producer can set the order in which a local license server looks up the hostid. See Specifying the Order of Hostid Types.

However, it is important to know that the Ethernet hostid might change, depending on the adapter that is in use. Consider the following scenario: A laptop plugged into a docking station uses the Ethernet address of the docking station; however, when it is disconnected from the docking station, the Ethernet address is no longer available. A wireless adapter also has an Ethernet address and this address is not available when either the wireless adapter is removed from the machine or when the wireless adapter is disabled, but still physically attached to the machine.

In cases as described above, it is advisable to specify the hostid manually to ensure continuous operation.

Manually Specifying a Hostid

The license server administrator can specify the server hostid (a value/type pair) in one of the following ways:

- By calling the REST API `/hostids/selected`. Any appropriate REST API administrative client—for example, the FlexNet License Server Administrator command-line tool (flexnetlsadmin) or a third-party tool such as Postman—can be used to call this API. For more information, see Hostid Selected for Instance in Chapter 4, License Server REST APIs.
- Using the configuration files (depending on the platform):
    - `flexnetls.settings`, setting `ACTIVE_HOSTID` (Windows)
    - `local-configuration.yaml`, setting `active-hostid`

    Important ▪ *The license server administrator can only set the hostid using a configuration file if no hostid has yet been specified for the license server. Once a hostid has been set, it can only be changed using the REST API.*

Important ▪ *If you manually specify a server hostid, the `server.hostType.order` property (if set) is ignored.*

Specifying the Order of Hostid Types

You can specify the order in which the local license server picks the hostid type, using the property `server.hostType.order` in `producer-settings.xml`. For more information about this property, see Reference: Policy Settings for the License Server, entry server.hostType.order.

Retrieving Server Hostids

Task ***To retrieve a server hostid, do one of the following***

- Call the JSON REST endpoint `http://licenseServerHostName/api/1.0/hostids` (or `/hostids/selected`)

- Execute `flexnetlsadmin.bat -server` *`licenseServer_baseURL`* `-config`

  (`flexnetlsadmin.bat` is located in the `enterprise` directory)

Important ▪ *(Linux only) To retrieve the VM UUID for local license servers that run on certain virtual Linux machines and that use the VM UUID as their hostid, you need to run the FNLicensingService (located in the "service" directory.*

Activating the License Rights for the Server

If the license server is not already registered on FlexNet Operations—this is the typical case—you must do so by sending one or more activation IDs in an activation request to the back office. For information about passing activation IDs and counts to the FlexNet Operations, see Activating Licenses on the Server in Chapter 4, License Server REST APIs.

Note ▪ *If you want to programmatically activate a server with a server host type that is not the default setting, ensure that the System Configuration setting **Allow Server without Activation IDs** is disabled in FlexNet Operations (under System > Configure > Embedded Devices > Capability Request Handling). For more information, see the FlexNet Operations User Guide.*

Important ▪ *If the license server has not been provisioned with any features, it responds to client requests with a 503 error with the error message "`Server instance <instanceID> is not ready`". This is expected behaviour because the server is not ready in this state.*

Using the REST API Option

To activate the license server using the `/activation_request` REST API, create a text file (`act-ids.json`) with contents similar to the following:

```
[{"id":"example-act-id-1", "copies":"100"}]
```

The "`id`" option specifies the activation ID and "`copies`" indicates the number of activation copies requested.

To pass the file as input to the license server and trigger the capability request, use a Curl command similar to the following:

```
curl -d @act-ids.json -X POST -H "Content-Type:application/json" licenseServer_baseURL/
    activation_request
```

This method is typically used for external customer code.

Note ▪ *You can also use a plugin for Chrome (such as "Advanced REST Client") or FireFox (such as "REST Client") to PUT or POST REST endpoints.*

To determine if the features are available to the license server, view the server's log or point a browser to the license server's REST `/features` endpoint.

Using the Administration Tools

You can also use the FlexNet Embedded local license server administration tool, FlexNet License Server Administrator (flexnetlsadmin), to perform an online activation (for example, **`flexnetlsadmin -server`** ***`licenseServer_baseURL`*** **`-activate -id`** ***`rightsID`*** **`-count`** ***`n`***) or an offline activation.

If administrative security is enabled, you might need to provide authorization credentials to perform this task. See the *FlexNet Embedded License Server Administration Guide* for more details about using these tools to activate licenses and provide authorization credentials when security is enabled.

Granting Client Requests

After the license server has obtained its pool of licenses, the FlexNet Embedded clients (that is, enterprise end-users) either obtain features or report metered-license usage by sending capability requests to the FlexNet Embedded local license server.

The following sections provide an overview of capability-request processing or usage reporting:

- Basic Capability-Request Processing
- Usage Capture

Basic Capability-Request Processing

The FlexNet Embedded clients check out features from the FlexNet Embedded local license server by sending capability requests that describe the *desired features*. The **CapabilityRequest** example in your FlexNet Embedded SDK provides a sample implementation for specifying desired features in the capability request, as shown here in an excerpt from the example in the FlexNet Embedded C SDK:

```
FlxCapabilityRequestAddDesiredFeature(capabilityRequest, "survey", "1.0", 1, error);
```

Clients should send their capability requests to the license server's `/request` endpoint, as described in the Processing Requests from Clients section in Chapter 4, License Server REST APIs. To demonstrate this using the **CapabilityRequest** example, you would run a command similar to the following to send a capability request for desired features to the license server:

```
capabilityrequest -server licenseServer_baseURL/request
```

The license server then sends the client a capability response containing the requested licenses. You can view the served licenses using the license server's REST `/clients` endpoint.

As an alternative to implementing FlexNet Embedded SDK functionality in your code to license your product, you can use a REST interface to perform capability exchanges (REST-driven licensing). See Chapter 4, License Server REST APIs, for more information about REST-driven licensing.

Default Behavior for Granting Licenses

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 grant 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 this client.

Exceptions to this behavior occur when named license pools or license reservations are used to pre-allocate licenses or when certain options that affect which licenses the license server grants are included in the request. See the next section, Other Considerations, for further direction.

If several features are available for checkout that only differ in their expiry date, the license server first serves the feature with the earliest expiry date that also satisfies the borrow period.

Other Considerations

The following are other points to consider about the licensing process:

- The license server can also process special options in a capability request that affect the scope of features sent back in the response. For more information, see License Checkout: Support for Special Capability-Request Options in Chapter 6, Advanced License Server Features.
- The license server can pre-allocate features to a group of client devices or users using named license pools. When the license server receives a capability request from a client, it tries to serve the feature counts according to the conditions specified in the model definition. For more information, see Allocating Licenses Using Named License Pools in Chapter 5, More About Basic License Server Functionality.
- The license server can pre-allocate features to individual client devices or users through the use of license reservations. The reservations are then used during the license-checkout process. For more information, see License Reservations in Chapter 5, More About Basic License Server Functionality.
- For more information about implementing licensing code in your product to generate capability requests and process responses, refer to the user guide for the FlexNet Embedded client SDK specific to your programming language.

Usage Capture

When usage capture is implemented, the FlexNet Embedded client code sets the operation type inside capability requests to either “report”, “request”, or “undo” to specify how the FlexNet Embedded local license server is to track usage for requested *metered features*.

For example, if the capability request is one that expects a response (that is, the request does not specify the “report” or “undo” operation), the FlexNet Embedded local license server sends a capability response with the requested metered-feature count, which the client then processes into trusted storage—and the server tracks as “consumed”. If the server cannot satisfy a requested count because it exceeds a metered feature’s usage cap, the response indicates as such.

If the capability request specifies a “report” operation, the requested metered-feature count is simply tracked by the license server as “consumed”. Generally no usage cap is defined, and no response is sent.

For more information about the different capability-request operations used in a usage-capture implementation, refer to the **UsageCaptureClient** example in your FlexNet Embedded SDK and to the accompanying user guide. For information about the license server’s role in usage capture, see Usage Capture and Management in Chapter 6, Advanced License Server Features.

Upgrading the License Server

You can upgrade the license server while retaining the older license server data and configuration.

Task ***To upgrade the local license server***

1. Create a backup of the following:
    - Current license server's installation directory
    - Trusted storage files (the `.ks`, `.db`, and `.0` files)
        - On Windows, the default storage location is `C:\Windows\ServiceProfiles\NetworkService\flexnetls\`*`producer_name`*
        - On Linux, the default storage location is `/var/opts/flexnetls/`*`producer_name`*
    - Current license server's configuration file (`producer-settings.xml`, in the "bin\tools" directory)
2. Uninstall the license server that is currently installed.

    For more information, see Uninstalling the License Server.

    Important *▪ Do not delete the trusted storage files. This will result in the loss of license server data and configurations.*
3. Install the latest version of the license server.

    For more information, see Installing and Starting the License Server.
4. Optional: If you want to continue to use the previous configuration, copy the `producer-settings.xml` file from the backup taken in Step 1 to the new installation directory.

If you are changing the trusted storage file location, you will lose the license server data unless you copy the trusted storage files (from the backup taken in Step 1) to the new location.

Uninstalling the License Server

Use the appropriate procedure to uninstall the license server:

- Uninstall the License Server Service
- Uninstall the License Server Running in Console Mode

Uninstall the License Server Service

Use the appropriate procedure to uninstall the license server service:

- Uninstall the Windows Service
- Uninstall the Linux Service

Uninstall the Windows Service

Use this procedure to uninstall the license server service on Windows.

Task ***To uninstall the license server service on Windows***

1. As an Administrator, open a command prompt and navigate to the license server's installation directory.
2. Execute the command `flexnetls.bat -stop` to stop the service.
3. Execute the command `flexnetls.bat -uninstall` to uninstall the license server service. (If you attempt to uninstall the service before stopping it, a message appears indicating that you need to stop the service first.)
4. To ensure there are no hanging instances or services, execute `sc delete FNLS-`*`producer_name`* as an administrator.

   Either the command will fail with the message `The specified service does not exist as an installed service`, or it will succeed with the message `[SC] DeleteService SUCCESS`. Both outcomes indicate that the service is no longer present.
5. Delete the files in the license server's installation folder.
6. Optionally, delete these files (listed here with their default locations):
   - The trusted storage files in `C:\Windows\ServiceProfiles\NetworkService\flexnetls\`*`producer_name`* (the `.ks`, `.db`, and `.0` files)
   - The log files in `C:\Windows\ServiceProfiles\NetworkService\flexnetls\`*`producer_name`*`\logs`

   Trusted storage and log locations are defined by the license server policies `server.trustedStorageDir` and `logging.directory`, respectively, the defaults for which are based on `${bases.dir}`. Depending on the values set for these policies on your server, your trusted storage and log files might be in locations different from those mentioned in this step. See Appendix A, Reference: Policy Settings for the License Server.

Uninstall the Linux Service

Use these steps to uninstall the license server service on Linux.

Task ***To uninstall the license server on Linux***

1. Run the following commands in this order:
   a. `sudo systemctl stop flexnetls-`*`producer`* (to shut down the service)
   b. `sudo systemctl disable flexnetls-`*`producer`* (to make the service ineligible to start on system reboot)
   c. `sudo rm /etc/systemd/system/flexnetls-`*`producer`*`.service`
   d. `sudo rm -r /etc/systemd/system/flexnetls-`*`producer`*`.service.d`
2. Optionally, remove these files (listed here with their default locations):
   - The trusted storage files in `/var/opts/flexnetls/`*`producer_name`* (the `.ks`, `.db`, and `.0` files)
   - The log files in `/var/opts/flexnetls/`*`producer_name`*`/logs`

***Important** ▪ Deleting the trusted storage files erases all the licenses and their usage data along with configuration information. This can be safely done in your test environment where there is no impact on production data. In some cases, Revenera support may ask you to delete trusted storage files to resolve an issue. For general upgrades, we do not recommend deleting the trusted storage files.*

Trusted storage and log locations are defined by the license server policies `server.trustedStorageDir` and `logging.directory`, respectively, the defaults for which are based on `${bases.dir}`. Depending on the values set for these policies on your server, your trusted storage and log files might be in locations different from those mentioned in this step. See Appendix A, Reference: Policy Settings for the License Server.

Uninstall the License Server Running in Console Mode

Use these procedures to uninstall the license server running in console mode.

Uninstall on the License Server in Console Mode on Windows

Use this procedure to uninstall the license server service in console mode on Windows.

Task ***To uninstall the license server running in console mode on Windows***

1. Close the command window running the license server.
2. Delete the files in the server's installation folder.
3. Optionally, delete the following files (listed here with their default locations):
    - The trusted storage files in `C:\Users\`*`user_name`*`\flexnetls\`*`producer_name`* (the `.ks`, `.db`, and `.0` files)
    - The log files in `C:\Users\`*`user_name`*`\flexnetls\`*`producer_name`*`\logs`

***Important** ▪ Deleting the trusted storage files erases all the licenses and their usage data along with configuration information. This can be safely done in your test environment where there is no impact on production data. In some cases, Revenera support may ask you to delete trusted storage files to resolve an issue. For general upgrades, we do not recommend deleting the trusted storage files.*

Uninstall the License Server in Console Mode on Linux

No "uninstalling the license server" in console mode is necessary. You simply press Ctrl + C to stop the license server in console mode. To uninstall the license server, uninstall the service, as described in Uninstall the Linux Service. (The license server running in console mode and as a service use the same trusted-storage and log locations.)

4

License Server REST APIs

You can provide the enterprise customer with an administration client (tool) that communicates with the license server REST APIs described in this chapter to perform license-server configuration and management operations and to retrieve server status information. These APIs use a REST-style interface and JSON messages (and in some cases, binary messages).

You can also use REST APIs to perform JSON capability exchanges without the need of FlexNet Embedded SDK functionality in your application code—a type of licensing called “REST-driven licensing”.

The following sections describe these APIs:

- Accessing the REST APIs
- Writing a REST API Client for a Secured License Server
- APIs for Basic License Server Operations
- APIs to Manage Clients
- APIs to Obtain Feature Information
- APIs to Manage Named License Pools
- APIs to Manage Reservations
- APIs to Show Statistics
- APIs to Manage Instance Configuration
- APIs to Perform Offline Binary Operations
- API to View Binding-Break Status
- APIs to Manage Administrative Security
- API for Enabling JSON Security for the Cloud Monetization API
- Online REST API Reference Documentation Available
- Exception Handling

Accessing the REST APIs

The REST APIs can be directly accessed by using HTTP utilities such as cURL, Postman, or browser plugins. Some of the endpoints return data that can be viewed in a browser; but to post data to an endpoint requires `cURL` (`http://curl.haxx.se/`) or a plugin for your browser such as **Advanced REST Client** for Chrome or **REST Client** for Firefox. The endpoints use either the content-type header `application/json` for text input or output or `application/octet-stream` for binary messages such as capability requests and responses. (The content-type header `text/plain` is supported for specific cases, such as importing mvel scripts.)

The following sections provide information about accessing the APIs:

- Base URLs
- Sample cURL Command
- Managing Large Amounts of Returned JSON Data

Base URLs

The REST APIs described in this chapter deal with configuring and managing license servers at the instance level. Consequently, most are prefixed with a base URL containing the instance ID for the local license server or the CLS instance. The base URL generally has the following format:

```
http://licenseServerHost/api/1.0/instances/instanceID
```

The following sections provide more information about the base URL:

- More About the Base URL
- HTTPS Protocol for Secured REST APIs
- “licenseServer_baseURL” as Base URL Designation

More About the Base URL

The following provides more detail about the base URL:

- For a CLS instance, the *licenseServerHost* part of the base URL is the address of the Cloud Licensing Service (which can optionally include a port number); the *instanceID* is the actual ID generated for the instance when it is created (data type: `string`). The path parameter *instanceID* has the data type `string`. The following shows a base URL for a CLS instance:

  `https://siteID.compliance.flexnetoperations.com/api/1.0/instances/XYZ10203040`

  Replace *siteID* with the specific site ID supplied by Revenera. This is usually your organization's DNS name, but can also be the tenant ID that was assigned to your organization when FlexNet Operations was first implemented. In this case, the site ID would have the format **flex*NNNN***.

  It is your responsibility to provide the base URL to the enterprise license-server administrator.

- For a local license server, *licenseServerHost* includes the server’s host name and port number in the format *licenseServerHostName*:*port*. The host name can be either a network address (such as `myserver.enterprise.com`) or an IP address. The *instanceID* is the tilde “~” character (data type: `string`). A prefix for a local license server looks similar to this:

```
http://11.11.1.111:7070/api/1.0/instances/~
```

- Exceptions to the base-URL format for a local license server occur for those REST APIs that must be accessed at the global level, not the instance level—for example, the `/hostids` APIs (see Viewing Instance hostids) and the `/configuration` APIs for certain policies (see APIs to Manage Instance Configuration). In such cases, the instance ID is not included in the URL, as shown here for `/hostids`:

```
http://licenseServerHostName/api/1.0/hostids
```

HTTPS Protocol for Secured REST APIs

When REST APIs are secured (that is, administrative security is enabled), use of the HTTPS protocol on the local license server is strongly recommended to protect disclosure of credentials during transmission across the network.

The base URL for a local license server might look similar to this:

```
https://11.11.1.111:1443/api/1.0/instances/~
```

The default port for a HTTPS URL is 443, unless one is supplied. The local license server defaults to 1443 to avoid privilege issues on Linux with port numbers less than 1024, but any appropriate port can be used. For more information about administrative security, see Administrative Security in Chapter 6, Advanced License Server Features.

To enforce the use of HTTPS when administrative security is enabled, set the `security.http.auth.enabled` license server policy to **`false`** in `producer-settings.xml`. See Policies to Configure Administrative Security in Chapter 6, Advanced License Server Features.

"licenseServer_baseURL" as Base URL Designation

In this chapter (and throughout this book), the base URL in commands is represented by the variable *licenseServer_baseURL*.

Sample cURL Command

An example cURL command to get the license server's version information might be the following. (Endpoints that accept input require the input to be in JSON format.)

```
curl -X GET --url licenseServer_baseURL/version
```

The returned information appears in JSON format, similar to the following:


```
{
    "LLS" : {
        "version" : "2018.12",
        "buildDate" : "2019-01-17T10:11:17Z",
        "buildVersion" : "244007",
        "branch" : "2018.12",
        "patch" : "0",
        "fneBuildVersion" : "229775",
        "serverInstanceID" : "GEDD2SNZK9RC"
    }
}
```

Managing Large Amounts of Returned JSON Data

APIs that can return large amounts of data, such as those related to features or clients, support pagination using the page and size query parameters to specify the (zero-based) page number and number of items per page, as in the following:

```
licenseServer_baseURL/features?page=1&size=25
```

The license server provides an HTTP Link header in order to make it simpler for external code to handle pagination. It outputs navigation links to the next, previous, first, and last pages of information. For example, if you want to traverse the features one at a time, you would issue a request that includes **page=1&size=1**. The JSON data is returned with an HTTP header like the following:

```
Link: <http://localhost:7070/api/1.0/instances/~/features?page=0&size=1>;rel="self",
<http://localhost:7070/api/1.0/instances/~/features?page=1&size=1>;rel="next",
<http://localhost:7070/api/1.0/instances/~/features?page=1&size=1>;rel="last"
```

The header does not include a `rel=”first”` link in this example because the header is actually located on the first page. Also, a custom HTTP header that returns the total number of elements is available. For this scenario, the header would return `Total-Elements: 2`. These HTTP headers are generated only for those endpoints that return JSON arrays and accept page and size attributes.

Writing a REST API Client for a Secured License Server

An administrative security facility is available on the local license server or CLS instance to control access to the REST APIs, helping to prevent unauthorized queries or operations on the license server. When this administrative security is enabled on the license server, credentials are needed to access REST API endpoints. For details, see Administrative Security in Chapter 6, Advanced License Server Features.

This section provides some assistance on the code flow needed to support administrative security in your custom REST API administration client:

- Using the Base URL
- Authorizing Access to Secured REST APIs
- Creating User Accounts

Using the Base URL

The REST API administration client should use the base URL for a license server instance when accessing the REST endpoints. See the previous section Base URLs for details. When administrative security is enabled on the license server, use of the HTTPS protocol is strongly recommended. Also note the exceptions to the base URL format when accessing the `/hostids` APIs and the `/configuration` APIs (see APIs for Basic License Server Operations).

Authorizing Access to Secured REST APIs

The administration client does not know ahead of time whether or not administrative security is enabled on the license server. Basically, if a user enters credentials, the client uses them to generate the authorization token needed to access the REST APIs. If no credentials are entered, the client proceeds to access the APIs without a token.

Step 1: Generate the Token

If credentials are entered, the administration client uses them to generate an authorization token. To do so, the client accesses the `/authorize` REST API with the `POST` method, providing the credentials in a request body (with content-type `application/json`). The following show a sample request body in JSON format:

```
{
"user" : "admin",
"password" : "Admin@111"
}
```

If the credentials match those in a current user account on the license server, the server returns a “200 OK” status code, as well as a response body that includes the token string and its expiration date, similar to this:

```
{
  "expires" : "2019-02-25T20:07:16.782Z",
  "token" : "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzUxMiJ9..." [token value truncated for purposes of the
example]}
```

Step 2: Provide the Token for Each API Call

Each time a REST endpoint is called, the client needs to provide the token string in an “Authorization” HTTP header, specifying the “Bearer” type:

```
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzUxMiJ9...  [token value truncated for purposes
of the example]
```

If a REST API does not require authorization, the token is ignored.

Creating User Accounts

The administrative security facility supports user accounts defined with specified roles that enable the user to perform specific operations. User accounts with ROLE_ADMIN privileges (that is, administrator accounts) can create and manage other user accounts on the license servers. The other accounts generally have limited administrative privileges, although a ROLE_ADMIN account can create other ROLE_ADMIN accounts. For more information about roles, see Administrative Security in Chapter 6, Advanced License Server Features.

To create, edit, or delete a user account, the code calls the `/users` REST API (which requires a administrator’s token to access). A JSON request body similar to this is used to create or update a user account:

```
{
  "password": "User003!",
  "roles": [
       "ROLE_READ"
  ],
  "user": "user003"
}
```

For more information about the `/users` REST API, see APIs to Manage User Accounts.

APIs for Basic License Server Operations

Basic license server operations are performed by the common REST API actions described in this section:

- API Summary
- Viewing Instance Details
- Viewing Instance hostids
- Processing Requests from Clients
- Sending Requests to the Back Office
- Activating Licenses on the Server
- Suspending or Resuming Operations
- Initiating On-demand Synchronization

API Summary

The following are the APIs that perform basic license server operations. Prefix each with the appropriate base URL for the license server, as described in Base URLs.

Table 4-1 ▪ Summary of License Server REST APIs

URI	Methods	Description
[*licenseServer_baseURL* only]	GET	Returns information about the specific license server instance.
/hostids	GET	(Accessed at the global level) Gets the available hostids on the license server instance.
/hostids/selected	GET	(Accessed at the global level) Returns the hostid currently used by the instance.
	PUT	(Accessed at the global level) Sets the hostid to be used by the instance. This endpoint is accessed at the global level. ***Important ▪*** *f you specify the server hostid using the API, the `server.hostType.order` property (if set) is ignored.*
/request	POST	Accepts a capability request from a FlexNet Embedded client device and responds with a capability response.

Table 4-1 ▪ Summary of License Server REST APIs (cont.)

URI	Methods	Description
/capability_request	`GET, POST`	Causes a new capability request to be sent to the back office to obtain license updates for the specified license server instance.
/activation_request	`POST`	Causes a new capability request containing the specified activation IDs to be sent to the back office for the license server instance.
/suspended	`PUT`	Sets the suspended/resumed state of the instance.
/sync_message	`GET, POST`	Initiates a online synchronization session outside of the online synchronizations run at set intervals on the license server instance. (This synchronization is also known as on-demand synchronization.)

Viewing Instance Details

The `/instances/{instanceID}` API returns information about a specific CLS instance or the local license server.

Note that `lastUpdateHostid` is only updated when licenses are mapped to the license server or after a successful capability polling attempt.

URI	`/instances/`*`instanceID`*
Method	`GET`
Query parameters	`syncinfo` The `/instances/`*`instanceID`* API can list an additional parameter (`recordsPendingSync`) that shows the number of client records on the license server that need to be synchronized to the back-office server (and thus possibly signals the need for synchronization). To include this property, specify the `syncinfo` parameter. Possible values are `true` and `false`. (The *Response* section that follows shows example output that includes this property.)
Request body	N/A
Error codes	`glsErr.serverNotFound` `glsErr.queryParamNeedsint`

Sample Response

```
{
  "id": 1,
  "pendingDeletion": false,
  "hostUUID": "74390cb0-6112-4c28-af09-2f8c7deb021e",
```

```
  "ready": true,
  "serverInstanceId": "T0FJQJ9TQBGR",
  "suspended": false,
  "requestId": "1",
  "tenantId": "",
  "publisherName": "demo",
  "lastUpdateTime": "2018-07-09T20:49:54.000Z",
  "failOverRole": "MAIN",
  "lastSyncTime": "2018-07-01T00:00:00.000Z",
  "lastUpdateHostid": {
    "hostidType": "ETHERNET",
    "hostidValue": "7C7A91BA832B"
  },
  "identityName": "demo-med-rsa",
  "lastClientDropTime": "2018-07-23T14:15:45.3292",
  "recordsPendingSync": 10
}
```


Viewing Instance hostids

The following APIs return information about the hostids available for the license server instance. Note that these APIs are accessed at a global level; therefore, you must access them with the following URL:

`http://licenseServerHostName/api/1.0/hostids` (or `/hostids/selected`)

All Available Hostids

The `/hostids` API returns a list of all hostids currently available to the license server instance.

URI	`/hostids`
Method	`GET`
Query parameters	N/A
Request body	N/A
Error codes	`glsErr.queryParamUnsupported` `glsErr.hostidNotFound`

Sample Response


```
{
  "selected" : {
    "type" : "ETHERNET",
    "value" : "5C514FF53778"
  },
  "hostids" : [ {
    "type" : "ETHERNET",
    "value" : "5C514FF53778"
  }, {
    "type" : "ETHERNET",
```

```
    "value" : "28D2444D94FC"
  } ]
}
```

Sample Response for Extended Hostid

```
{
  "hostids" : [ {
    "type" : "EXTENDED",
    "value" : "5C514FF53778-3-HOSTID1"
  }, {
    "type" : "EXTENDED",
    "value" : "28D2444D94FC-3-HOSTID1"
  }, {
    "type" : "ETHERNET",
    "value" : "5C514FF53778"
  }, {
    "type" : "ETHERNET",
    "value" : "28D2444D94FC"
  } ],
  "selected" : {
    "type" : "EXTENDED",
    "value" : "5C514FF53778-3-HOSTID1"
  }
}
```

Hostid Selected for Instance

The `/hostids/selected` API queries or sets the "selected" hostid for the license server. The `GET` method returns the current "selected" hostid—that is, the hostid currently used by the license server.

The `PUT` method designates a new "selected" hostid. It returns a JSON array of native hostids, plus the selected one (or returns 404 if no match exists in the available native hostids). The `PUT` method also triggers a capability request to the back office if the "selected" hostid is changed.

Important ▪ *f you specify the server hostid using the API, the `server.hostType.order` property (if set) is ignored.*

URI	`/hostids/selected`
Method	`GET`
Query parameters	N/A
Request body	N/A
Error codes	`glsErr.hostidNotFound`

Sample Response

```
{
  "type" : "ETHERNET",
  "value" : "5C514FF53778"
```

```
}}
```

URI	/hostids/selected
Method	PUT
Query parameters	N/A
Request body	{ "type" : "ETHERNET", "value" : "90B11C974E66" }
Error codes	glsErr.queryParamUnsupported glsErr.hostidNotFound

Sample Response

```
{
  "hostids" : [ { "type" : "ETHERNET", "value" : "90B11C974E66" } ],
  "selected" : { "type" : "ETHERNET", "value" : "90B11C974E66" }
}
```

Processing Requests from Clients

The /request API accepts a capability request from a FlexNet Embedded client device and responds back with a capability response.

Important ▪ *If the license server has not been provisioned with any features, it responds to client requests with a 503 error with the error message "Server instance <instanceID> is not ready". This is expected behavior because the server is not ready in this state.*

URI	/request
Method	POST
Query parameters	N/A
Request body	application/octet-stream
Error codes	glsErr.serverNotFound glsErr.optimisticLocking glsErr.backOfficePollingInProgress glsErr.mixingHostIds

Sending Requests to the Back Office

The /capability_request API causes a new capability request to be sent to the configured back office URL to check for license updates. When the operation is successful, the returned response is processed automatically (and no response body is sent). This process is a synchronous operation. A response body is available only when errors are generated.

***Note** ▪ This REST API is used to perform a capability exchange between the license server and the back office. Use “/request” if you want to perform a capability exchange between devices and the license server instance.*

URI	`/capability_request`
Method	`GET, POST`
Query parameters	`force` Forces a capability request to be sent immediately and forces the license server to relay the response from the back office to the client making the request. Possible values are `true` and `false`.
Request body	N/A

Error codes	`glsErr.serverNotFound`
	`glsErr.serverNoHostid`
	`glsErr.parsingServerIdentity`
	`glsErr.missingServerIdentity`
	`glsErr.signatureInvalid`
	`glsErr.optimisticLocking`
	`glsErr.sslMisconfigured`
	`glsErr.connectionFailed`
	`glsErr.connectionFailedStatus`
	`glsErr.connectionRefused`
	`glsErr.licenseFailsAnchor`
	`glsErr.lfsError`
	`glsErr.lfsWrongMessageType`
	`glsErr.lfsCapSessionFailed`
	`glsErr.serviceBusyDueToReservations`
	`glsErr.assigningReservations`

Sample Response

***Note** ▪ This response is found in the logs.*

```
MessageType="Capability response"
ServerIDType=String
ServerID=BACK_OFFICE
SourceIds=BACK_OFFICE
Lifetime=86400
ResponseTime="Mar 3, 2019 11:20:44 AM"
RequestHostID=6C626DA08750
RequestHostIDType=Ethernet
HostUUID=dfb867a4-4cac-4e16-96c6-9a353231ed17
MachineType=Unknown
EnterpriseID=35331653
License=(Name=MeteredCappedRe Vendor=cmsperf5 Version=1.0
  Expires=1-jan-2020 Count=10 Metered Reusable
  ServerHostID=6C626DA08750/Ethernet HostID=ANY Issued=2-mar-2019
```

```
  Start=1-jan-2019 VendorString=42F9-634D-9B31-22CE
  Notice=AutomationAccount4 Issuer=FlexManufacturer
  SerialNumber=c8c83a8d-a1b5-4eac-9b94-ac38f8d21275)
License=(Name=MeteredUncappedRe Vendor=cmsperf5 Version=1.0
  Expires=1-jan-2020 Count=10 Metered Reusable UncappedOverdraft
  ServerHostID=6C626DA08750/Ethernet HostID=ANY Issued=2-mar-2019
  Start=1-jan-2019 VendorString=5132-B427-99A3-3DD1
  Notice=AutomationAccount4 Issuer="FlexNet Test Site"
  SerialNumber=f8414bcd-c4ea-4c7b-a997-3bef2c04750e)
```

Activating Licenses on the Server

The /activation_request API causes a new capability request, with an included set of activation IDs (rights IDs) and counts, to be sent to the configured back-office URL for the specified license server instance. When the operation is successful, the returned response is processed automatically (and no response body is sent). (This is a synchronous operation.) A response body is available only when errors are generated.

***Note** ▪ This REST API is for performing a capability exchange between the license server and the back office. Use "/request" if you want to perform a capability exchange between devices and the license server instance.*

URI	/activation_request
Method	POST
Query parameters	force Forces an activation request to be sent immediately and forces the license server to relay the response from the back office to the client making the request. Possible values are `true` and `false`.

Request body	`[ {"id" : "li1", "copies" : 3}, ... ]`
	
	Multiple id entries may be included in the request. If the copies field is omitted in an entry, it defaults to one copy.
	
	For a given id entry, you can also include the "partial" option to obtain all available copies of a rights ID if the requested count cannot be satisfied:
	
	`[ {"id" : "li1", "copies" : 3, "partial" : "true"}, ... ]`

Error codes	glsErr.serverNotFound glsErr.serverNoHostid glsErr.parsingServerIdentity glsErr.missingServerIdentity glsErr.signatureInvalid glsErr.optimisticLocking glsErr.sslMisconfigured glsErr.connectionFailed glsErr.connectionFailedStatus glsErr.connectionRefused glsErr.licenseFailsAnchor glsErr.lfsError glsErr.lfsWrongMessageType glsErr.lfsCapSessionFailed glsErr.serviceBusyDueToReservations glsErr.assigningReservations

Sample Response

On success, there is no response. If there are problems processing the request, the response will contain a list of status messages:

```
[ { "message" : "Specified rights ID is invalid: li1", "code" : "RIGHTS_ID_INVALID", "detail" : "li1" } ]
```

Suspending or Resuming Operations

The `/suspended` API changes to the suspended or resumed state of the license server are both updates of this resource. A suspended server retains its most recent state but responds to client requests either by just closing the connection or with “503 Service Unavailable”.

URI	/suspended
Method	PUT
Query parameters	N/A
Request body	true or false
Error codes	glsErr.serverNotFound

Sample Response

No response is returned. The connection is closed on completion.

Initiating On-demand Synchronization

The `/sync_message` API initiates an online synchronization session from the license server to the back office at any desired time, outside of the online synchronization sessions run at set intervals.

URI	/sync_message

Method	GET or POST
Query parameters	N/A
Request body	N/A
Error codes	glsErr.serverNotFound glsErr.parsingServerIdentity glsErr.missingServerIdentity glsErr.optimisticLocking

Sample Response

For a successful GET or POST, no response is returned.

If the method fails, the response contains the error code and its corresponding message and a list of error arguments that varies based on the error. For example, you might see:

```
{
  "key" : "glsErr.serverNotFound",
  "message" : "Server instance not found: 100",
  "arguments" : [ "100" ]
}
```

APIs to Manage Clients

These APIs show the FlexNet Embedded clients that have features currently checked out from the specified license server.

Table 4-2 • License Server REST APIs Used to Manage FlexNet Embedded Clients

URI	Method	Description
/clients	GET	Gets currently active (served) clients.
/clients.stream	GET	Asynchronously returns active clients as individual newline-delimited JSON (NDJSON) objects (for local license servers only).
/clients/{id}	GET	Gets the information for the client specified by its system-generated ID.
/clients/{id}	DELETE	Deletes a FlexNet Embedded client record specified by its system-generated ID.
/clients/{id}/features	GET	Gets the feature information for the client specified by its system-generated ID.

Query parameters can be added to set a new page size, show a specific page, or filter records by a specific hostid (of a specific hostid type if necessary) or host UUID. The includeAll parameter displays a history of client records that includes all current client records and all previous client records awaiting synchronization.

The following sections describe these APIs along with some of the parameter combinations:

- Viewing All Clients
- Retrieving Client Data as NDJSON Objects
- Viewing a Specified Client Record
- Viewing Features for a Specified Client Record
- Removing a Specified Client
- Viewing Client Record History for Specified Hostid
- Viewing Client Record History Including Features for a Specified hostid

Viewing All Clients

The `/clients` API returns a page of current clients for the license server.

***Note** ▪ If you need to query large tables, it is recommended that you use the `/clients.stream` API instead of the `/clients` API (local license servers only).*

URI	`/clients`
Method	`GET`
Query parameters	`current, hostid, hostidtype, uuid, page, size, features, includeAll, includeUsageExpiry` For more information, see *Query Parameters*, below.
Request body	N/A

Error codes	`glsErr.serverNotFound`
	`glsErr.queryParamNeedsInt`
	`glsErr.queryParamTooSmall`
	`glsErr.queryParamTooLarge`
	`glsErr.queryParamUnsupported`
	`glsErr.invalidUUID`
	`glsErr.bothUuidAndHostId`
	`glsErr.HostIdTypeWithoutHostId`

Query Parameters

The following are optional parameters:

`current`	When set to `true`, returns only the currently active client record for the host. Possible values are `true` and `false`.

features	When set to **true**, returns all current clients and the features checked out for each client. Possible values are **true** and **false**. See also Viewing All Current Clients and Their Features.
hostid	Filters records by a specific hostid. Must be a string. See also Viewing Client Record History for Specified Hostid and Viewing Client Record History Including Features for a Specified hostid.
hostidtype	Filters records by a specific hostid type. Must be a string.
includeAll	When set to **true**, returns a history of client records that includes all current client records and all previous client records awaiting synchronization. Possible values are **true** and **false**.
page	Retrieves a specific page. Must be a number. See also Managing Large Amounts of Returned JSON Data.
size	Specifies how many records per page should be returned. Must be a number. See also Managing Large Amounts of Returned JSON Data.
uuid	Filters records by a specific host UUID. Must be a string.
includeUsageExpiry	When set to **true**, the response includes the usageExpiry field for each feature, which shows the date and time when the feature expires on the client. Possible values are **true** and **false**.

Sample Response

```
[ {
  "id" : 25,
  "trusted" : true,
  "servedStatus" : "NORMAL",
  "updateTime" : "2018-10-13T19:09:10.177Z",
  "machineType" : "UNKNOWN",
  "requestHostid" : {
    "hostidValue" : "User-3",
    "hostidType" : "STRING"
  },
  "serverHostid" : {
    "hostidValue" : "7C7A91BA832B",
    "hostidType" : "ETHERNET"
  },
  "requestOperation" : "REQUEST",
  "correlationId" : "b7254007-aef7-4807-a753-87f55252a36c",
  "collectedHostIds" : "STRING User-3",
  "hostid" : {
    "hostidValue" : "User-3",
    "hostidType" : "STRING"
  },
  "expiry" : "2018-10-14T19:09:10.177Z"
}, {
  "id" : 28,
```

```
      "trusted" : true,
      "servedStatus" : "NORMAL",
      "updateTime" : "2018-10-13T21:16:43.772Z",
      "machineType" : "UNKNOWN",
      "requestHostid" : {
        "hostidValue" : "User-2",
        "hostidType" : "STRING"
      },
      "serverHostid" : {
        "hostidValue" : "7C7A91BA832B",
        "hostidType" : "ETHERNET"
      },
      "requestOperation" : "REQUEST",
      "correlationId" : "6bc685d5-3db6-4236-9b46-9efc8ffeb590",
      "collectedHostIds" : "STRING User-2",
      "hostid" : {
        "hostidValue" : "User-2",
        "hostidType" : "STRING"
      },
      "expiry" : "2018-10-14T21:16:43.772Z"
    } ]
```


JSON Response Details

The following table describes the data types of fields that might be returned in the JSON response.

"id"	Number
"hostid"	JSON Object
"hostidValue"	String
"hostidType"	String
"updateTime"	Date and time in ISO format
"servedStatus"	String
"machineType"	String
"trusted"	Boolean
"correlationId"	String
"collectedHostIds"	String
"requestOperation"	String
"requestHostid"	JSON object
"hostidValue"	String
"hostidType"	String
"serverHostid"	JSON object
"hostidValue"	String
"hostidType"	String
"expiry"	String (could be either the text 'permanent' or date in 'yyyy-mm-dd' format)
"usages"	List
"id"	Number
"usageKind"	String
"feature"	JSON object
"id"	Number

```
        "type"                      String
        "featureName"               String
        "featureVersion"            String
        "expiry"                    String (could be either the text 'permanent' or date in 'yyyy-mm-dd' format)
        "featureCount"              String/number (string if uncounted feature, else number)
        "overdraftCount"            String/number (string if uncapped feature, else number)
        "used"                      Number
        "vendorString"              String
        "notice"                    String
        "featureId"                 String
        "starts"                    String (could be either the text 'permanent' or date in 'yyyy-mm-dd' format)
        "featureKind"               String
        "vendor"                    String
        "meteredUndoInterval"       Number
        "meteredReusable"           Boolean
        "receivedTime"              Date and time in ISO format
        "selectorsDictionary"       JSON of  <String,Object>
        "reserved"                  Number
        "concurrent"                Boolean
        "metered"                   Boolean
        "uncounted"                 Boolean
        "uncappedOverdraft"         Boolean
    "useCount"                      Number
    "usageExpiry"                   Date and time in ISO format
    "reservedCount"                 Number
    "partitionName"                 String
    "featureSlice"                  JSON Object
        "id"                        Number
        "slice"                     String/number (string if uncapped feature is present, else number)
        "used"                      Number
        "orphaned"                  Boolean
        "uncounted"                 Boolean
        "computedCount"             String/number (string if uncapped feature is present, else number)
        "requested"                 String
        "status"                    String
```

Viewing All Current Clients and Their Features

The `features` parameter (`/clients?features=true`) returns all current clients and the features checked out for each client.

The following snippet, truncated for purposes of this document, highlights the following information:

- One client, identified with id 22 and hostid User-1, currently has 10 counts of f2 and 5 counts of f4 checked out.
- Another client, identified with id 25 and hostid User-3, currently has 4 counts of f3 checked out.

Note that the features checked out for a given client are listed under the usages section for that client; the checked out count for each feature in the usages section is shown as the useCount value for that feature.

This type of information is repeated for each current client.

Sample Response


```
[ {
  "id" : 22,
  "trusted" : true,
  "servedStatus" : "NORMAL",
  "updateTime" : "2018-10-12T19:34:29.518Z",
  "usages" : [ {
    "id" : 39,
    "reservedCount" : 0,
    "feature" : {
      "id" : 17,
      "starts" : "2018-10-11",
      "used" : 10,
      "featureVersion" : "2.0",
      "meteredReusable" : false,
      "featureKind" : "ORPHANED_FEATURE",
      "featureCount" : 100,
      "borrowInterval" : 604800,
      "issued" : "2018-10-12",
      "meteredUndoInterval" : 0,
      "receivedTime" : "2018-10-12T19:15:52.000Z",
      "featureName" : "f2",
      "type" : "CONCURRENT",
      "featureId" : "vcrAFaCMXROTkh62DqZ0gw",
      "overdraftCount" : 0,
      "renewInterval" : 15,
      "vendor" : "demo",
      "expiry" : "permanent",
      "uncounted" : false,
      "concurrent" : true,
      "reserved" : 0,
      "metered" : false,
      "uncappedOverdraft" : false
    },
    "usageKind" : "CONCURRENT_USAGE",
    "useCount" : 10
  }, {
    "id" : 38,
    "reservedCount" : 0,
    "feature" : {
      "id" : 19,
      "starts" : "2018-10-11",
      "used" : 5,
      "featureVersion" : "1.0",
      "meteredReusable" : false,
      "featureKind" : "ORPHANED_FEATURE",
```

```
        "featureCount" : 10,
        "borrowInterval" : 604800,
        "issued" : "2018-10-12",
        "meteredUndoInterval" : 0,
        "receivedTime" : "2018-10-12T19:15:52.000Z",
        "entitlementExpiration" : "2020-07-04",
        "featureName" : "f4",
        "type" : "CONCURRENT",
        "featureId" : "qAKQDQVGgS+sg/oav0eBBA",
        "overdraftCount" : 0,
        "renewInterval" : 15,
        "vendor" : "demo",
        "expiry" : "2021-11-30",
        "uncounted" : false,
        "concurrent" : true,
        "reserved" : 0,
        "metered" : false,
        "uncappedOverdraft" : false
      },
      "usageKind" : "CONCURRENT_USAGE",
      "useCount" : 5
    } ],
    "machineType" : "UNKNOWN",
    "requestHostid" : {
      "hostidValue" : "User-1",
      "hostidType" : "STRING"
    },
    "serverHostid" : {
      "hostidValue" : "7C7A91BA832B",
      "hostidType" : "ETHERNET"
    },
    "requestOperation" : "REQUEST",
    "correlationId" : "12582448-53f0-4611-bfbf-3be985d0c5de",
    "collectedHostIds" : "STRING User-1",
    "hostid" : {
      "hostidValue" : "User-1",
      "hostidType" : "STRING"
    },
    "expiry" : "2018-10-13T19:34:29.518Z"
  }, {
    "id" : 25,
    "trusted" : true,
    "servedStatus" : "NORMAL",
    "updateTime" : "2018-10-13T19:09:10.177Z",
    "usages" : [ {
      "id" : 45,
      "reservedCount" : 0,
      "feature" : {
        "id" : 20,
        "starts" : "2018-10-12",
        "used" : 7,
        "featureVersion" : "1.0",
        "meteredReusable" : false,
        "featureKind" : "NORMAL_FEATURE",
        "featureCount" : 10,
        "borrowInterval" : 604800,
```

```
        "issued" : "2018-10-13",
        "meteredUndoInterval" : 0,
        "receivedTime" : "2018-10-13T18:57:45.000Z",
        "entitlementExpiration" : "2020-12-31",
        "featureName" : "f3",
        "type" : "CONCURRENT",
        "featureId" : "DGr/2lmfZ8vw44T4+1MbtQ",
        "overdraftCount" : 0,
        "renewInterval" : 15,
        "vendor" : "demo",
        "expiry" : "2021-01-31",
        "uncounted" : false,
        "concurrent" : true,
        "reserved" : 0,
        "metered" : false,
        "uncappedOverdraft" : false
      },
      "usageKind" : "CONCURRENT_USAGE",
      "useCount" : 4
    .....
    .....
    ...],
    "machineType" : "UNKNOWN",
    "requestHostid" : {
      "hostidValue" : "User-3",
      "hostidType" : "STRING"
    } .....
  } ]
```


JSON Response Details

For information about the data types of fields in the JSON response, see Viewing All Clients.

Retrieving Client Data as NDJSON Objects

The `/clients.stream` API asynchronously returns client data as individual newline-delimited JSON (NDJSON) objects. The content type is "application/x-ndjson", not "application/json". For information about the NDJSON standard, see https://github.com/ndjson/ndjson-spec.

Note that the `/clients.stream` API is available only for the local license server.

The `/clients.stream` API returns only active clients.

URI	`/clients.stream`
Method	`GET`
Query parameters	`sort, direction, limit`
Request body	N/A
Error codes	`glsErr.userAccessDenied`

It is recommended that you use the `/clients.stream` API instead of the `/clients` API if you need to query large client tables. The Appendix F, Consuming Streaming Content, contains example code which shows how to deserialise the content to a representation which enables the client to consume the streamed content one line at a time.

Query Parameters

The following are optional parameters:

`sort`	Sorts the returned data by client id and update time. Possible values are **`id`** and **`updateTime`**. By default, data is sorted by update time.
`direction`	Sorts the returned data in ascending or descending order. Possible values are **`asc`** and **`desc`**.
`limit`	Specifies the maximum number of client records to be retrieved. Must be a number.

Sample Response

For a successful `GET` method, all active clients for the license server instance are returned as individual newline-delimited JSON objects.


```
{
    "id": 8,
    "hostid": {
        "hostidValue": "h1",
        "hostidType": "STRING"
    },
    "updateTime": "2021-02-25T11:30:05.204Z",
    "servedStatus": "NORMAL",
    "machineType": "UNKNOWN",
    "trusted": true,
    "correlationId": "138e26de-4922-484a-9224-69d22a4c7c49",
    "collectedHostIds": "STRING h1",
    "requestOperation": "REQUEST",
    "requestHostid": {
        "hostidValue": "h1",
        "hostidType": "STRING"
    },
    "serverHostid": {
        "hostidValue": "14ABC537C170",
        "hostidType": "ETHERNET"
    },
    "expiry": "2021-03-01T11:30:05.204Z"
}
{
    "id": 7,
    "hostid": {
        "hostidValue": "h4",
        "hostidType": "STRING"
    },
    "updateTime": "2021-02-25T10:44:46.069Z",
```

```
    "servedStatus": "NORMAL",
    "machineType": "UNKNOWN",
    "trusted": true,
    "correlationId": "3a0a5d0f-bf0a-4d86-9c1d-9f4b9ccce645",
    "collectedHostIds": "STRING h4",
    "requestOperation": "REQUEST",
    "requestHostid": {
        "hostidValue": "h4",
        "hostidType": "STRING"
    },
    "serverHostid": {
        "hostidValue": "14ABC537C170",
        "hostidType": "ETHERNET"
    },
    "expiry": "2021-03-01T10:44:46.069Z"
}
```

Viewing a Specified Client Record

The `/clients/{id}` API returns information on the specific client record identified by its system-generated ID. The path parameter `{id}` must be of data type `number`.

URI	`/clients/{id}`
Method	`GET`
Query parameters	N/A
Request body	N/A
Error codes	`glsErr.serverNotFound` `glsErr.clientNotFound`

Sample Response

```
{
  "id" : 2,
  "machineType" : "UNKNOWN",
  "requestOperation" : "REQUEST",
  "servedStatus" : "NORMAL",
  "correlationId" : "e63434bd-aa97-4c59-86a3-706e29953bc6",
  "updateTime" : "2018-07-06T22:37:08.489Z",
  "collectedHostIds" : "STRING 222201",
  "trusted" : false,
  "serverHostid" : {
    "hostidValue" : "7C7A91BA832B",
    "hostidType" : "ETHERNET"
  },
  "hostid" : {
    "hostidValue" : "222201",
    "hostidType" : "STRING"
  },
  "requestHostid" : {
```

```
      "hostidValue" : "222201",
      "hostidType" : "STRING"
    },
    "expiry" : "2018-07-13T22:37:08.489Z"
  }
```

JSON Response Details

The following table describes the data types of fields that might be returned in the JSON response.

"id"	Number
"hostid"	JSON object
"hostidValue"	String
"hostidType"	String
"updateTime"	Date and time in ISO format
"servedStatus"	String
"machineType"	String
"trusted"	Boolean
"correlationId"	String
"collectedHostIds"	String
"requestOperation"	String
"requestHostid"	JSON object
"hostidValue"	String
"hostidType"	String
"serverHostid"	JSON object
"hostidValue"	String
"hostidType"	String
"expiry"	String (could be either the text ‘permanent’ or date in ‘yyyy-mm-dd’ format)

Viewing Features for a Specified Client Record

The `/clients/{id}/features` API returns the list of checked-out features for a specific client record identified by its system-generated ID. The path parameter `{id}` must be of data type `number`.

URI	`/clients/{id}/features`
Method	`GET`
Query parameters	`page, size, includeUsageExpiry` For more information, see *Query Parameters*, below.
Request body	N/A

Error codes	glsErr.serverNotFound glsErr.clientNotFound glsErr.queryParamNeedsInt glsErr.queryParamTooSmall glsErr.queryParamTooLarge glsErr.queryParamUnsupported

Query Parameters

The following are optional parameters:

page	Retrieves a specific page. Must be a number. See also Managing Large Amounts of Returned JSON Data.
size	Specifies how many records per page should be returned. Must be a number. See also Managing Large Amounts of Returned JSON Data.
includeUsageExpiry	When set to `true`, the response includes the usageExpiry field for each feature, which shows the date and time when the feature expires on the client. Possible values are `true` and `false`.

Sample Response


```
[ {
  "id" : 1,
  "usageKind" : "CONCURRENT_USAGE",
  "useCount" : 5,
  "feature" : {
    "id" : 3,
    "issued" : "2018-07-06",
    "meteredUndoInterval" : 0,
    "type" : "CONCURRENT",
    "meteredReusable" : false,
    "featureId" : "u65ZlP9xojax9KjoNV8CFw",
    "vendor" : "demo",
    "borrowInterval" : 604800,
    "featureName" : "f3",
    "renewInterval" : 15,
    "used" : 5,
    "featureCount" : 10,
    "expiry" : "2019-01-31",
    "featureVersion" : "1.0",
    "featureKind" : "NORMAL_FEATURE",
    "overdraftCount" : 0,
    "receivedTime" : "2018-07-06T22:15:52.000Z",
    "starts" : "2018-07-05",
    "entitlementExpiration" : "2018-12-31",
    "metered" : false,
    "uncappedOverdraft" : false,
    "reserved" : 0,
    "concurrent" : true,
    "uncounted" : false
  },
```

```
      "reservedCount" : 0
    }, {
      "id" : 2,
      "usageKind" : "CONCURRENT_USAGE",
      "useCount" : 20,
      "feature" : {
        "id" : 2,
        "issued" : "2018-07-06",
        "meteredUndoInterval" : 0,
        "type" : "CONCURRENT",
        "meteredReusable" : false,
        "featureId" : "z3vg4xKuAy0OXrPb+9v2ow",
        "vendor" : "demo",
        "borrowInterval" : 604800,
        "featureName" : "f2",
        "renewInterval" : 15,
        "used" : 35,
        "featureCount" : 100,
        "expiry" : "permanent",
        "featureVersion" : "2.0",
        "featureKind" : "NORMAL_FEATURE",
        "overdraftCount" : 0,
        "receivedTime" : "2018-07-06T22:15:52.000Z",
        "starts" : "2018-07-05",
        "metered" : false,
        "uncappedOverdraft" : false,
        "reserved" : 0,
        "concurrent" : true,
        "uncounted" : false
      },
      "reservedCount" : 0
    } ]
```


JSON Response Details

The table below describes the data types of fields returned in the JSON response. Each object has the following structure:

"id"	Number
"usageKind"	String
"feature"	JSON object
"id"	Number
"type"	String
"featureName"	String
"featureVersion"	String
"expiry"	String (could be either the text ‘permanent’ or date in ‘yyyy-mm-dd’ format)
"featureCount"	String/number (string if uncounted feature, else number)
"overdraftCount"	String/number (string if uncapped feature, else number)
"used"	Number
"vendorString"	String
"notice"	String
"featureId"	String

"starts"	String (could be either the text 'permanent' or date in 'yyyy-mm-dd' format)
"featureKind"	String
"vendor"	String
"meteredUndoInterval"	Number
"meteredReusable"	Boolean
"receivedTime"	Date and time in ISO format
"selectorsDictionary"	JSON of <String,Object>
"reserved"	Number
"concurrent"	Boolean
"metered"	Boolean
"uncounted"	Boolean
"uncappedOverdraft"	Boolean
"useCount"	Number
"reservedCount"	Number
"partitionName"	String
"featureSlice"	JSON object
"id"	Number
"slice"	String/number (string if uncapped feature is present, else number)
"used"	Number
"orphaned"	Boolean
"uncounted"	Boolean
"computedCount"	String/number (string if uncapped feature is present, else number)
"requested"	String
"status"	String

Removing a Specified Client

The `/clients/{id}` API deletes the FlexNet Embedded client specified by any of its system-generated client record IDs. Usage data for the client, however, is still retained until it is successfully synced to the back office. The path parameter `{id}` must be of data type `number`.

URI	`/clients/{id}`
Method	`DELETE`
Query parameters	N/A
Request body	N/A

Error codes	`glsErr.serverNotFound`
	`glsErr.clientNotFound`
	`glsErr.dropClientDelayEnforced`
	`glsErr.dropClientDisallowed`
	`glsErr.dropClientFailed`

For a successful `DELETE`, the status code 410 is returned, but no response is provided.

If the status code 429 or 400 is returned, this can be due to the license server policy setting `licensing.dropClientEnforcedDelay`. When this setting is specified, the following behavior can occur:

- If set to a time value (other than `0`): A delay between client deletion requests is enforced. A message specifies when the next client can be deleted (error code `glsErr.dropClientDelayEnforced`).
- If set to `DROP_CLIENT_DISALLOWED`: Deletion of client records is not allowed (error code `glsErr.dropClientDisallowed`).

The license server policy setting `licensing.dropClientEnforcedDelay` is only editable by the producer.

Status code 404 indicates that the specified client could not be found.

Viewing Client Record History for Specified Hostid

The `hostid` parameter enables you to view the client record history—current and previous records—for a specific hostid, each record for the hostid identified by a different “id”.

If no hostid type is specified, it is assumed that the hostid is of type STRING.

Example:**`/clients?hostid=user1`**

To view the client history for a specified hostid that is not of type STRING, you must specify the hostid type.

Example: **`clients?hostid=6C626DA08750&hostidtype=ETHERNET`**

Sample Response


```
[ {
  "id" : 10,
  "trusted" : true,
  "servedStatus" : "NORMAL",
  "updateTime" : "2018-10-05T16:12:45.452Z",
  "machineType" : "UNKNOWN",
  "requestHostid" : {
    "hostidValue" : "user1",
    "hostidType" : "STRING"
  },
  "serverHostid" : {
    "hostidValue" : "7C7A91BA832B",
    "hostidType" : "ETHERNET"
  },
  "requestOperation" : "REQUEST",
  "correlationId" : "d32a6605-3380-4159-bcdb-0515888fd781",
  "collectedHostIds" : "STRING user1",
  "hostid" : {
    "hostidValue" : "user1",
    "hostidType" : "STRING"
  },
  "expiry" : "2018-10-06T16:12:45.452Z"
}, {
  "id" : 9,
  "trusted" : true,
  "servedStatus" : "NORMAL",
  "updateTime" : "2018-10-05T14:28:16.620Z",
  "machineType" : "UNKNOWN",
  "requestHostid" : {
```

```
      "hostidValue" : "user1",
      "hostidType" : "STRING"
    },
    "serverHostid" : {
      "hostidValue" : "7C7A91BA832B",
      "hostidType" : "ETHERNET"
    },
    "requestOperation" : "REQUEST",
    "correlationId" : "d05d58dc-a243-49a0-b90e-224b121c2358",
    "collectedHostIds" : "STRING user1",
    "hostid" : {
      "hostidValue" : "user1",
      "hostidType" : "STRING"
    },
    "expiry" : "2018-10-05T14:28:16.620Z"
  } ]
```

JSON Response Details

For information about the data types of fields in the JSON response, see Viewing All Clients.

Viewing Client Record History Including Features for a Specified hostid

The `features` parameter combined with the `hostid` parameter (for example, `/clients?features&hostid=User-2`) enables you to view the client record history—current and previous client records—for a specific hostid along with the features associated for each record. Each record for the hostid is identified by a different "id".

The following snippet, truncated for purposes of this document, shows two records ("ids" 26 and 23) in the client record history for hostid `User-2`. Each record shows the features checked out (under `usages`) at the update time for the record.

Sample Response

```
  [ {
    "id" : 26,
    "trusted" : true,
    "servedStatus" : "NORMAL",
    "updateTime" : "2018-10-13T19:09:32.872Z",
    "usages" : [ {
      "id" : 48,
      "reservedCount" : 0,
      "feature" : {
        "id" : 22,
        "starts" : "2018-10-12",
        "used" : 1,
        "featureVersion" : "2.0",
        "meteredReusable" : false,
        "featureKind" : "NORMAL_FEATURE",
        "featureCount" : 100,
        "borrowInterval" : 604800,
        "issued" : "2018-10-13",
        "meteredUndoInterval" : 0,
        "receivedTime" : "2018-10-13T18:57:45.000Z",
```

```
            "featureName" : "f2",
      .....
      .....
        } ],
        "machineType" : "UNKNOWN",
        "requestHostid" : {
          "hostidValue" : "User-2",
          "hostidType" : "STRING"
        },
        "serverHostid" : {
          "hostidValue" : "7C7A91BA832B",
          "hostidType" : "ETHERNET"
        },
        "requestOperation" : "REQUEST",
        "correlationId" : "4be08ff8-ad19-4ccb-89d0-15086fd6ed28",
        "collectedHostIds" : "STRING User-2",
        "hostid" : {
          "hostidValue" : "User-2",
          "hostidType" : "STRING"
        },
        "expiry" : "2018-10-14T19:09:32.872Z"
      }, {
        "id" : 23,
        "trusted" : true,
        "servedStatus" : "NORMAL",
        "updateTime" : "2018-10-13T19:06:41.213Z",
        "usages" : [ {
          "id" : 41,
          "reservedCount" : 0,...
      ...
        } ],
        "machineType" : "UNKNOWN",
        "requestHostid" : {
          "hostidValue" : "User-2",
          "hostidType" : "STRING"
        },
        "serverHostid" : {
          "hostidValue" : "7C7A91BA832B",
          "hostidType" : "ETHERNET"
        },
        "requestOperation" : "REQUEST",
        "correlationId" : "f78bf841-cd1d-467d-86a4-4fa6d58da7a0",
        "collectedHostIds" : "STRING User-2",
        "hostid" : {
          "hostidValue" : "User-2",
          "hostidType" : "STRING"
        },
        "expiry" : "2018-10-13T19:06:41.213Z"
      } ]
```


JSON Response Details

For information about the data types of fields in the JSON response, see Viewing All Clients.

APIs to Obtain Feature Information

These APIs show those features (license rights) that the specified license server is currently serving to clients.

Table 4-3 ▪ License Server REST APIs Used to Obtain Feature Information

URI	Method	Description
/features	GET	Returns the first page of feature information. Add query parameters to set page size and offset.
/features	HEAD	Returns the number of features associated with the instance.
/features/summaries	GET	Returns a summary of feature information, grouped by feature name.
/features.stream	GET	Asynchronously returns features as individual newline-delimited JSON (NDJSON) objects (for local license servers only).
/features/{id}	GET	Returns information about a specific feature.
/features/{id}/clients	GET	Returns usage information about a specific feature. Add query parameters to set page size and offset.
/overages	GET	Returns overage count and expiration information. Add query parameters to set page size and offset.

The following sections describe these APIs:

- Viewing All Features
- Viewing A Summary Of Features
- Retrieving Feature Data as NDJSON Objects
- Viewing Only Active Features
- Viewing Features in an Expiration Grace Period
- Viewing Number of Features
- Viewing Specified Feature
- Viewing Feature Usage
- Listing Overages

Viewing All Features

The /features API returns the first page of features served by this instance, including the total served counts, used counts, and overage counts. Query parameters may be added to filter the results.

***Note** ▪ If you need to query large tables, it is recommended that you use the /features.stream API instead of the /features API (local license servers only).*

URI	/features
Method	GET
Query parameters	activeOnly, includeSlices, inGracePeriod, name, page, size For more information, see *Query Parameters*, below.
Request body	N/A
Error codes	glsErr.serverNotFound glsErr.queryParamNeedsInt glsErr.queryParamTooSmall glsErr.queryParamTooLarge glsErr.queryParamUnsupported

Query Parameters

The following are optional parameters:

activeOnly	When set to **true**, filters out expired features and features that have not yet started. Possible values are **true** and **false**. See also Viewing Only Active Features.
includeSlices	When set to **true**, returns information about the slices that belong to a feature. Possible values are **true** and **false**.
inGracePeriod	When set to **true**, shows features currently in a grace period before they expire. Possible values are **true** and **false**. See also Viewing Features in an Expiration Grace Period.
name	Specifies the name of the feature to return. Must be a string.
page	Retrieves a specific page. Must be a number. See also Managing Large Amounts of Returned JSON Data.
size	Specifies how many records per page should be returned. Must be a number. See also Managing Large Amounts of Returned JSON Data

Sample Response


```
[ {
  "id" : 1,
  "issued" : "2018-07-06",
  "meteredUndoInterval" : 0,
  "type" : "CONCURRENT",
  "meteredReusable" : false,
  "featureId" : "OkAZjIUQ8wuD9BeLSZb8HQ",
  "vendor" : "demo",
  "borrowInterval" : 604800,
  "featureName" : "f4",
  "renewInterval" : 15,
  "used" : 0,
  "featureCount" : 10,
  "expiry" : "2018-08-31",
  "featureVersion" : "1.0",
  "featureKind" : "NORMAL_FEATURE",
  "overdraftCount" : 0,
  "receivedTime" : "2018-07-06T22:15:52.000Z",
  "starts" : "2018-07-05",
  "entitlementExpiration" : "2018-07-04",
  "metered" : false,
  "uncappedOverdraft" : false,
  "reserved" : 0,
  "concurrent" : true,
  "uncounted" : false
}, {
  "id" : 2,
  "issued" : "2018-07-06",
  "meteredUndoInterval" : 0,
  "type" : "CONCURRENT",
  "meteredReusable" : false,
  "featureId" : "z3vg4xKuAy0OXrPb+9v2ow",
  "vendor" : "demo",
  "borrowInterval" : 604800,
  "featureName" : "f2",
  "renewInterval" : 15,
  "used" : 20,
  "featureCount" : 100,
  "expiry" : "permanent",
  "featureVersion" : "2.0",
  "featureKind" : "NORMAL_FEATURE",
  "overdraftCount" : 0,
  "receivedTime" : "2018-07-06T22:15:52.000Z",
  "starts" : "2018-07-05",
  "metered" : false,
  "uncappedOverdraft" : false,
  "reserved" : 0,
  "concurrent" : true,
  "uncounted" : false
}...]
```

JSON Response Details

The following table describes the data types of fields that might be returned in the JSON response.

"id"	Number
"type"	String
"featureName"	String
"featureVersion"	String
"expiry"	String (could be either the text 'permanent' or date in 'yyyy-mm-dd' format)
"featureCount"	String/number (string if uncounted feature is present, else number)
"overdraftCount"	String/number (string if uncapped feature is present, else number)
"used"	Number
"vendorString"	String
"notice"	String
"featureId"	String
"starts"	String (could be either the text 'permanent' or date in 'yyyy-mm-dd' format)
"featureKind"	String
"vendor"	String
"meteredUndoInterval"	Number
"meteredReusable"	Boolean
"receivedTime"	Date and time in ISO format
"selectorsDictionary"	JSON of <String,Object>
"uncounted"	Boolean
"metered"	Boolean
"uncappedOverdraft"	Boolean
"concurrent"	Boolean
"reserved"	Number
"slices"	List
"id"	Number
"slice"	String/number (string if uncapped feature is present, else number)
"used"	Number
"orphaned"	Boolean
"uncounted"	Boolean
"computedCount"	String/number (string if uncapped feature is present, else number)
"requested"	String
"status"	String

Viewing A Summary Of Features

The `/features/summaries` API returns a summary of available features, grouped by feature name. The response does not include expired features, features that have not yet started, and orphaned features.

URI	`/features/summaries`

Method	GET
Query parameters	N/A
Request body	N/A
Error codes	glsErr.serverNotFound glsErr.userAccessDenied

Sample Response

In the response, each section for a feature of a particular version includes a "links" element. This references the /feature/{id} API for the relevant feature ID.


```
{
    "f1" : {
        "1.0" : {
            "links" : [ {
                "rel" : "item",
                "href" : "http://localhost/api/1.0/instances/FWZUSJEZ0YW6/features/3"
            } ],
            "totalCount" : 100,
            "totalOverdraftCount" : 0,
            "totalUsed" : 0,
            "hasUncappedOverdraft" : false,
            "totalLineItems" : 1,
            "firstExpiryDate" : "permanent",
            "lastExpiryDate" : "permanent",
            "isMeteredReusable" : false,
            "hasOverdraft" : false,
            "totalAvailable" : 100,
            "isUncounted" : false,
            "isMetered" : false,
        },
        "2.0" : {
            "links" : [ {
                "rel" : "item",
                "href" : "http://localhost/api/1.0/instances/FWZUSJEZ0YW6/features/4"
            } ],
            "totalCount" : 100,
            "totalOverdraftCount" : 0,
            "totalUsed" : 0,
            "hasUncappedOverdraft" : false,
            "totalLineItems" : 1,
            "firstExpiryDate" : "permanent",
            "lastExpiryDate" : "permanent",
            "isMeteredReusable" : false,
            "hasOverdraft" : false,
            "totalAvailable" : 100,
            "isUncounted" : false,
            "isMetered" : false,
        }
    }
}
```

Retrieving Feature Data as NDJSON Objects

The /features.stream API asynchronously returns feature data as individual newline-delimited JSON (NDJSON) objects. The content type is "application/x-ndjson", not "application/json". For information about the NDJSON standard, see https://github.com/ndjson/ndjson-spec.

Note that the /features.stream API is available only for the local license server.

The /features.stream API returns all features (active and inactive).

URI	/features.stream
Method	GET
Query parameters	sort, direction, limit
Request body	N/A
Error codes	glsErr.userAccessDenied

It is recommended that you use the /features.stream API instead of the /features API if you need to query large tables. The Appendix F, Consuming Streaming Content, contains example code which shows how to deserialise the content to a representation which enables the client to consume the streamed content one line at a time.

Query Parameters

The following are optional parameters:

sort	Sorts the returned data by feature id and feature name. Possible values are **id** and **featureName**. By default, data is sorted by feature name.
direction	Sorts the returned data in ascending or descending order. Possible values are **asc** and **desc**.
limit	Specifies the maximum number of feature records to be retrieved. Must be a number.

Sample Response

For a successful GET method, all features served by the license server instance are returned as individual newline-delimited JSON objects.

```
{
    "id": 32,
    "type": "CONCURRENT",
    "featureName": "f9",
    "featureVersion": "1.0",
    "expiry": "permanent",
    "featureCount": "uncounted",
    "overdraftCount": 0,
    "used": 0,
```

```
    "vendorString": "%%ROLE:ANY,REGION:EMEA%%",
    "notice": "integration-helper",
    "featureId": "2ac4df45-0cec-4ff1-abe1-99307778ca0d",
    "starts": "2021-02-24",
    "featureKind": "NORMAL_FEATURE",
    "vendor": "fnedemo",
    "meteredUndoInterval": 0,
    "meteredReusable": false,
    "receivedTime": "2021-02-25T09:41:41.000Z",
    "selectorsDictionary": {
        "ROLE": "ANY",
        "REGION": "EMEA"
    },
    "uncappedOverdraft": false,
    "metered": false,
    "uncounted": true,
    "reserved": 0,
    "concurrent": true
}
{
    "id": 100,
    "type": "CONCURRENT",
    "featureName": "f9",
    "featureVersion": "2.0",
    "expiry": "permanent",
    "featureCount": "uncounted",
    "overdraftCount": 0,
    "used": 0,
    "vendorString": "%%ROLE:ANY,REGION:EMEA%%",
    "notice": "integration-helper",
    "featureId": "e40beacb-cca4-461d-98a5-7c5bd360853f",
    "starts": "2021-02-25",
    "featureKind": "NORMAL_FEATURE",
    "vendor": "fnedemo",
    "meteredUndoInterval": 0,
    "meteredReusable": false,
    "receivedTime": "2021-02-26T11:31:25.000Z",
    "selectorsDictionary": {
        "ROLE": "ANY",
        "REGION": "EMEA"
    },
    "uncappedOverdraft": false,
    "metered": false,
    "uncounted": true,
    "reserved": 0,
    "concurrent": true
}
{
    "id": 92,
    "type": "CONCURRENT",
    "featureName": "f8",
    "featureVersion": "2.0",
    "expiry": "permanent",
    "featureCount": "uncounted",
    "overdraftCount": 0,
    "used": 0,
```

```
        "vendorString": "%%ROLE:ANY,REGION:EMEA%%",
        "notice": "integration-helper",
        "featureId": "1bc8e03c-0cb9-4106-b55a-4c925253c7eb",
        "starts": "2021-02-25",
        "featureKind": "NORMAL_FEATURE",
        "vendor": "fnedemo",
        "meteredUndoInterval": 0,
        "meteredReusable": false,
        "receivedTime": "2021-02-26T11:31:25.000Z",
        "selectorsDictionary": {
            "ROLE": "ANY",
            "REGION": "EMEA"
        },
        "uncappedOverdraft": false,
        "metered": false,
        "uncounted": true,
        "reserved": 0,
        "concurrent": true
    }
```

Viewing Only Active Features

When calling the `/features` API with the `activeOnly` query parameter set to **`true`** (as in **`features?activeOnly=true`**), only active features are returned. A feature is considered active if it has started (its start date is in the past, or no start date is present) and has not reached its expiration date as defined in the back office (the `expiry` date in the API output).

By default, the `/features` API returns all features (**`features?activeOnly=false`**).

Viewing Features in an Expiration Grace Period

The `/features` API used with the `inGracePeriod` query parameter set to **`true`** (as in **`features?inGracePeriod=true`**) retrieves a list of features currently in an expiration grace period. A feature is considered to be in an expiration grace period when the current date is later than the `entitlementExpiration` date but earlier than the `expiry` date, as shown in the API output.

The following explains the two types of expiration dates in the output:

- **"expiry" date**—The final expiration 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 (see the next bullet) *plus the grace period* defined in the back office. If no grace period is defined, the `expiry` date is the same as the `entitlementExpiration` date.
- **"entitlementExpiration" 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 `entitlementExpiration` date is usually the original expiration date without the grace period applied (and is therefore earlier than the `expiry` date). If no grace period is defined, the `entitlementExpiration` and `expiry` dates are the same.

Providing a license-server administrator tool that lists features currently in a grace period enables administrators to initiate prompt renewals.

Note ▪ *Support for feature grace periods was introduced in FlexNet Embedded 2017 R2. Features defined in the back office before this 2017 R2 release do not support grace periods or the inclusion of the "entitlementExpiration" dates in capability responses.*

The following shows two features in a grace period. Note that, for each feature, the current date (the `issued` date) is later than the `entitlementExpiration` date but earlier than the `expiry` date:


```
[ {
  "id" : 1,
  "issued" : "2018-07-06",
  "meteredUndoInterval" : 0,
  "type" : "CONCURRENT",
  "meteredReusable" : false,
  "featureId" : "OkAZjIUQ8wuD9BeLSZb8HQ",
  "vendor" : "demo",
  "borrowInterval" : 604800,
  "featureName" : "f4",
  "renewInterval" : 15,
  "used" : 0,
  "featureCount" : 10,
  "expiry" : "2018-08-31",
  "featureVersion" : "1.0",
  "featureKind" : "NORMAL_FEATURE",
  "overdraftCount" : 0,
  "receivedTime" : "2018-07-06T22:15:52.000Z",
  "starts" : "2018-07-05",
  "entitlementExpiration" : "2018-07-04",
  "metered" : false,
  "uncappedOverdraft" : false,
  "reserved" : 0,
  "concurrent" : true,
  "uncounted" : false
}, {
  "id" : 4,
  "issued" : "2018-07-06",
  "meteredUndoInterval" : 0,
  "type" : "CONCURRENT",
  "meteredReusable" : false,
  "featureId" : "2QwgJ/T+E6aF/jFClRC9jw",
  "vendor" : "demo",
  "borrowInterval" : 604800,
  "featureName" : "f1",
  "renewInterval" : 15,
  "used" : 0,
  "featureCount" : 10,
  "expiry" : "2018-08-31",
  "featureVersion" : "1.0",
  "featureKind" : "NORMAL_FEATURE",
  "overdraftCount" : 0,
  "receivedTime" : "2018-07-06T22:15:52.000Z",
  "starts" : "2018-07-05",
  "entitlementExpiration" : "2018-07-01",
  "metered" : false,
  "uncappedOverdraft" : false,
```

```
    "reserved" : 0,
    "concurrent" : true,
    "uncounted" : false
  } ]
```

Viewing Number of Features

The /features API invoked as the HEAD request returns the number of features associated with the instance.

URI	/features
Method	HEAD
Query parameters	N/A
Request body	N/A
Error codes	N/A

The response only contains a response header (with a string format of Long), but no response body. The response header might look similar to the output of the following cURL command:

```
$ curl --head http://localhost:7070/gls/api/1.0/instances/B9W5Q808M6BZ/features
```

Output

```
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0HTTP/1.1 200
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
Total-Elements: 3
X-SERVER-UTILIZATION: 1
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Frame-Options: DENY
Content-Length: 0
Date: Fri, 15 Jan 2021 12:12:51 GMT
```

Viewing Specified Feature

The /feature/{id} API returns information about a specific feature. The path parameter {id} must be of data type number.

URI	/features/{id}

Method	GET
Query parameters	N/A
Request body	N/A
Error codes	glsErr.serverNotFound glsErr.featureNotFound

Sample Response


```
{
  "id" : 2,
  "issued" : "2018-07-06",
  "meteredUndoInterval" : 0,
  "type" : "CONCURRENT",
  "meteredReusable" : false,
  "featureId" : "z3vg4xKuAy0OXrPb+9v2ow",
  "vendor" : "demo",
  "borrowInterval" : 604800,
  "featureName" : "f2",
  "renewInterval" : 15,
  "used" : 20,
  "featureCount" : 100,
  "expiry" : "permanent",
  "featureVersion" : "2.0",
  "featureKind" : "NORMAL_FEATURE",
  "overdraftCount" : 0,
  "receivedTime" : "2018-07-06T22:15:52.000Z",
  "starts" : "2018-07-05",
  "metered" : false,
  "uncappedOverdraft" : false,
  "reserved" : 0,
  "concurrent" : true,
  "uncounted" : false
}
```


JSON Response Details

The following table describes the data types of fields that might be returned in the JSON response.

"id"	Number
"type"	String
"featureName"	String
"featureVersion"	String
"expiry"	String (could be either the text 'permanent' or date in 'yyyy-mm-dd' format)
"featureCount"	String/number (string if uncounted feature is present, else number)
"overdraftCount"	String/number (string if uncapped feature is present, else number)
"used"	Number
"vendorString"	String
"notice"	String
"featureId"	String

`"starts"`	String (could be either the text 'permanent' or date in 'yyyy-mm-dd' format)
`"featureKind"`	String
`"vendor"`	String
`"meteredUndoInterval"`	Number
`"meteredReusable`	Boolean
`"receivedTime"`	Date and time in ISO format
`"selectorsDictionary"`	JSON of <String,Object>
`"uncounted"`	Boolean
`"metered"`	Boolean
`"uncappedOverdraft"`	Boolean
`"concurrent"`	Boolean
`"reserved"`	Number
`"slices"`	List (
`"id"`	Number
`"slice"`	String/number (string if uncapped feature is present, else number)
`"used"`	Number
`"orphaned"`	Boolean
`"uncounted"`	Boolean
`"computedCount"`	String/number (string if uncapped feature is present, else number)
`"requested"`	String
`"status"`	String

Viewing Feature Usage

The `/features/{id}/clients` API returns usage information for a specific feature along with client details. The path parameter `{id}` must be of data type `number`.

URI	`/features/{id}/clients`
Method	`GET`
Query parameters	`includeAll, page, size, includeUsageExpiry` For more information, see *Query Parameters*, below.
Request body	N/A

Error codes	`glsErr.serverNotFound`
	`glsErr.featureNotFound`
	`glsErr.queryParamNeedsInt`
	`glsErr.queryParamTooSmall`
	`glsErr.queryParamTooLarge`
	`glsErr.queryParamUnsupported`

Query Parameters

The following are optional parameters:

includeAll	When set to **true**, also returns expired usages. Possible values are **true** and **false**.
page	Retrieves a specific page. Must be a number. See also Managing Large Amounts of Returned JSON Data.
size	Specifies how many records per page should be returned. Must be a number. See also Managing Large Amounts of Returned JSON Data.
includeUsageExpiry	When set to **true**, the response includes the usageExpiry field for each feature, which shows the date and time when the feature expires on the client. Possible values are **true** and **false**.

Sample Response


```
[ {
  "id" : 2,
  "usageKind" : "CONCURRENT_USAGE",
  "useCount" : 20,
  "client" : {
    "id" : 1,
    "machineType" : "UNKNOWN",
    "requestOperation" : "REQUEST",
    "servedStatus" : "NORMAL",
    "correlationId" : "880bbb5b-d060-46bf-b321-7b5dcf243b5b",
    "updateTime" : "2018-07-06T22:31:18.700Z",
    "collectedHostIds" : "STRING 1234567890",
    "trusted" : false,
    "serverHostid" : {
      "hostidValue" : "7C7A91BA832B",
      "hostidType" : "ETHERNET"
    },
    "hostid" : {
      "hostidValue" : "1234567890",
      "hostidType" : "STRING"
    },
    "requestHostid" : {
      "hostidValue" : "1234567890",
      "hostidType" : "STRING"
    },
    "expiry" : "2018-07-13T22:31:18.700Z"
  },
  "reservedCount" : 0
} ]
```

JSON Response Details

The following table describes the data types of fields that might be returned in the JSON response.

"id"	Number
"usageKind"	String
"client"	JSON object
"id"	Number
"hostid"	JSON object
"hostidValue"	String
"hostidType"	String
"updateTime"	Date and time in ISO format
"servedStatus"	String
"machineType"	String
"trusted"	Boolean
"correlationId"	String
"collectedHostIds"	String
"requestOperation"	String
"requestHostid"	JSON object
"hostidValue"	String
"hostidType"	String
"serverHostid"	JSON Object
"hostidValue"	String
"hostidType"	String
"expiry"	String (could be either the text 'permanent' or date in 'yyyy-MM-dd'T'HH:mm:ss.SSS'Z'' format
"useCount"	Number
"reservedCount"	Number

Listing Overages

The `/overages` API returns the first page of overages, including the overage count and expiration together with the identifiers of the device holding the feature and the feature itself. Order is by expiry descending, device ID ascending.

URI	`/overages`
Method	`GET`
Query parameters	`page, size` Use `page` to retrieve a specific page. Must be a number. Use `size` to specify how many records per page should be returned. Must be a number. See also Managing Large Amounts of Returned JSON Data.

Request body	N/A
Error codes	glsErr.serverNotFound glsErr.queryParamNeedsInt glsErr.queryParamTooSmall glsErr.queryParamTooLarge glsErr.queryParamUnsupported

Sample Response

```
[ {
  "feature" : {
    "id" : 1,
    "type" : "CONCURRENT",
    "featureName" : "f1",
    "featureVersion" : "1.0",
    "expiry" : "2020-12-31",
    "featureCount" : 5,
    "overdraftCount" : 0,
    "overdraftUsedCount" : 0,
    "used" : 10,
    "sharedUsed" : 10,
    "assignedReserved" : 0,
    "unassignedReserved" : 0,
    "vendorString" : "vendor defined",
    "issued" : "2018-07-05",
    "borrowInterval" : 604800,
    "renewInterval" : 15,
    "featureId" : "li1-0-6",
    "starts" : "2018-07-04",
    "featureKind" : "NORMAL_FEATURE",
    "vendor" : "demo",
    "meteredUndoInterval" : 0,
    "meteredReusable" : false,
    "receivedTime" : "2018-07-05T21:32:18.000Z",
    "metered" : false,
    "uncounted" : false,
    "reserved" : 0,
    "concurrent" : true,
    "uncappedOverdraft" : false
  },
  "overageCount" : 5
} ]
```

APIs to Manage Named License Pools

The Named License Pools functionality available with the FlexNet Embedded license server (the local license server or a CLS instance) enables the license server administrator to allocate licenses to a group of client devices or users to help ensure that these entities have access to the features they need. For more information about named license pools, see Allocating Licenses Using Named License Pools.

Important* ▪ *The license server administrator can use either reservations or named license pools. Reservations and named license pools cannot coexist alongside each other. If you were previously using reservation, you must delete all reservation groups before you can use named license pools. For procedural information, see Managing Reservation Groups. For general information about reservations, see License Reservations.

Note* ▪ *In previous releases of the FlexNet Embedded license server, named license pools used to be referred to as "partitions". However, the term "partition" still persists in some areas such as API endpoint names, output, and error messages.

The following sections describe these APIs:

- Uploading the Model Definition
- Viewing the Model Definition
- Deleting the Model Definition
- Viewing License Pools
- Viewing a Specified License Pool By ID

The last section, JSON Response Details, explains the most relevant lines in the JSON response that is returned when the `/partitions` endpoint is called using `GET`.

API Summary

The following APIs manage the model definition and license pools (named and default) for the license server.

Table 4-4 ▪ License Server REST APIs Used to Manage License Pools

URI	Method	Description
/rules	POST	Uploads the model definition to the license server, persisting the license pools and feature slices. A Content-Type of text/plain is expected.
	GET	Returns the model definition for that server instance.
	DELETE	Deletes the model definition. The default model definition is applied to the server instead.
/partitions	GET	Returns the license pools along with feature counts and feature slices. Also includes information about used and allocated counts for each feature slice.
/partitions/{id}	GET	Returns the specified named license pool with the ID specified in the URL along with feature counts and feature slices which belong to that named license pool. Also includes information about remaining counts for each feature slice.

Uploading the Model Definition

The /rules API used with the POST method uploads the model definition to the license server and saves it in the database (requires administrator authorization).

If the upload fails, the model definition is not stored in the database. Instead, the server uses the previous model definition.

Only one model definition can be active for each local license server or CLS instance.

You cannot upload a model definition with the name reservations or default, because these names are reserved.

Important ▪ *If you were previously using reservations, you must delete all reservation groups. Otherwise, uploading a model definition will fail. For more information, see Named License Pools vs. Reservations.*

The request body contains the model definition. For details on its grammar and syntax, see Model Definition Grammar for Named License Pools.

URI	/rules
Method	POST
Query parameters	N/A

Request body	Include the model definition in the request. Use the content-type header text/plain with character set UTF-8.
	`model "example" {`
	`    partitions {`
	`        partition "p1" {`
	`           "f1" 1.0 20;`
	`        }`
	`        partition "p2" {`
	`           "f1" 1.0 10;`
	`        }`
	`    }`
	`    on hostid("h1") {`
	`        use "p1"`
	`        continue`
	`    }`
	`    on hostid("h2") {`
	`        use "p2", "default"`
	`        accept`
	`    }`
	`    on any() {`
	`        use "default"`
	`        accept`
	`    }`
	`}`

Error codes	glsErr.invalidModelName glsErr.modelSyntaxError glsErr.restUnimplemented

Sample response

On success, no response body is provided.

For a failed POST method, the response contains the HTTP error code.

If the status code 400 is returned, this can indicate a syntax error in the script that was uploaded. The returned response body will describe the exception, as shown in the following example:

```
{
    "key" : "glsErr.modelSyntaxError",
    "message" : "Model syntax error: Error while parsing action
      'model/Partitions/Sequence/PartitionList' at input position (line 17, pos 5): \n on
      hostname(\n"cr100\") {\n ^\n\norg.parboiled.errors.ParsingException: Partition p3 already
      exists.",
    "arguments" : [
      "Error while parsing action 'model/Partitions/Sequence/PartitionList' at input position
      (line 17, pos 5): \n on hostname("cr100") {\n
      ^\n\norg.parboiled.errors.ParsingException: Partition p3 already exists"
    ]
}
```

Error Messages

The following table lists possible error messages and the required action:

Table 4-5 • License Server Errors

Exception	Definition
glsErr.invalidModelName	The model definition names default and reservations are reserved names. Change the name and re-upload the model definition.
glsErr.modelSyntaxError	The uploaded model definition contains a syntax error. Correct the model definition and re-upload it.
glsErr.restUnimplemented	The back-up license server does not support named license pools. Any PUT requests to /rules are rejected. During a failover period, feature counts will only be distributed from the default license pool.

Viewing the Model Definition

When the GET method is invoked with the /rules API, this endpoint returns the model definition that is currently applied to the license server instance.

URI	/rules
Method	GET
Query parameters	N/A
Request body	N/A

Error codes	N/A

Sample response

For a successful GET method, the current model definition is returned. The response will look similar to this:

```
model "example" {
    partitions {
        partition "engineering" {
            "f1" 1.0 5
        }
        partition "sales" {
            "f1" 1.0 5
        }
    }

    on dictionary("business-unit" : "engineering") {
        use "engineering"
        accept
    }

    on dictionary("business-unit" : "sales") {
        use "sales"
        accept
    }

    on any() {
        use "default"
        accept
    }
}
```

If no model definition has been posted, the default model is returned. The response will look similar to this:

```
model "default" {
    on any() {
        use "default"
        accept
    }
}
```

If the method fails, the response contains the HTTP status code and an empty response body.

Deleting the Model Definition

The /rules API used with the DELETE method deletes the model definition that is currently applied to the license server instance (requires administrator authorization). The default model definition is applied to the server.

URI	/rules
Method	DELETE
Query parameters	N/A

Request body	N/A
Error codes	N/A

For a successful `DELETE`, the status code 410 is returned, but no response body is provided.

For a failed `DELETE` method, the response contains the HTTP error code and an empty response body.

Viewing License Pools

When the `GET` method is invoked with the `/partitions` API, this endpoint returns all license pools (named and default) for the license server instance. For each license pool, the response lists information about every feature and every feature slice in the license pool (feature slices are portions of feature counts allocated to a named license pool).

URI	`/partitions`
Method	`GET`
Query parameters	name Shows the license pool named in the URL along with the features and feature counts that belong to that license pool. Must be a string. See also Viewing a Specified License Pool By Name.
Request body	N/A
Error codes	`glsErr.partitionNotFound` (only when using the `name` parameter)

Sample response

For a successful `GET` method, all license pools (named and default) for the license server instance are returned.

The following successful sample response has been shortened for documentation purposes. For information about elements in the response see JSON Response Details.


```
{
    "partitions": [
        {
            "id": 1,
            "name": "default",
            "lastModified": 1563445226020,
            "default": true,
            "activeFeatureSlices": [
                {
                    "id": 1,
                    "feature": {
                        "id": 1,
                        "type": "CONCURRENT",
                        "featureName": "f_con_1",
                        "featureVersion": "1.0",
```

```
                "expiry": "permanent",
                "featureCount": 100,
                "used": 96,
            },
            "slice": 97,
            "used": 95,
            "computedCount": 97,
            "requested": "97",
            "uncounted": false
            "status": "NORMAL",
        },
        {
            "id": 2,
            "feature": {
                "id": 2,
                "type": "CONCURRENT",
                "featureName": "f_con_2",
                "featureVersion": "1.0",
                "expiry": "permanent",
                "featureCount": 80,
                "used": 40,
            },
            "slice": 50,
            "used": 26,
            "computedCount": 50,
            "requested": "50",
            "uncounted": false
            "status": "NORMAL",
        },
    ]
},
{
    "id": 2,
    "name": "p1",
    "lastModified": 1564133869116,
    "default": false,
    "activeFeatureSlices": [
        {
            "id": 3,
            "feature": {
             "id": 1,
                "type": "CONCURRENT",
                "featureName": "f_con_1",
                "featureVersion": "1.0",
                "expiry": "permanent",
                "featureCount": 100,
                "used": 96,
            },
            "slice": 3,
            "used": 1,
            "computedCount": 3,
            "requested": "3",
            "uncounted": false
            "status": "NORMAL",
        }
    ]
```

```
        }
    ]
}
```

Viewing a Specified License Pool By Name

The `name` parameter (for example, `/~/partitions?name={licensePoolName}`) shows the license pool (named or default) specified in the URL along with the features and feature counts that belong to that license pool.

For a successful `GET` method, the license pool is returned.

If the method fails, the response contains the HTTP status code. In case of a status code 404, the response body indicates that the requested license pool does not exist. For example, you might see:

```
{

    "key": "glsErr.partitionNotFound",
    "message": "Partition not found: engineering-EMEA",
    "arguments": [ "engineering-EMEA" ]
}
```

The following successful sample response has been shortened for documentation purposes. For information about elements in the response see JSON Response Details.

Sample response

```
{
    "partitions": [
        {
            "id": 1,
            "name": "default",
            "lastModified": 1563445226020,
            "default": true,
            "activeFeatureSlices": [
                {
                    "id": 1,
                    "feature": {
                        "id": 1,
                        "type": "CONCURRENT",
                        "featureName": "f_con_1",
                        "featureVersion": "1.0",
                        "expiry": "permanent",
                        "featureCount": 100,
                        "used": 96,
                    },
                    "slice": 97,
                    "used": 95,
                    "computedCount": 97,
                    "requested": "97",
                    "uncounted": false
                    "status": "NORMAL",
                },
                {
                    "id": 2,
                    "feature": {
                        "id": 2,
                        "type": "CONCURRENT",
```

```
                        "featureName": "f_con_2",
                        "featureVersion": "1.0",
                        "expiry": "permanent",
                        "featureCount": 80,
                        "used": 40,
                    },
                    "slice": 50,
                    "used": 26,
                    "computedCount": 50,
                    "requested": "50",
                    "uncounted": false
                    "status": "NORMAL",
                },
            ]
        },
    ]
}
```

Viewing a Specified License Pool By ID

The `/partitions/{id}` API shows the license pool (named or default) with the ID specified in the URL along with the features and counts that belong to that license pool. The ID is assigned to the license pool by the license server after the model definition has been uploaded. The path parameter `{gid}` must be of data type `number`.

URI	/partitions/{id}
Method	GET
Query parameters	N/A
Request body	N/A
Error codes	glsErr.partitionNotFound

Sample response

For a successful `GET` method, the license pool with the ID specified in the URL is returned. The response is identical with the output that is returned when using the `name` parameter (for example, **`/~/partitions?name={licensePoolName}`**). For sample output, see Viewing a Specified License Pool By Name.

If the method fails, the response contains the HTTP status code. In case of a status code 404, the response body indicates that the requested license pool does not exist. For example, you might see:

```
{
    "key": "glsErr.partitionNotFound",
    "message": "Partition not found: 1234",
    "arguments": [ "1234" ]
}
```

JSON Response Details

The following section explains the most relevant lines in the JSON response that is output when you call the `/partitions` endpoint using `GET`.

Table 4-6 ▪ Elements in a JSON Response

Parent Element	Child Element	Description
`partitions`	`"id": n`	The ID is assigned to the license pool (named or default) by the license server after the model definition has been uploaded.
	`"name": "name"`	The license pool name. The names `default` and `reservations` are reserved and cannot be used.
	`"lastModified": yyyy-MM-dd'T'HH:mm:ss.SSS'Z'`	Indicates when the license pool was last changed (for example, the allocated feature count or rules for that license pool have changed). The date is specified using the ISO 8601 date/time notation, expressed in UTC (for example, `2019-10-22T11:16:31.000Z`).
	`"default": true|false`	Indicates whether the license pool is the default license pool or not. The default license pool cannot be deleted.
`activeFeatureSlices`	`"id": n`	Each feature slice is assigned an ID by the license server after the model definition has been uploaded.
	`"slice": n`	Total license count for the feature that has been allocated to the license pool. Typically, this would be the count specified in the model definition, assuming that sufficient counts are available on the license server. If fewer counts are available on the license server than are specified in the model definition, or if existing usage prevents allocation into the license pool after a new model definition has been uploaded, "`slice`" specifies the actual number of allocated licenses.
	`"used": n`	The number of feature counts that have been served from this slice.

Table 4-6 ▪ Elements in a JSON Response

<table>
<tr><th>Parent Element</th><th>Child Element</th><th>Description</th></tr>
<tr><td><code>activeFeatureSlices</code> (continued)</td><td><code>"computedCount": n</code></td><td>The number of feature counts the slice should have, based on the model definition and the number of feature counts the server has been provisioned with.</td></tr>
<tr><td></td><td><code>"requested": n</code></td><td>The license count for the feature that has been specified in the model definition, irrespective of whether sufficient counts are available on the license server.

The following sample code from a model definition requests 5 counts of feature <code>f1</code> for the named license pool called <code>engineering</code>:
<pre>model "example1" {
    partitions {
      partition "engineering" {
        feature "f1" 1.0 5
      }
    }
}</pre></td></tr>
<tr><td></td><td><code>"status": "value"</code></td><td>The status of the feature slice indicates how the slices of a server are used.

Possible values:
<code>OVER_ALLOCATED</code>—The slice has more feature counts than it should have, based on the model definition and the number of counts the server has been provisioned with ("slice" > "computedCount").
<code>NORMAL</code>—The slice has the number of feature counts that it should have, based on the model definition and the number of counts the server has been provisioned with ("slice" = "computedCount").
<code>UNDER_ALLOCATED</code>—The slice has fewer feature counts than it should have, based on the model definition and the number of counts the server has been provisioned with ("slice" < "computedCount").</td></tr>
</table>

APIs to Manage Reservations

License reservations are specific feature counts on the specified license server that are pre-allocated to specific client devices or users. For more information about reservations and the reservation hierarchy, see License Reservations in Chapter 5, More About Basic License Server Functionality.

Important ▪ *A license server administrator can use either reservations or named license pools. Reservations and named license pools cannot coexist alongside each other. If you were previously using named license pools, you must delete the model definition before you can create reservations. For procedural information, see Deleting the Model Definition. For general information about named license pools, see Allocating Licenses Using Named License Pools.*

The following sections describe the APIs used to manage reservations:

- API Summary
- Managing Reservation Groups
- Managing a Specific Reservation Group
- Managing All Reservations in a Specific Reservation Group
- Managing a Specific Reservation in a Reservation Group
- Managing a Specific Reservation Entry

API Summary

The following lists the APIs that manage reservations:

Table 4-7 ▪ License Server REST APIs Used to Manage License Reservations

URI	Method	Description
/reservationgroups	`GET, POST, DELETE`	Returns a list of the currently existing reservation groups, creates a new reservation group, or deletes all existing reservation groups.
/reservationgroups/{gid}	`GET, POST, DELETE`	Returns the content of or deletes a given reservation group (*gid*); or, within the given reservation group, creates one or more reservations (each with a hostid identifying a specific client device or user), and adds reservation entries to each new reservation.
/reservationgroups/{gid}/reservations	`GET`	Returns information about existing reservations in the specified reservation group.
/reservationgroups/{gid}/reservations/{rid}	`GET, DELETE`	Returns the contents of the specified reservation (*rid*), or deletes the entire reservation.
/reservationgroups/{gid}/reservations/{rid}/entries	`POST`	Creates new reservation entries in the reservation (*rid*).
/reservationgroups/{gid}/reservations/{rid}/entries/{eid}	`GET, DELETE`	Returns data about a given reservation entry (*eid*) or deletes a given reservation entry.

Table 4-7 ▪ License Server REST APIs Used to Manage License Reservations (cont.)

URI	Method	Description
/reservationgroups/{gid}/reservations/{rid}/entries/{eid}/state	PUT	Changes the state of a given reservation entry (*eid*) to ENABLED or DISABLED.

Managing Reservation Groups

***Important ▪** If you were previously using named license pools, you must delete the model definition before you can manage reservations. See Deleting the Model Definition.*

When the GET method is invoked, this endpoint returns information about all existing reservation groups:

URI	/reservationgroups
Method	GET
Query parameters	name, page, size See section *Query Parameters*, below.
Request body	N/A
Error codes	glsErr.serverNotFound glsErr.queryParamNeedsInt glsErr.queryParamTooSmall glsErr.queryParamTooLarge glsErr.queryParamUnsupported glsErr.reservationNotFound

This endpoint, invoked with the POST method, adds a new reservation group. In the request body, the definition for the reservation group to be added is in JSON format. The name attribute is optional.

URI	/reservationgroups
Method	POST
Query parameters	N/A

Request body	`{ "name" :  "group1", "reservations" :`
	`[`
	`{ "hostId": { "value" : "dev1", "type" : "STRING" },`
	`"reservationEntries": [`
	`{ "featureName": "f1", "featureVersion": "1.0",`
	`"featureCount": 1 },`
	`{ "featureName": "f2", "featureVersion": "1.0",`
	`"featureCount": 2 } ]},`
	`{ "hostId": { "value" : "dev2", "type" : "STRING" },`
	`"reservationEntries": [`
	`{ "featureName": "f3", "featureVersion": "1.0",`
	`"featureCount": 3 }]}`
	`]}`

Error codes	glsErr.serverNotFound glsErr.optimisticLocking glsErr.reservationAlreadyExists glsErr.invalidReservationRequest glsErr.serviceBusyDueToReservations

When the DELETE method is invoked, the endpoint deletes all existing reservation groups:

URI	/reservationgroups
Method	DELETE
Query parameters	N/A
Request body	N/A
Error codes	glsErr.serverNotFound glsErr.optimisticLocking glsErr.serviceBusyDueToReservations glsErr.assigningReservations

Query Parameters

The following are optional parameters for use in GET requests:

name	Specifies the name of the reservation group to return. Must be a string (see example in the *Sample Responses* section, below). Cannot be used in combination with the page and size parameters.
page	Retrieves a specific page. Must be a number. Cannot be used in combination with the name parameter. See also Managing Large Amounts of Returned JSON Data.

size	Specifies how many records per page should be returned. Must be a number. Cannot be used in combination with the name parameter. See also Managing Large Amounts of Returned JSON Data.

No query parameters are available for POST or DELETE requests.

Sample Responses

For the GET method without any query parameters, the response contains information about the existing reservations groups, named or not:

```
[ {
  "name" : "group1",
  "id" : 3,
  "creationDate" : "2019-02-11T03:43:22.969Z",
  "ipAddress" : "0:0:0:0:0:0:0:1"
}, {
  "name" : "group2",
  "id" : 4,
  "creationDate" : "2019-02-11T03:44:02.181Z",
  "ipAddress" : "0:0:0:0:0:0:0:1"
} ]
```

For the GET method with name=*groupname* parameter, the response contains the information about the named reservation group. For example, name=group1, generates a response similar to this:

```
[ {
  "name" : "group1",
  "id" : 1,
  "ipAddress" : "0:0:0:0:0:0:0:1",
  "creationDate" : "2019-03-07T03:10:22.667Z"
} ]
```

For a successful POST method, the response contains a confirmation of success where *gid* is an id assigned to the newly created reservation group:

```
{
uri: "/api/1.0/instances/{instanceid}/reservationgroups/{gid}"
status: "SUCCESS"
}
```

For a failed POST method, the response contains the error code and its corresponding message and a list of error arguments that varies based on the error. For example, you might see:

```
{
key: "glsErr.reservationAlreadyExists"
message: "Reservation already exists: group2"
arguments: [1] 0:  "group2"
}
```

No response is returned for the DELETE method.

Managing a Specific Reservation Group

When the GET method is invoked, this endpoint returns information about the reservation group identified by its group ID (*gid*). The path parameter {gid} must be of data type number.

URI	/reservationgroups/{gid}
Method	GET
Query parameters	summary Returns a summary of the enabled reservation entries found in the specified reservation group (see example in the *Sample Responses* section, below). Possible values are true and false.
Request body	N/A
Error codes	glsErr.serverNotFound glsErr.reservationNotFound

When the POST method invoked, this endpoint adds new reservation (for a specific client device or user) to the specified reservation group. The definition for the reservation to be added must be in JSON format:

URI	/reservationgroups/{gid}
Method	POST
Query parameters	N/A
Request body	{ "hostId": { "value" : "dev3", "type" : "STRING" }, "reservationEntries": [{ "featureName": "f1", "featureVersion": "1.0", "featureCount": 1 }]}
Error codes	glsErr.serverNotFound glsErr.optimisticLocking glsErr.reservationNotFound glsErr.invalidReservationRequest glsErr.serviceBusyDueToReservations

When the DELETE method is invoked, the endpoint deletes the entire specified reservation group:

URI	/reservationgroups/{gid}
Method	DELETE
Query parameters	N/A
Request body	N/A

Error codes	glsErr.serverNotFound glsErr.optimisticLocking glsErr.reservationNotFound glsErr.serviceBusyDueToReservations glsErr.assigningReservations

Sample Responses

For the GET method without any query parameters, the response shows information about the specified reservation group:

```
[ {
  "name" : "group1",
  "id" : 1,
  "ipAddress" : "0:0:0:0:0:0:0:1",
  "creationDate" : "2019-03-07T03:10:22.667Z"
} ]
```

For the GET method with the summary=true query parameter, the response contains a summary of the enabled reservation entries found in the specified reservation group:

```
[ {
  "featureName" : "f1",
  "featureVersion" : "1.0",
  "featureCount" : 2
}, {
  "featureName" : "f1",
  "featureVersion" : "2.0",
  "featureCount" : 1
} ]
```

For a successful POST method, the response contains a confirmation of success where *rid* is the reservation ID assigned to the newly created reservation:

```
{
uri: "/api/1.0/instances/{instanceid}/reservations/{rid}"
status: "SUCCESS"
}
```

For a failed POST method, the response contains the error code and its corresponding message and a list of error arguments that varies based on the error. For example, you might see:

```
{
key: "glsErr.invalidReservationRequest"
message: "Invalid reservation request"
arguments: [1] 0: "host ID STRING:dev4 is already used in a different reservation group"
}
```

No response is returned for the DELETE method.

Managing All Reservations in a Specific Reservation Group

When the GET method is invoked, this endpoint returns information about all reservations defined in the specified reservation group. The path parameter {gid} must be of data type number:

URI	/reservationgroups/{gid}/reservations
Method	GET
Query parameters	disabled, hostid/hostidtype, page, size For more information, see *Query Parameters*, below.
Request body	N/A
Error codes	glsErr.serverNotFound glsErr.hostidNotFound glsErr.queryParamNeedsInt glsErr.queryParamTooSmall glsErr.queryParamTooLarge glsErr.queryParamUnsupported glsErr.HostIdTypeWithoutHostId glsErr.reservationNotFound glsErr.reservationNotFound glsErr.queryParamIncompatible

Query Parameters

The following are optional parameters:

disabled	Returns only those reservations in the specified group that contain at least one reservation entry with a DISABLED status. Possible values are **true** and **false**. Cannot be used in combination with the hostid/hostidtype parameter. For more information, see the *Sample Responses* section.
hostid/hostidtype	The hostidtype parameter is used in conjunction with the "hostid" parameter and is optional (hostidtype defaults to the value "STRING"). Returns information about the reservation for the given hostid (if found in the specified reservation group). Cannot be used in combination with the disabled, page and size parameters. Values must be of data type string. For more information, see the *Sample Responses* section.
page	Retrieves a specific page. Must be a number. Cannot be used in combination with the hostid/hostidtype parameter. See also Managing Large Amounts of Returned JSON Data.

size	Specifies how many records per page should be returned. Must be a number. Cannot be used in combination with the hostid/hostidtype parameter. See also Managing Large Amounts of Returned JSON Data.

Sample Responses

For the GET method without any query parameters, the response shows information about all the reservations defined for the given reservation group:


```
[ {
"id" : 3,
"lastModified" : "2019-02-11T05:39:23.664Z",
"ipAddress" : "0:0:0:0:0:0:0:1",
"reservationEntries" : [ {
"id" : 5,
"state" : "DISABLED",
"featureName" : "f1",
"featureVersion" : "1.0",
"featureCount" : 1
}, {
"id" : 3,
"state" : "ENABLED",
"featureName" : "f1",
"featureVersion" : "2.0",
"featureCount" : 1
} ],
"hostId" : {
"type" : "STRING",
"value" : "dev1"
}
}, {
"id" : 4,
"lastModified" : "2019-02-11T05:39:23.664Z",
"ipAddress" : "0:0:0:0:0:0:0:1",
"reservationEntries" : [ {
"id" : 4,
"state" : "ENABLED",
"featureName" : "f1",
"featureVersion" : "1.0",
"featureCount" : 1
} ],
"hostId" : {
"type" : "STRING",
"value" : "dev2"
}
}, {
"id" : 5,
"lastModified" : "2019-02-11T05:50:01.052Z",
"ipAddress" : "0:0:0:0:0:0:0:1",
"reservationEntries" : [ {
"id" : 6,
"state" : "ENABLED",
"featureName" : "f1",
"featureVersion" : "1.0",
```

```
"featureCount" : 1
} ],
"hostId" : {
"type" : "STRING",
"value" : "dev3"
}
} ]
```

For the GET method with the disabled=true query parameter, the response shows only those reservations in the specified group that contain at least one reservation entry with a DISABLED status:

```
[ {
"id" : 3,
"lastModified" : "2019-02-11T05:39:23.664Z",
"ipAddress" : "0:0:0:0:0:0:0:1",
"reservationEntries" : [ {
"id" : 5,
"state" : "DISABLED",
"featureName" : "f1",
"featureVersion" : "1.0",
"featureCount" : 1
}, {
"id" : 3,
"state" : "ENABLED",
"featureName" : "f1",
"featureVersion" : "2.0",
"featureCount" : 1
} ],
"hostId" : {
"type" : "STRING",
"value" : "dev1"
}
} ]
```

For the GET method with hostid=*hostid_value*&hostidtype=*hostidtype_value* query parameters, the response shows information about the reservation for the given hostid (if found in the specified reservation group). The hostidtype parameter is optional and defaults to STRING. For example, hostid=dev2 produces a response similar to this:

```
[ {
"id" : 4,
"lastModified" : "2019-02-11T05:39:23.664Z",
"ipAddress" : "0:0:0:0:0:0:0:1",
"reservationEntries" : [ {
"id" : 4,
"state" : "ENABLED",
"featureName" : "f1",
"featureVersion" : "1.0",
"featureCount" : 1
} ],
"hostId" : {
"type" : "STRING",
"value" : "dev2"
}
} ]
```

Managing a Specific Reservation in a Reservation Group

When the `GET` method is invoked, this endpoint returns all existing reservation entries for a specific reservation, identified by a reservation ID (*rid*), within the reservation group (*gid*). A *reservation* contains all the reservation entries assigned to a given client device or user, as identified by its hostid in the reservation definition. The path parameters `{gid}` and `{rid}` must be of data type `number`.

URI	`/reservationgroups/{gid}/reservations/{rid}`
Method	`GET`
Query parameters	N/A
Request body	N/A
Error codes	`glsErr.serverNotFound` `glsErr.reservationNotFound`

When the `DELETE` method is invoked, the endpoint deletes the entire reservation (*rid*) within the reservation group (*gid*):

URI	`/reservationgroups/{gid}/reservations/{rid}`
Method	`DELETE`
Query parameters	N/A
Request body	N/A

Error codes	`glsErr.serverNotFound`
	`glsErr.optimisticLocking`
	`glsErr.reservationNotFound`
	`glsErr.serviceBusyDueToReservations`
	`glsErr.assigningReservations`

When the `POST` method invoked, this endpoint adds new reservation entries to the specified reservation (*rid*) within the reservation group (*gid*):

URI	`/reservationgroups/{gid}/reservations/{rid}/entries`
Method	`POST`
Query parameters	N/A

Request body	`[`
	`{ "featureName": "f1", "featureVersion": "1.0", "featureCount": 1 },`
	`{ "featureName": "f2", "featureVersion": "1.0", "featureCount": 2 },`
	`{ "featureName": "f3", "featureVersion": "1.0", "featureCount": 3 }`
	`]`

Error codes	glsErr.serverNotFound glsErr.optimisticLocking glsErr.reservationNotFound glsErr.invalidReservationRequest glsErr.serviceBusyDueToReservations

Sample Responses

For the GET method, the response contains information about the specified reservation (*rid*) within the reservation group (*gid*):

```
{
   "id" : 3,
   "lastModified" : "2019-02-11T05:39:23.664Z",
   "ipAddress" : "0:0:0:0:0:0:0:1",
   "reservationEntries" : [ {
     "id" : 5,
     "state" : "DISABLED",
     "featureName" : "f1",
    "featureVersion" : "1.0",
     "featureCount" : 1
   }, {
     "id" : 3,
     "state" : "ENABLED",
     "featureName" : "f1",
     "featureVersion" : "2.0",
     "featureCount" : 1
   } ],
   "hostId" : {
     "type" : "STRING",
     "value" : "dev1"
   }
}
```

No response is returned for the DELETE method.

For a successful POST method, the response contains a confirmation of success where *gid* is the reservation group to which the reservation entries for the given reservation were added:

```
{
uri: "/api/1.0/instances/{instanceid}/reservationgroups/{gid}"
status: "SUCCESS"
}
```

For a failed POST method, the response contains the error code and its corresponding message and a list of error arguments that varies based on the error. For example, you might see:

```
{
key: "glsErr.invalidReservationRequest"
message: "Invalid reservation request"
arguments: [1] 0: "feature f11 is not found"
}
```

Managing a Specific Reservation Entry

When the `GET` method is invoked, this endpoint returns data about a specific reservation entry, identified by an entry ID (*eid*), within the reservation (*rid*) in the reservation group (*gid*). The path parameters `{gid}`, `{rid}`, and `{eid}` must be of data type `number`.

URI	`/reservationgroups/{gid}/reservations/{rid}/entries/{eid}`
Method	`GET`
Query parameters	N/A
Request body	N/A
Error codes	`glsErr.serverNotFound` `glsErr.reservationNotFound`

When the `DELETE` method is invoked, the endpoint deletes the specified reservation entry:

URI	`/reservationgroups/{gid}/reservations/{rid}/entries/{eid}`
Method	`DELETE`
Query parameters	N/A
Request body	N/A

Error codes	`glsErr.serverNotFound`
	`glsErr.optimisticLocking`
	`glsErr.reservationNotFound`
	`glsErr.serviceBusyDueToReservations`
	`glsErr.assigningReservations`

When the `PUT` method invoked, this endpoint changes the current state (`DISABLED` or `ENABLED`) of the specific reservation entry (*eid*) to the state specified in the request:

URI	`/reservationgroups/{gid}/reservations/{rid}/entries/{eid}/`*`state`*
Method	`PUT`
Query parameters	N/A

Request body	The request body contains only the desired state—either “ENABLED” or “DISABLED”.
Error codes	glsErr.serverNotFound glsErr.optimisticLocking glsErr.reservationNotFound glsErr.invalidReservationRequest glsErr.serviceBusyDueToReservations glsErr.assigningReservations

Sample Responses

For the GET method, the response contains information about the specified reservation entry (*eid*):

```
{
  "id" : 5,
  "state" : "DISABLED",
  "featureName" : "f1",
  "featureVersion" : "1.0",
  "featureCount" : 4
}
```

No response is returned for the DELETE method.

For a successful PUT method, the response contains a confirmation of success where *gid* is the reservation group to which the reservation containing the reservation entry whose state was changed belongs:

```
{
uri: "/api/1.0/instances/{instanceid}/reservationgroups/{gid}"
status: "SUCCESS"
}
```

For a failed PUT method, the response contains the error code and its corresponding message and a list of error arguments that varies based on the error. For example, you might see:

```
{
key: "glsErr.invalidReservationRequest"
message: "Invalid reservation request"
arguments: [1] 0: "feature f1 does not have enough (unreserved) counts" }
```

APIs to Show Statistics

Statistics provide general information on the version and health of the specified license server.

Table 4-8 ▪ License Server REST APIs Used to Show Statisitcs

URI	Method	Description
/version	GET	Gets license server’s version information.
/health	GET	Gets basic license server status, such as whether a database connection check succeeds.

The following sections describe these APIs:

- Viewing Instance Version
- Viewing Success of Database Connection

Viewing Instance Version

The `/version` API returns JSON wrapping a version string and any related details.

URI	`/version`
Method	`GET`
Query parameters	N/A
Request body	N/A
Error codes	-

Sample Response


```
{
  "LLS" : {
    "version" : "2018.12",
    "buildDate" : "2019-01-02T13:38:17Z",
    "buildVersion" : "244007",
    "branch" : "release",
    "fneBuildVersion" : "229775"
  }
}
```

Viewing Success of Database Connection

The /health API returns process-level health information, such as whether a database connection check succeeds.

URI	/health
Method	GET
Query parameters	N/A
Request body	N/A
Error codes	-

Sample Responses

```
{
  "LLS" : {
    "version" : "2018.12",
    "buildDate" : "2019-01-02T13:38:17Z",
    "buildVersion" : "244007",
    "patchLevel" : "2019.01.0.0",
    "branch" : "release",
    "fneBuildVersion" : "229775",
    "trustStatus" : true,
    "database" : {
    "lastBackOfficeConnection": "2018-07-11T12:24:15.835Z",
      "connectionCheck" : "success"
    }
  }
}
```

The lastBackOfficeConnection element is returned only if the /health API is called at instance level, but not at global level. The element returns the timestamp of the last successful connection to the back office. The timestamp is updated each time a capability polling or synchronization attempt was successful. An attempt is considered to be successful when a connection to the back office for synchronization or for capability polling was established, irrespective of whether features have been updated.

The trustStatus element indicates whether a trust break has occurred.

APIs to Manage Instance Configuration

Configuration resources consist of metadata and policy values that configure the license server instance. The following APIs manage these policies on the license server.

Table 4-9 ▪ License Server REST APIs Used to Manage License Server Policies

URI	Method	Description
/configuration	`GET, PUT`	Gets or updates policy key-value pairs. (Certain policies are accessed at global level only.)
/configuration/metadata	`GET`	Returns the policy name-value pairs used in configuration, along with a description of each. (Certain policies are accessed at the policy level only.)

Refer to the following sections for more information about the APIs:

- Policies Accessed at a Global Level
- Viewing Configuration Metadata

For a description of license server policies, see Appendix A, Reference: Policy Settings for the License Server.

Policies Accessed at a Global Level

Most policies that the license server administrator can view or edit are done so at the instance level, using the base URL to access the `/configuration` API:

```
http://LicenseServerHostName/api/1.0/instances/instanceID/configuration
```

However, certain policies are edited by the administrator only at the global level. The following URL is used to access the `/configuration` API at a global level to edit these policies:

```
http://LicenseServerHostName/api/1.0/configuration
```

Policies that the license server administrator must access at a global level include the following. Note that these policies are editable only for the local license server, not the CLS instance.

- `database.backup-enabled`
- `graylog.host`
- `graylog.threshold`
- `lfs.syncTo.Threads`
- `licensing.clientExpiryTimer`
- `licensing.security.json.enabled`
- `logging.threshold`
- `security.enabled`
- `security.ip.whitelist`

- `security.http.auth.enabled`

Managing Instance Policies

The `/configuration` API obtains a list of the license server's current policy settings and enables you to edit policy values as needed.

View Current Policy Settings

When invoked with the `GET` method, this endpoint returns configuration settings for the specified scope.

URI	`/configuration`
Method	`GET`
Query parameters	`includeAll` Returns all settings, ignoring the visibility attributes.
Request body	N/A
Error codes	`glsErr.serverNotFound` `glsErr.configSetTenantRequired`

Sample Response

```
{
    "values": [
      "1d"
    ],
    "id": "licensing.responseLifetime",
    "parentValues": [
      "1d"
    ]
  },
  {
    "values": [
      "1w"
    ],
    "id": "licensing.borrowInterval",
    "parentValues": [
      "1w"
    ]
  },
  {
    "values": [
      "15"
    ],
    "id": "licensing.renewInterval",
    "parentValues": [
      "15"
    ]
```

```
  },
    }...
```

Set Policies

The `/configuration` API used with the `PUT` method sets or removes a policy. See Policies Accessed at a Global Level for changes required for the base URL use with the API to edit certain policies. For a description of license server policies, see Appendix A, Reference: Policy Settings for the License Server.

URI	`/configuration`
Method	`PUT`
Query parameters	N/A

Request body	Example:
	`{ "logging.threshold" : "WARN" }`
	To remove a setting, provide the key with a value of “null”, as in this example:
	`{ "logging.threshold" : null }`

Error codes	`glsErr.serverNotFound`
	`glsErr.configItemNotEditable`
	`glsErr.configSetTenantRequired`

Sample Response

```
[
  ...
    { "id" : "logging.threshold",
    "values" : [ "WARN" ],
    "parentValues" : [ "INFO" ] },
  ...
]
```

Viewing Configuration Metadata

The `/configuration/metadata` API returns configuration metadata for the specified scope. This includes grouping, sort order, visibility attributes, and descriptive strings which can support a data-driven UI for managing key-value pairs that support license server configuration.

URI	`/configuration/metadata`
Method	`GET`

Query parameters	tenant, enterprise, role, locale Query parameters tenant and enterprise can be used to get the configuration for a specific scope. If enterprise is specified, tenant is required. If none are specified, the default scope is used.
Request body	N/A
Error codes	glsErr.serverNotFound glsErr.configSetTenantRequired

Sample Response

```
[ {
  "order" : 0,
  "name" : "License Generation",
  "id" : "licensing",
  "hidden" : false,
  "description" : "Configuration settings that control license generation.",
  "editable" : false,
  "displayed" : true,
  "items" : [ {
    "type" : "DURATION",
    "defaultValue" : "1d",
    "required" : true,
    "multiValued" : false,
    "name" : "Response Lifetime",
    "id" : "licensing.responseLifetime",
    "hidden" : false,
    "description" : "The default response blah blah blah...",
    "editable" : false,
    "displayed" : true
  }, {
    "type" : "ENUM",
    "defaultValue" : "SECOND",
    "required" : true,
  "multiValued" : false,

    "name" : "Default Borrow Granularity",
    "id" : "licensing.defaultBorrowGranularity",
    "hidden" : false,
    "description" : "The default borrow granularity...",
    "editable" : false,
    "displayed" : true,
    "enumValues" : [
        { "value" : "DAY", "key" : "Day", "hidden" : false },
        { "value" : "HOUR", "key" : "Hour", "hidden" : false },
        { "value" : "MINUTE", "key" : "Minute", "hidden" : false },
        { "value" : "SECOND", "key" : "Second", "hidden" : false }
    ]
  } ]
} // , ... more groups
]
```

APIs to Perform Offline Binary Operations

For situations where the license server cannot communicate directly with FlexNet Operations, the license server provides endpoints for downloading a capability request, uploading a capability response, and performing an offline synchronization. Unlike the other endpoints, these endpoints involve binary data, not JSON content.

The following sections describe these APIs:

- API Summary
- Generating an Offline Activation Request
- Generating an Offline Capability Request
- Processing an Offline Capability Response
- Generating an Offline Sync Message
- Processing an Offline Sync Acknowledgment

API Summary

The following lists the APIs that handle operations offline:

Table 4-10 ▪ License Server REST APIs Used to Perform Offline Binary Operations

URI	Method	Description
/activation_request/offline	`POST`	Generates a new capability request containing the specified activation IDs to be sent to the back office and returns them as binary capability request data to save to a file.
/capability_request/offline	`GET, POST`	Generates a new capability request to be sent to the configured back-office URL for the specified instance and returns them as binary capability request data to save to a file rather than sending it to the back office.
/capability_response	`POST`	Takes a capability response whose target host is the instance identified in the URL and processes it, updating the served features of the instance as appropriate.
/sync_message/offline	`GET`	Downloads sync data for the specified server instance that is saved as a file to be uploaded to the back office.

Table 4-10 ▪ License Server REST APIs Used to Perform Offline Binary Operations (cont.)

URI	Method	Description
/sync_ack/offline	POST	Processes the sync acknowledgment sent by the back office to indicate that the latest instance data was successfully synchronized. The process updates the `lastSyncTime` property on the instance with this most recent synchronization time.

Generating an Offline Activation Request

The `/activation_request/offline` API generates the bytes of a new capability request, with an included set of activation IDs (rights IDs) and counts, and returns them as the response body.

URI	`/activation_request/offline`
Method	POST
Query parameters	`force` Forces an activation request to be sent immediately and forces the license server to generate the response immediately. Possible values are `true` and `false`.

Request body	`[ {"id" : "li1", "copies" : 3}, ... ]`
	
	Multiple `id` entries may be included in the request. If the `copies` field is omitted in an entry, it defaults to one copy.
	
	For any given `id` entry, you can also include the "`partial`" option to obtain all available copies of the rights ID if the requested count cannot be satisfied:
	
	`[ {"id" : "li1", "copies" : 3, "partial" : "true"}, ... ]`

Error codes	`glsErr.serverNotFound`
	`glsErr.serverNoHostid`
	`glsErr.parsingServerIdentity`
	`glsErr.missingServerIdentity`
	`glsErr.rightsIdMissing`
	`glsErr.rightsCopiesInvalid`

Sample Output

Instead of the typical JSON response, this API returns a binary body with the `application/octet-stream` content-type header.

Generating an Offline Capability Request

The /capability_request/offline API generates the bytes of a new capability request and returns them as the response body.

URI	/capability_request/offline
Method	GET,POST
Query parameters	force Forces a capability request to be sent immediately and forces the license server to generate the response immediately. Possible values are **true** and **false**.
Request body	application/octet-stream
Error codes	glsErr.serverNotFound glsErr.serverNoHostid glsErr.parsingServerIdentity glsErr.missingServerIdentity

Sample Output

Instead of the typical JSON response, this API returns a binary body with the application/octet-stream content-type header.

Processing an Offline Capability Response

The /capability_response API processes a capability response whose target host is the instance identified in the URL, updating the served features of the instance as appropriate.

URI	/capability_response
Method	POST
Query parameters	N/A
Request body	application/octet-stream

Error codes	glsErr.serverNotFound glsErr.serverNoHostid glsErr.parsingServerIdentity glsErr.missingServerIdentity glsErr.signatureInvalid glsErr.optimisticLocking glsErr.sslMisconfigured glsErr.connectionFailed glsErr.connectionFailedStatus glsErr.connectionRefused glsErr.parsingBinary glsErr.emptyMessage glsErr.wrongMessageType glsErr.licenseFailsAnchor glsErr.lfsError glsErr.lfsWrongMessageType glsErr.assigningReservations glsErr.serverNotFound glsErr.serverNoHostid glsErr.parsingServerIdentity glsErr.missingServerIdentity

Sample Response

Contents of the processed response is found in the license server logs. The following shows an example of logged response contents:

```
MessageType="Capability response"
ServerIDType=String
ServerID=BACK_OFFICE
SourceIds=BACK_OFFICE
Lifetime=86400
ResponseTime="Feb 3, 2019 11:20:44 AM"
RequestHostID=6C626DA08750
RequestHostIDType=Ethernet
HostUUID=dfb867a4-4cac-4e16-96c6-9a353231ed17
MachineType=Unknown
EnterpriseID=35331653
License=(Name=MeteredCappedRe Vendor=cmsperf5 Version=1.0
  Expires=1-jan-2022 Count=10 Metered Reusable
  ServerHostID=6C626DA08750/Ethernet HostID=ANY Issued=2-jun-2018
  Start=1-jan-2019 VendorString=42F9-634D-9B31-22CE
  Notice=AutomationAccount4 Issuer=FlexManufacturer
  SerialNumber=c8c83a8d-a1b5-4eac-9b94-ac38f8d21275)
License=(Name=MeteredUncappedRe Vendor=cmsperf5 Version=1.0
  Expires=1-jan-2022 Count=10 Metered Reusable UncappedOverdraft
  ServerHostID=6C626DA08750/Ethernet HostID=ANY Issued=3-mar-2018
  Start=1-jan-2019 VendorString=5132-B427-99A3-3DD1
  Notice=AutomationAccount4 Issuer="FlexNet Test Site"
  SerialNumber=f8414bcd-c4ea-4c7b-a997-3bef2c04750e)
```

Generating an Offline Sync Message

The /sync/message/offline API generates bytes of sync data accumulated since last sync data collection and returns them as the response body.

URI	/sync_message/offline
Method	GET
Query parameters	force Generates sync data accumulated since last successful sync time, as specified in the most recent sync acknowledgment processed on the server. Possible values are true and false.
Request body	application/octet-stream
Error codes	glsErr.serverNotFound glsErr.serverNotReady glsErr.parsingServerIdentity glsErr.missingServerIdentity glsErr.syncInProgress

Sample Response

Instead of the typical JSON response, this API returns a binary body with the application/octet-stream content-type header.

Additionally, an event similar to this is recorded in the license server logs:

```
16:20:56,360 INFO  Successfully processed offline sync request for instance 394RU2RH3AES
```

Processing an Offline Sync Acknowledgment

The /sync_ack/offline API processes the sync acknowledgment sent by the back office. This process updates the current synchronization timestamp on the instance with the timestamp sent in the acknowledgment. The new timestamp then becomes the latest time that data on the instance was successfully synchronized to the back office.

URI	/sync_ack/offline
Method	POST
Query parameters	N/A
Request body	application/octet-stream

Error codes	glsErr.serverNotFound glsErr.serverNotReady glsErr.parsingServerIdentity glsErr.missingServerIdentity glsErr.optimisticLocking glsErr.parsingBinary glsErr.emptyMessage glsErr.wrongMessageType glsErr.lfsWrongMessageType glsErr.syncInProgress

Sample Response

An event similar to the following is recorded in the license server logs:

```
Successfully processed sync acknowledgment
```

If there is a mismatch in the sync time the following message is returned:

```
Sync acknowledgment has mismatched start time
```

A mismatch in sync time can happen for various reasons, and usually (but not always) the `-force` option available with the `serverofflinesync` tool can resolve the problem. For more information on how to correct a sync-time mismatch, refer to the description of the `-force` option in Offline Synchronization to the Back Office in Chapter 5, More About Basic License Server Functionality.

API to View Binding-Break Status

If you have configured the binding-break detection feature, as described in Binding-Break Detection with Grace Period in Chapter 6, Advanced License Server Features, use the following REST API to view the current binding status of the license server.

URI	/binding
Method	GET
Query parameters	N/A
Request body	N/A
Error codes	N/A

Sample Response

```
{
  "bindingPolicy": "120 seconds",
  "bindingStatus": "soft break",
  "gracePeriodExpiry": "2019-02-16T12:50:43Z"
}
```

The `bindingPolicy` is the `binding.break.policy` setting defined in `producer-settings.xml`, indicating the action taken when a binding break is detected:

- **hard**—An immediate hard binding break will be imposed—that is, the license server can no longer serve licenses.
- **soft**—A soft binding break will go into effect, allowing the license server to continue to serve licenses despite the break.
- ***n* seconds**—A soft binding break with a grace period will go into effect, allowing the license server to serve licenses until the grace period expires. Then a hard binding break goes into immediate effect, and licenses can no longer be served. The length of the grace period is expressed as the `n seconds` value.

The `bindingStatus` property is **ok** when no break is detected; or, when a break is detected, one of the following displays:

- **hard break**—The license server can no longer serve licenses. If the hard break has occurred because a grace period has expired, the `gracePeriodExpiry` property is included to show the expiration timestamp (in ISO-8601 format).
- **soft break**—The license server continues to serve licenses despite binding break. If the soft break has occurred because a grace period is currently in effect, the `gracePeriodExpiry` property is included to show the timestamp (in ISO-8601 format) when the grace period expires.

APIs to Manage Administrative Security

When administrative security is enabled on the CLS instance or local license server, the following APIs are used to authenticate a user's credentials and enable the creation and management of user accounts on the license server. See Administrative Security in Chapter 6, Advanced License Server Features, for details about enabling and managing administrative security on the license server.

The following sections describe the APIs:

- APIs to Authorize Access to Secured Endpoints
- APIs to Manage User Accounts

APIs to Authorize Access to Secured Endpoints

This section describes those APIs used to authorize a user's access to other secured REST APIs, enabling the user account to perform those administrative tasks to which it has privileges:

- API Overview
- Generate Authorization Tokens
- Revoke Tokens

API Overview

The following are the APIs used to authenticate a user's credentials and thus give the user access to other secured REST APIs. The /revoke API invalidates current user credentials on the license server.

Table 4-11 • License Server REST APIs Used to Access Secured Endpoints

URI	Method	Description
/authorize	POST	Returns a JSON Web Token for use on secured endpoints.
/revoke	PUT	Invalidates existing JSON Web Tokens on the instance.

Generate Authorization Tokens

When a user provides credentials to perform administrative operations, the following APIs use these credentials to generate a JSON Web Token, which is then applied to subsequent REST API calls to authorize the user's access to the secured endpoints.

URI	/authorize
Method	POST
Query parameters	N/A

Request body	Example:
	`{`
	`  "password": "Admin01!",`
	`  "user": "admin"`
	`}`

Error codes	glsErr.restParsing glsErr.userAuthFailed

Sample response

For a successful POST method, you receive the JSON Web Token generated for the user credentials and its expiration time:

```
{
  "expires": "2018-08-24T17:44:32.573Z",
  "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzUxMiJ9..." [code truncated for example purposes]
}
```

For a failed POST method, the response contains the error code and its corresponding message and a list of error arguments that varies based on the error. For example, you might see:

```
{
  "message": "Authorization attempt at uri=/api/1.0/instances/~/authorize failed for user d
    (error BadCredentialsException)",
```

```
    "key": "glsErr.userAuthFailed",
    "arguments": [
      "uri=/api/1.0/instances/~/authorize",
      "d",
      "BadCredentialsException"
    ]
  }
```

Revoke Tokens

Use the following API to invalidate current JSON Web Tokens on the instance.

URI	`/revoke`
Method	PUT
Query parameters	N/A
Request body	N/A
Error codes	`glsErr.userAuthFailed`

No response is returned after the tokens are successfully revoked.

APIs to Manage User Accounts

This section describes those APIs used to create, modify, and delete user accounts on the license server:

- API Overview
- View Current User Accounts
- Create a User Account
- Modify an Existing User Account
- Delete a User Account

API Overview

The following APIs are used to manage user accounts for an instance. Each user account is assigned specific roles that enable the user to perform certain administrative tasks. The credentials defined for each account give the user access to only those secured REST APIs appropriate to the assigned roles.

Administrative credentials are needed to access the `/users` APIs described here.

For more information about roles, see Administrative Security in Chapter 6, Advanced License Server Features.

For information about the APIs used to process credentials, see APIs to Authorize Access to Secured Endpoints.

Table 4-12 ▪ License Server REST APIs Used to Manage User Accounts for Security

URI	Method	Description
/users	GET	Returns details for all user accounts or details for the account identified by the user name.
/users/{id}	GET	Returns details about the user account identified by its ID.
/users	POST	Creates a new user account.
/users	PUT	Updates the password or roles for an existing user account identified by the user name.
/users/{id}	PUT	Updates the password or roles for an existing user account identified by its ID.
/users/{id}	DELETE	Deletes a user account identified by its ID.

View Current User Accounts

The following APIs obtain details about current user accounts set up for the instance. Administrative credentials are needed to access the APIs described here.

View All Accounts or Specific Account Identified by User Name

This API can retrieve all existing accounts or the specific account with a user name identified by the `name` parameter.

URI	`/users`
Method	`GET`
Query parameters	`name, page, size` See section *Query Parameters*, below.
Request body	N/A

Error codes	`glsErr.restPathVariableParsing`
	`glsErr.userAccessDenied`
	`glsErr.userAuthFailed`

Query Parameters

The following are optional parameters for use in GET requests:

name	Retrieves a specific account. Must be a string. Cannot be used in combination with the page and size parameters.
page	Retrieves a specific page. Must be a number. Cannot be used in combination with the name parameter. See also Managing Large Amounts of Returned JSON Data.
size	Specifies how many records per page should be returned. Must be a number. Cannot be used in combination with the name parameter. See also Managing Large Amounts of Returned JSON Data.

Sample response

For a successful GET method, you receive a list of all existing user accounts and their detail. If you entered a specific user name (using the name parameter), you receive details for that account only. The following shows an excerpt from a listing of all user accounts:


```
[ {
  "id" : 1,
  "user" : "producer",
  "enabled" : true,
  "userExpiry" : "permanent",
  "roles" : [ {
    "id" : 3,
    "role" : "ROLE_DROPCLIENT"
  }, {
    "id" : 7,
    "role" : "ROLE_PRODUCER"
  }, {
    "id" : 4,
    "role" : "ROLE_RESERVATIONS"
  }, {
    "id" : 6,
    "role" : "ROLE_READ"
  } ]
}, {
  "id" : 2,
  "user" : "admin",
  "enabled" : true,
  "userExpiry" : "permanent",
  "roles" : [ {
    "id" : 8,
    "role" : "ROLE_ADMIN"
  }, {
    "id" : 5,
    "role" : "ROLE_DROPCLIENT"
  }, {
    "id" : 2,
    "role" : "ROLE_RESERVATIONS"
  }, {
```

```
      "id" : 1,
      "role" : "ROLE_READ"
    } ]
  }, {
    "id" : 3,
    "user" : "admin01",
    "enabled" : true,
    "userExpiry" : "permanent",
    "roles" : [ {
      "id" : 11,
      "role" : "ROLE_ADMIN"
    }, {
      "id" : 12,
      "role" : "ROLE_DROPCLIENT"
    }, {
      "id" : 10,
      "role" : "ROLE_RESERVATIONS"
    }, {
      "id" : 9,
      "role" : "ROLE_READ"
    } ]
  }, {
    "id" : 5,
    "user" : "user002",
    "enabled" : true,
    "userExpiry" : "permanent",
    "roles" : [ {
      "id" : 14,
      "role" : "ROLE_READ"
    } ]
  },
```

For a failed GET method, the response contains the error code and its corresponding message and a list of error arguments that varies based on the error. For example, you might see:

```
{
  "message": "Authorization attempt at uri=/api/1.0/instances/~/users failed for user (unknown)
(error BadCredentialsException)",
  "key": "glsErr.userAuthFailed",
  "arguments": [
    "uri=/api/1.0/instances/~/users",
    "(unknown)",
    "BadCredentialsException"
  ]
}
```

View Specific User Account Identified by System ID

To list a specific user account identified by its system-generated ID, use the following API. (This is the “id” item listed for each user account in the response returned for /users using the GET method.) The path parameter {id} must be of data type number.

URI	/users/{id}
Method	GET

Query parameters	N/A
Request body	N/A

Error codes	`glsErr.restPathVariableParsing`
	`glsErr.userAccessDenied`
	`glsErr.userAuthFailed`

Sample response

For a successful GET method, you receive details for the account identified by its ID.

Create a User Account

This API creates a user account with credentials and one or more roles. For more information about roles, see Administrative Security in Chapter 6, Advanced License Server Features. Administrative credentials are needed to access the API.

URI	`/users`
Method	POST
Query parameters	N/A

Request body	Example:
	`{`
	`  "password": "User@001",`
	`  "roles": [`
	`       "ROLE_RESERVATIONS"`
	`  ],`
	`  "user": "user001"`
	`}`

Error codes	`glsErr.restParsing`
	`glsErr.userAccessDenied`
	`glsErr.userAuthFailed`

For a successful POST method, you receive details about the newly created account.

For a failed POST method, the response contains the error code and its corresponding message and a list of error arguments that varies based on the error. For example, you might see:

```
{
  "message": "Access denied to uri=/api/1.0/instances/~/users for user (unknown)",
  "key": "glsErr.userAccessDenied",
  "arguments": [
    "uri=/api/1.0/instances/~/users",
    "(unknown)"
  ]
}
```

Modify an Existing User Account

These APIs edit the password or roles or both for an existing user account, identified by either its user name or system-generated ID. Administrative credentials are needed to access the API.

Modify a User Account Identified by User Name

The following API updates a user account identified by its user name identified by the `name` parameter.

URI	`/users`
Method	`PUT`
Query parameters	`name` Mandatory parameter to update a specific user account. Value must be of type `string`.

Request body	Example:
	`{`
	`"password": "User001!",`
	`"roles": [`
	`{`
	`"role": "ROLE_RESERVATIONS"`
	`},`
	`{`
	`"role": "ROLE_READ"`
	`} ],`
	`"user": "user001"`
	`}`

Error codes	`glsErr.restParsing`
	`glsErr.restPathVariableParsing`
	`glsErr.userAccessDenied`
	`glsErr.userAuthFailed`

Sample response

For a successful `PUT` method, you receive details for the updated account.

```
{
  "id" : 9,
  "user" : "user001",
  "enabled" : true,
  "userExpiry" : "permanent",
  "roles" : [ {
    "id" : 19,
    "role" : "ROLE_RESERVATIONS"
  }, {
    "id" : 20,
    "role" : "ROLE_READ"
  } ]
}
```

For a failed PUT method, the response contains the error code and its corresponding message and a list of error arguments that varies based on the error. For example, you might see:

```
{
  "message": "Access denied to uri=/api/1.0/instances/~/users for user (unknown)",
  "key": "glsErr.userAccessDenied",
  "arguments": [
    "uri=/api/1.0/instances/~/users",
    "(unknown)"
  ]
}
```

Modify a User Account Identified by System ID

The following API updates a user account identified by its system-generated ID. (This is the “id” item listed for each user account in the response returned for /users using the GET method.) The path parameter {id} must be of data type number.

URI	/users/{id}
Method	PUT
Query parameters	N/A

Request body	Example:
	{
	  "password": "User@002",
	  "roles": [
	      "ROLE_RESERVATIONS"
	  ],
	  "user": "user002"
	}

Error codes	glsErr.restParsing glsErr.restPathVariableParsing glsErr.userAccessDenied glsErr.userAuthFailed

For a successful PUT method, you receive details for the updated account:

```
{
  "id" : 10,
  "user" : "user002",
  "enabled" : true,
  "userExpiry" : "permanent",
  "roles" : [ {
    "id" : 21,
    "role" : "ROLE_RESERVATIONS"
  } ]
}
```

For a failed PUT method, the response contains the error code and its corresponding message and a list of error arguments that varies based on the error. For example, you might see:

```
{
```

```
    "message": "Access denied to uri=/api/1.0/instances/~/users for user (unknown)",
    "key": "glsErr.userAccessDenied",
    "arguments": [
      "uri=/api/1.0/instances/~/users",
      "(unknown)"
    ]
  }
```

Delete a User Account

This API deletes an existing user account, identified by the system-generated ID. Administrative credentials are needed to access the API. (The ID can be obtained from the response returned for `/users` using the `GET` method.) The path parameter `{id}` must be of data type `number`.

URI	`/users/{id}`
Method	`DELETE`
Query parameters	N/A
Request body	N/A

Error codes	`glsErr.restPathVariableParsing`
	`glsErr.userAccessDenied`
	`glsErr.userAuthFailed`

For a successful `DELETE`, the status code 410 is returned, but no response body is provided.

For a failed `DELETE` method, the response contains the error code and its corresponding message and a list of error arguments that varies based on the error. For example, you might see:

```
{
  "message": "REST request path variable problem: type mismatch on path component for
/api/1.0/instances/~/users/user002!",
  "key": "glsErr.restPathVariableParsing",
  "arguments": [
    "/api/1.0/instances/~/users/user002!"
  ]
}
```

API for Enabling JSON Security for the Cloud Monetization API

The FlexNet Embedded CLS instance and local license server support a collection of functions and interfaces called the FlexNet Embedded Cloud Monetization API (CMAPI), which performs capability exchanges using REST APIs, instead of the FlexNet Embedded SDK language-based APIs that you incorporate in your client code.

For details how to use the CMAPI to perform actual capability exchanges, refer to the *Cloud Monetization API User Guide*.

Purpose of this Section

This section focuses on the process of setting up JSON security for these capability exchanges. When you use SDK-driven licensing functionality, the binary messages sent in capability exchanges are secured with encrypted signatures derived from the producer identity and other identification elements. However, when you use the Cloud Monetization API (CMAPI), the message bodies sent in capability exchanges are in JSON format and thus require a form of security suitable for JSON messaging.

The setup for JSON security largely involves the generation of a public-private key pairs and then the transmission of the appropriate public key between the client and the license server (and vice versa)—a task mainly handled by the
`/rest_licensing_keys` REST API.

The following topics provide the necessary details:

- License Server Administration Security vs JSON Security
- HTTPS Protocol Recommended
- Implementing JSON Security for the Cloud Monetization API
- “/rest_licensing_keys” API Details
- Sample Code Implementations for Setting Up JSON Security

License Server Administration Security vs JSON Security

The FlexNet Embedded license server uses different access control strategies on its REST endpoints according to use case—APIs used to administer the license server and APIs used to directly license clients.

When license-server administration security is enabled, users must provide valid log-in credentials. These credentials are used internally to generate a JSON Web Token, which, in turn, is used to give the user appropriate access to REST APIs that manage the server (see Writing a REST API Client for a Secured License Server and APIs to Manage Administrative Security).

However, providing log-in credentials is not appropriate for securing JSON capability exchanges performed by the CMAPI. Instead, JSON capability exchanges use a combination of JSON Web Tokens and JSON Web Signatures to secure the incoming capability requests on the license server and to sign the capability responses that the license server sends back to the client.

The next sections provide instructions for implementing JSON security for capability exchanges performed by the CMAPI.

Note About Opting Out of JSON Security

Although implementing JSON security for the CMAPI is strongly recommended, it is optional for local license servers. Keep in mind that setting the `licensing.security.json.enabled` license server policy to `false` (see Step 2: Producer or Enterprise Configures Appropriate Security Policies on License Server) not only allows unauthenticated capability requests, but also disallows requests attempting to use authentication. Hence, disabling JSON security is a critical decision that must be reflected in the client application.

JSON security is implemented by default for CLS instances.

HTTPS Protocol Recommended

When JSON security is implemented on a local license server to secure capability exchanges performed by the CMAPI, the use of the HTTPS protocol is strongly recommended. This protocol protects the machine-to-machine transmission of sensitive data, specifically the JWT used to secure capability requests and the authorization credentials needed to access the `/rest_licensing_keys` REST API (described later in this section).

The requirement for HTTPS cannot be disabled for CLS instances.

Implementing JSON Security for the Cloud Monetization API

The following steps are used to secure the JSON capability exchanges in the CMAPI:

- Step 1: Producer Generates Private-Public Key Pair
- Step 2: Producer or Enterprise Configures Appropriate Security Policies on License Server
- Step 3: Producer or Enterprise Uploads Public Key to License Server
- Step 4: Licensed Client Application Generates JWT Signed by Private Key to Include in Capability Requests

Step 1: Producer Generates Private-Public Key Pair

You, as producer, must generate a private and public key pair in RSA 2048-bit format for use by the client application. Various third-party tools are available for generating this key pair. Refer to the documentation supplied with the tool for instructions.

This key pair is instrumental is enabling the license server to authenticate an incoming JSON capability request. The public key is uploaded to the license server, as described in Step 3: Producer or Enterprise Uploads Public Key to License Server. (Note that the public key must be uploaded in DER format as a byte array; the server does not accept the PEM format.)

The private key, which must be kept “secret” by you or the enterprise, is used to sign a generated JSON Web Token (JWT) that will be attached to each capability request, as described in Step 4: Licensed Client Application Generates JWT Signed by Private Key to Include in Capability Requests. Then, when a client sends a JSON capability request to the license server, the server determines that the request is from a valid user only if the JWT can be read using the public key on the server, implying that the JWT was signed by the matching private key.

For an example on how to generate this key pair, see the later section Example for “Step 1: Producer or Enterprise Generates Private-Public Key Pair”.

Consideration

You might want to generate different key pairs and tokens for a CLS instance or given local license server. The responsibility of generating these additional key pairs could potentially be pushed to the enterprise.

Step 2: Producer or Enterprise Configures Appropriate Security Policies on License Server

For CLS instances, the policies used to configure JSON security are already set for you.

However, for local license servers, you or the enterprise license server administrator must ensure that certain license server policies are appropriately configured, as described here:

- JSON security for the CMAPI requires that the `licensing.security.json.enabled` license server policy be set to **`true`** (which is the default value). This policy forces the licensed client application to attach the JWT to each capability request sent to the license server.
- Setup for JSON security requires that the `security.enabled` license server policy be set to **`true`**, thus enabling license-server administrative security. Access to the `/rest_licensing_keys` REST API (used to upload the public key to the license server) requires valid license-server administrator credentials. The use of credentials is supported only when license server-administrative security is enabled. For more information about providing administrative credentials and generating the producer authorization token, see the following:
    - Administrative Security section in Chapter 6, Advanced License Server Features
    - APIs to Manage Administrative Security section in this chapter
- The license server policy, `security.http.auth.enabled`, made available only when license-server administration security is enabled, applies not only to REST APIs that administer the license server but also to the `/access_request` or `/signed_access_request` REST APIs, used to perform capability exchanges in the CMAPI. When set to **`false`**, this policy enforces the use of HTTPS to access the REST APIs, a protocol especially critical for the secure transmission of the JWT in capability requests. Basically, the use of the JWT as a security mechanism is meaningless unless its transmission to the license server is protected. Hence, setting this policy to **`false`** is strongly recommended.

For more information about setting license server policies, see the following sections:

- License Server Configuration Utility section in Chapter 7, Producer Tools, for instructions on how to generate the `producer-settings.xml` file, containing the license policies shipped with a local license server
- APIs to Manage Instance Configuration section in this chapter for a description of the REST APIs used to override current license server policies
- Reference: Policy Settings for the License Server appendix for descriptions of the license server policies

Step 3: Producer or Enterprise Uploads Public Key to License Server

You or the enterprise uploads the public key, generated in the process described in Step 1: Producer Generates Private-Public Key Pair, to the license server.

This upload process is performed using the `POST` method on the `/rest_licensing_keys` REST API. Note that this process not only uploads the public key for JWT validation of incoming capability requests but also triggers processes on the license server that enable response-signing. License-server administrator credentials are required to access this API using the `POST` method.

Any appropriate REST API administrative client—for example, the FlexNet License Server Administrator command-line tool (`flexnetlsadmin`) or a third-party tool such as Postman—can be used to call this API.

The following sections provide more description about the processes performed by the `/rest_licensing_keys` REST API:

- Enablement of JWT Validation for Incoming Capability Requests
- Response-Signing Enablement

For an example procedure that uploads the public key, see the later section, Example for “Step 3: Producer or Enterprise Uploads Public Key to License Server”.

For details about the `/rest_licensing_keys` REST API itself, see “/rest_licensing_keys” API Details.

Enablement of JWT Validation for Incoming Capability Requests

The public key, once uploaded to the license server, enables the server to validate the JWT sent in incoming capability requests to determine whether the requests have been issued by authorized clients in the enterprise. When the public key is posted to the license server, the server checks that the key is in the correct format and, if it is, stores the key against the its identity data in the database.

Response-Signing Enablement

When the public key needed for the JWT validation of incoming capability requests on the license server is posted to the
`/rest_licensing_keys` REST API, the following additional processes are triggered on the license server to enable response-signing:

- Another private-public key pair that will be used for response-signing (if this pair does not already exist on the server) is generated. The private key will be used by the license server to generate a JSON Web Signature (JWS) for each capability response to a request posted using `/signed_access_request`. The public key will be used by the licensed client application to validate that the JWS with which a given capability response is signed was generated from the matching private key, thus authenticating the license server.

  The response-signing key pair is stored in the license server’s trusted storage, along with the public key used by the license server to validate incoming capability requests.
- A response message containing the public key (from the response-signing key pair) is returned to the REST API administrative client. You or the enterprise is responsible for providing this key to the licensed client application. The key must be stored in a location accessible to the client application for easy retrieval.

Even though response-signing is automatically enabled when the public key for the JWT is posted to the license server, the specific API that the licensed client application uses to request features—`/access_request` or `/signed_access_request`—determines whether or not the capability response is signed. These APIs are discussed in the chapter, *Performing a JSON Capability Exchange*, found in the *Cloud Monetization API User Guide*.

Once the public key for the JWT is posted to the license server, an occasion might require the retrieval of the public key stored on the server and used by the client for JWS validation. The `GET` method on `/rest_licensing_keys` provides access to this key. For more information, see Method to Retrieve the Signature-Validation Public Key Stored on Server.

Step 4: Licensed Client Application Generates JWT Signed by Private Key to Include in Capability Requests

Your licensed client application is provided with or generates a JWT signed by the private key that matches the public key uploaded to the license server (described in the previous section, Step 3: Producer or Enterprise Uploads Public Key to License Server) to validate incoming capability requests.

The generated token must be short-term, claim ROLE_CAPABILITY (so that server can differentiate it from other tokens processed on the server), and use the signing algorithm RS256. For an example implementation that generates the JWT, see the later section, Example for “Step 4: Licensed Client Application Generates JWT Signed by Private Key to Include in Capability Requests”.

Your application code must include this JWT with the capability requests sent by your client software to the license server. The license server then validates the token in the request against the public key stored on the server to authenticate the request. This validation process is described in more detail in the *Cloud Monetization API User Guide*.

Keep in mind that this JWT can only be used only with `/access_request`, `/signed_access_request`, or `/preview_request`. It cannot be used to access other REST APIs that are secured and used for administrative purposes; access to these APIs require instead the JWT (also called the *producer authorization token*) obtained from the user credentials posted on the `/authorize` endpoint, as described in Authorizing Access to Secured REST APIs. (Note that access to the
`/rest_licensing_keys` API requires the producer authorization token obtained from user credentials posted on the
`/authorize` endpoint.)

Consideration

As an extra security measure, you or the enterprise can put a time limit on the JWT produced with the private key and then periodically regenerate and distribute a new JWT with an updated time limit.

“/rest_licensing_keys” API Details

The following shows details about the `/rest_licensing_keys` API. This API should be accessed by you or the enterprise license-server administrator using a REST API administrative client.

- Method to Post Public Key on Server to Validate JWT in Capability Requests
- Method to Retrieve the Signature-Validation Public Key Stored on Server
- Method to Delete the Server’s Public Key Used to Validate JWT in Capability Requests

Method to Post Public Key on Server to Validate JWT in Capability Requests

The `POST` method on the `/rest_licensing_keys` API uploads the public key generated on the client to the license server, enabling the server to validate the JWT sent in capability requests. The `POST` method also generates another public-private key pair used for response-signing on the license server (if keys do not already exist on the server). The key pair is stored in license-server trusted storage, but the public key is sent in a response body. The producer or enterprise license server- administrator must provide the public key to the client applications performing response-signature (JWS) validation.

The POST method requires license-server administrator credentials (ROLE_ADMIN).

URI	/rest_licensing_keys
Method	POST
Query parameters	N/A
Request body	application/octet-stream
Error codes	glsErr.serverNotFound glsErr.userAccessDenied glsErr.userAuthFailed

Method to Retrieve the Signature-Validation Public Key Stored on Server

If you or the enterprise needs to retrieve the public key that is stored on the license server and used by the client to validate the JWS on capability responses, you can use the GET method on the /rest_licensing_keys API to access to this key.

This method requires READ permission only. However, keep in mind that the security provided by the signed responses is undermined should the client application use this endpoint directly to access this public key on the server. It is the producer or enterprise's responsibility to retrieve and distribute this key using a separate mechanism, as appropriate for the application.

URI	/rest_licensing_keys
Method	GET
Query parameters	N/A
Request body	N/A
Error codes	glsErr.serverNotFound glsErr.userAccessDenied glsErr.userAuthFailed

Method to Delete the Server’s Public Key Used to Validate JWT in Capability Requests

The DELETE method on the /rest_licensing_keys API removes the license server’s public key previously uploaded from the client and used by the server to validate the JWT sent in capability requests.

Be aware that, as long as licensing.security.json.enabled is set to true, this operation disables all use of the access_request and signed_access_request REST APIs.

The DELETE method requires license-server administrator credentials (ROLE_ADMIN).

URI	/rest_licensing_keys

Method	DELETE
Query parameters	N/A
Request body	N/A
Error codes	glsErr.serverNotFound glsErr.userAccessDenied glsErr.userAuthFailed

Sample Code Implementations for Setting Up JSON Security

The following provides example code and commands for setting up security for the CMAPI. The examples apply to previous sections describing how to prepare the client and license server environments for this type of licensing.

- Example for “Step 1: Producer or Enterprise Generates Private-Public Key Pair”
- Example for “Step 3: Producer or Enterprise Uploads Public Key to License Server”
- Example for “Step 4: Licensed Client Application Generates JWT Signed by Private Key to Include in Capability Requests”

Example for “Step 1: Producer or Enterprise Generates Private-Public Key Pair”

The following example demonstrates one method for generating the private and public key pair (as described in Step 1: Producer Generates Private-Public Key Pair) that will be used to generate and validate the JWT sent in the capability requests. While you can use any appropriate third-party tool to generate this key pair, this example uses OpenSSL, a common and widely used cryptographic library that is available pre-built for many platforms. (You can easily download and install an OpenSSL version appropriate for your operating system.)

The following sample OpenSSL command generates a private key:

```
openssl genrsa -out SaaSDemoPrivate.pem 2048
```

The public key is then generated from the private key and, as shown in this command, is output in DER format, ready to be uploaded to the license server:

```
openssl rsa -inform PEM -outform DER -in SaaSDemoPrivate.pem -pubout -out SaaSDemoPublic.der
```

Example for “Step 3: Producer or Enterprise Uploads Public Key to License Server”

The public key (generated client-side along with the private key used to create the JWT sent with requests) must be uploaded to the license server so that the server can validate the JWT. You or the enterprise license-server administrator can use any appropriate REST API client tool—a third-party tool or one shipped with the license server—to upload this key, as described in Step 3: Producer or Enterprise Uploads Public Key to License Server.

The following example shows basic command syntax used to upload the public key using the FlexNet License Server Administrator command-line tool:

```
flexnetlsadmin -server licenseServer_baseURL -authorize adminName {adminPassword|
    -passwordConsoleInput} -uploadPublicKey clientPublicKeyFileName
```

This command uses the following variables:

- **licenseServerBaseURL**—The license server base URL used to access REST APIs. See Base URLs for details.
- **adminName {adminPassword|-passwordConsoleInput}**—The license-server administrator credentials required to access the `/rest_licensing_keys` REST API. You or the enterprise can provide the password directly in the command; or, to avoid exposing the password as a command-line argument, you can use the `-passwordConsoleInput` option, which prompts for the password as console input.

  Note ▪ *(Linux only) If you supply a password directly on the command line you may need to escape some characters with a backslash in order to prevent the shell from interpreting them. The safest approach is to surround the whole password with single quotes, and backslash any single quote that is part of the password.*

- ***clientPublicKeyFileName***—The name of the binary file containing the DER-formatted public key generated in Step 1: Producer Generates Private-Public Key Pair.

The following shows a sample FlexNet License Server Administrator command that uploads the public key contained in the `SaaSDemoPublic.der`, created for the previous example:

```
flexnetlsadmin -server https://ABC.compliance.flexnetoperations.com/api/1.0/instances/XYZ10203040
    -authorize admin AV11krl -uploadPublicKey SaaSDemoPublic.der
```

Example for "Step 4: Licensed Client Application Generates JWT Signed by Private Key to Include in Capability Requests"

The following example provides a sample script that generates a JWT signed with the client's private key, as discussed in Step 4: Licensed Client Application Generates JWT Signed by Private Key to Include in Capability Requests. This script was created in Python, but you can use any appropriate programmatic method to generate the JWT. (Note that the https://jwt.io website also provides a graphical tool that can be used to achieve the same result as a programmatic method.)

The licensed client application runs such a script to create the JWT, which the application then includes as the authorization header for each capability request it sends to the license server.

These sections describe preparing and running this example implementation:

- Requirements for Running the Sample Python Script
- Sample Python Script
- Command to Run Script

Requirements for Running the Sample Python Script

The following are requirements for running the sample script, which was developed using Python version 3.7.0a2.

Python Modules

Certain Python modules are required to run this script and can be installed using these commands:

- `pip install PyJWT`
- `pip install cryptography`
- `pip install cffi`

Development Libraries

To install these modules successfully, you might need to have certain development libraries already installed. For example, Debian-based Linux systems might require the `libffi-dev` and `libssl-dev` libraries, installed using a command like this:

```
apt-get install libffi-dev libssl-dev
```

Sample Python Script

The following sample is a basic script that generates a JWT signed with the client's private key.

Note that the script defines an optional JWT expiration that the producer or enterprise can update (or not use) as suits their deployment model. This time limit can be used as an extra security measure, forcing a periodic regeneration and distribution of a new JWT with an updated expiration.

```
#! /usr/bin/python
import jwt
import datetime
import sys

# Read the supplied private key file
if len(sys.argv) < 2: exit("Please supply private key as a PEM file")

private_key = open(sys.argv[1], 'r')
private_key_string = private_key.read();
private_key.close()

# Setup the JWT's payload
payload = {
    # Subject field must be present but content is not significant
    'sub': 'Authorization',
    # Claim must include 'roles' as ROLE_CAPABILITY
    'roles': 'ROLE_CAPABILITY',
    # Other JWT, such as expiration, are optional
    'exp': datetime.datetime.utcnow() + datetime.timedelta(days=365)
}

# Use PyJWT to encode and sign a JWT
token = jwt.encode(
    payload,
    private_key_string,
    algorithm='RS256')
```

```
# Present token to the user
print('Bearer ' + token.decode('ASCII'))
```

Command to Run Script

To run this script, the licensed client application would use following command:

```
./SignJWT.py SaaSDemoPrivate.pem
```

The generated JWT is provided as output:

```
Bearer
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1NDI5OTk5NzgsInN1YiI6IkF1dGhvcml6YXRpb24iLCJyb2xlcy
I6IlJPTEVfQ0FQQUJJTElUWSJ9.be7LU7kjWuSP8hqKuUIY_7oaAUqTGKqKKYoAR6q2sZrbRQpi68Y1tObF7ynDY1OlfHoFfoHB
bgOtPUagJLnyzJcmqIlAdJUtMiqrr-d2YM843Cuv4uWH5JU76EiP53NIyDGOm-
EVYC_r_OAihF9BzmZA8XBNj0poD1ZxyVpW31pgbIJcj5qDwN5mmXEDNBsIdCe_n0AOmTZpGi1kcuPKPMALeF5Shs_D4am1nfwtT
m4r7G-VRyPTCZ9x8qlffJrolDIPh8kbNRRJpuDlG8T-9CB1n8qK0FDm5MoglyB4f6UYeh5xTx4qjGTJ9Axx25-
OdpADqsaavALBBIEc0gMeyA
```

Refer the *Cloud Monetization API User Guide* for information about attaching this JWT to the capability request and for an example script that the client application can use to validate the JWT.

Online REST API Reference Documentation Available

Online REST API reference documentation is available to help you author a license-server administrative client or to secure and perform capability exchanges using the Cloud Monetization API. This online documentation can be accessed at the following `/documentation` endpoint:

http://*licenseServer_URL*:*port*/documentation

Exception Handling

The HTTP status code is used to signal success or failure of a REST API invocation. The most common status codes include the following:

Table 4-13 ▪ License Server Status Codes

Status Code	Meaning
OK (200)	The operation was successful.
BAD_REQUEST (400)	The request was invalid. For example, perhaps a query parameter was invalid, or the request body contained invalid content.
NOT_FOUND (404)	Either some resource was not found, or the API does not exist.

Table 4-13 ▪ License Server Status Codes (cont.)

Status Code	Meaning
GONE (410)	For APIs that use the DELETE method, this signals the deletion was successful.
INTERNAL_SERVER_ERROR (500)	A problem occurred in handling the request. For example, perhaps the request body could not be parsed.

For status codes other than OK and GONE, the returned response body will describe the exception, as shown in the following example:

```
{
  "message" : "No such REST API and METHOD combination supported: /api/1.0/bad_request with POST",
  "key" : "glsErr.restNoSuchApi",
  "arguments" : [ "/api/1.0/bad_request", "POST" ]
}
```

The exception contains the formatted message, a key that can be used in code to test against, or as a resource name to localize the message, and, optionally, any arguments that were used when formatting the message.

List of Exceptions

The following table enumerates the errors the license server may generate:

Table 4-14 ▪ License Server Errors

Exception	Definition
glsErr.alreadyUndone	Undo has already been processed
glsErr.assigningReservations	Not enough counts to assign reservations.
glsErr.backOfficePollingInProgress	Server is busy (e.g. updating license rights from the back office or processing reservations).
glsErr.badPassword	Invalid password: {0}.
glsErr.bothUuidAndHostId	Cannot specify both UUID and hostid for client identification
glsErr.checkoutSyntaxError	Syntax error while compiling checkout script:{0}
glsErr.clientNotFound	Client not found
glsErr.configItemNotEditable	The configuration setting, {0}, is not editable by the {1} role at the {2} scope.
glsErr.configSetInstanceMixed	Cannot specify any other set identifiers with instanceId

Table 4-14 ▪ License Server Errors (cont.)

Exception	Definition
glsErr.configSetSiteRequires	Must specify at least publisher ID and enterprise ID if siteName specified
glsErr.configSetTenantRequired	Must specify at least tenant ID if enterprise ID specified
glsErr.connectionFailed	Connection to {0} failed with exception: {1}
glsErr.connectionFailedStatus	Connection to {0} failed with status {1}
glsErr.connectionRefused	Connection to {0} has been refused
glsErr.duplicateCorrelationId	Duplicate correlation ID in non-undo operation
glsErr.emptyMessage	Empty message received
glsErr.enterpriseNotFound	Enterprise not found: {0}
glsErr.featureConversion	Feature conversion failed
glsErr.featureLoading	Feature loading failed
glsErr.featureNotFound	Feature not found
glsErr.generatingUuid	Problem generating UUID for client
glsErr.hostidNotFound	Specified hostid does not exist: {0}
glsErr.hostIdTypeWithoutHostId	Cannot specify hostid type without hostid value
glsErr.identityMappingFailed	Identity mapping failed
glsErr.infoMessageNotSupported	This version of license server does not support information messages; use Capability Request with Report operation instead
glsErr.internalSigningError	Internal signing error
glsErr.invalidModelName	Invalid model name supplied (the model definition names `default` and `reservations` are reserved names)
glsErr.invalidReservationRequest	Invalid reservation request.
glsErr.invalidUUID	Invalid UUID value
glsErr.jsonLicensingSecurityNoToken	Valid token required for secure JSON licensing request.
glsErr.JsonValidationError	JSON fails endpoint validation.

Table 4-14 ▪ License Server Errors (cont.)

Exception	Definition
glsErr.lfsError	License Fulfillment Service returned error: {0}
glsErr.lfsCapSessionFailed	LicenseFulfillmentService capability session failed
glsErr.lfsWrongMessageType	License Fulfillment Service returned an unexpected message type: {0}
glsErr.licenseFailsAnchor	License response fails anchor criteria
glsErr.maintenanceIntervalPassed	Server maintenance interval has passed.
glsErr.missingServerIdentity	Instance has no identity data
glsErr.mixingHostIds	Cannot use a main hostid as secondary hostid or vice-versa
glsErr.modelSyntaxError	The uploaded model definition contains a syntax error
glsErr.modelUpdateInProgress	Server is busy updating the model definition
glsErr.needUuidOrHostId	Must specify either UUID or hostid for client identification
glsErr.noAccessToPost	Unable to access POST data
glsErr.noFeatures	No features found in offline capability response
glsErr.noQueryHandler	No query handler installed for {0}
glsErr.noSigner	No signer was available
glsErr.noSingleServer	Single server instance not found
glsErr.onlyClients	Capability requests are only expected from clients, not servers
glsErr.onlyTrustedClient	Only trusted clients are allowed for this endpoint
glsErr.onlyUntrustedClient	Only untrusted clients are allowed for this endpoint
glsErr.optimisticLocking	Too many optimistic lock failures. Retry later
glsErr.pageNotFound	That page does not exist
glsErr.parsingBinary	Binary message could not be parsed
glsErr.parsingServerIdentity	Unable to parse server identity data
glsErr.partitionsNotFound	Requested license pool does not exist

Table 4-14 • License Server Errors (cont.)

Exception	Definition
glsErr.passwordNotSpecified	A password was not specified.
glsErr.queryParamIncompatible	Incompatible query arguments.
glsErr.queryParamNeedsInt	Value for query parameter {0} should be an integer
glsErr.queryParamTooLarge	Value for query parameter "{0}" cannot be more than {1}
glsErr.queryParamTooSmall	Value for query parameter "{0}" cannot be less than {1}.
glsErr.queryParamUnsupported	Query parameter(s) not supported: {0}
glsErr.readingServerIdentity	Error reading binary identity data
glsErr.reqHostIdMismatch	Request has hostid: {0}, expected: {1}
glsErr.reqHostUuidMismatch	Request has host UUID: {0}, expected: {1}
glsErr.reqHostUuidMissing	Request missing host UUID:, expected: {0}
glsErr.reqIdentityMismatch	Identity mismatch in binary request
glsErr.reqUnsupportedHostIdType	The server cannot process requests from clients with the hostid type {0}
glsErr.reservationAlreadyExists	Reservation already exists: {0}
glsErr.reservationNotFound	Reservation not found: {0}
glsErr.responseBadSignature	Capability response is not signed correctly
glsErr.responseDupFeatureId	Capability response rejected: the response has features with duplicate feature ID: {0}
glsErr.responseEnterpriseMismatch	Capability response rejected: the response enterprise ID does not match the server instance
glsErr.responseExpired	Capability response rejected: response time too old
glsErr.responseHostidMismatch	Capability response hostid does not match. Expected {0}, got {1}
glsErr.responseNoHostid	Capability response must have at least one server hostid
glsErr.responseNotSingleHostid	Capability response must have exactly one server hostid
glsErr.responseStale	Capability response rejected: update time invalid - a later response has been processed

Table 4-14 ▪ License Server Errors (cont.)

Exception	Definition
glsErr.responseUuidMismatch	Capability response rejected: the response host UUID does not match the server instance
glsErr.restNoSuchApi	No such REST API and METHOD combination supported: {0} with {1}
glsErr.restParsing	REST request parsing problem: {0}
glsErr.restUnimplemented	REST service not implemented
glsErr.rightsCopiesInvalid	The rights ID copies must be non-negative: {0}
glsErr.rightsIdMissing	A rights ID must be specified
glsErr.rightsOnlyRequestOp	Rights IDs are only allowed for request operations
glsErr.securityGroupNotFound	Security group not found: {0}
glsErr.serializingSync	Serialization/signing error in sync message generation for instance {0}
glsErr.serverNoHostid	Server instance has no hostids available
glsErr.serverNoId	Server instance ID is missing
glsErr.serverNoIdentity	Server instance identity is missing
glsErr.serverNotFound	Server instance not found: {0}
glsErr.serverNotReady	Server instance {0} is not ready
glsErr.serverNoVm	License server not configured to allow running it on a virtual machine
glsErr.serverPendingDeletion	Server instance {0} is unavailable, pending deletion
glsErr.serverSuspended	Server instance {0} is suspended
glsErr.serviceBusyDueToReservations	Service is busy processing reservations.
glsErr.signatureInvalid	Signature of received binary message was not valid
glsErr.soapFailure	SOAP service {0} failed
glsErr.sslMisconfigured	SSL is misconfigured: {0}
glsErr.syncDisabled	Offline sync cannot be performed. Sync is disabled.

Table 4-14 ▪ License Server Errors (cont.)

Exception	Definition
glsErr.syncInProgress	Offline sync cannot be performed. Instance busy.
glsErr.tooManyFailedLogins	Too many failed logins for user {0} on instance {1}. Logins will be blocked for {2} minutes.
glsErr.unexpectedMessageType	Message of type {0} was not expected and cannot be processed
glsErr.unknownServer	Unknown server instance {0}
glsErr.userAccessDenied	Access denied to {0} for user {1}.
glsErr.userAlreadyExists	User already exists: {0}.
glsErr.userAuthFailed	Authorization attempt at {0} failed for user {1} (error {2}).
glsErr.userDoesNotExist	User cannot be found: {0}.
glsErr.userMismatch	That user record has a different name to the one supplied.
glsErr.userNotSpecified	A user name was not specified.
glsErr.userTokenExpired	Authorization attempt at {0} failed due to token expiry for user {1}.
glsErr.wrongFailoverRole	Wrong failover role: {0}
glsErr.wrongMessageType	Message of type {0} expected, but {1} found
glsErr.wrongServerIdentityType	Supplied identity data variant {0} is not the expected variant of {1}

5

More About Basic License Server Functionality

This chapter provides background information about functionality that you can enable for your FlexNet Embedded license server and that might require additional setup performed by you or the license server administrator:

- Synchronization Operations (applicable to local license servers only)
- Allocating Licenses Using Named License Pools (applicable to local license servers and CLS license servers)
- License Reservations (applicable to local license servers and CLS license servers)

The following is information to keep in mind when referring to this chapter.

“licenseServer_baseURL” Used in Commands

Some the commands referenced in this chapter use the license server’s base URL, indicated by *licenseServer_baseURL* in the command. For more information about this URL, see Base URLs in Chapter 4, License Server REST APIs.

Authorization Credentials for Administrative Security

If administrative security is enabled for the license server, the license server administrator might need to provide authorization credentials to perform certain operations described in this chapter. For more information about administrative security and defining authorization credentials, see Administrative Security in Chapter 6, Advanced License Server Features.

Synchronization Operations

Server synchronization is functionality that enables a FlexNet Embedded license server to synchronize its license-distribution or metered-usage information with the back office (FlexNet Operations) or its license-distribution information with another license server in a failover scenario (for local license servers only). Server synchronization enables you to offer new business models based on usage of the client capabilities and capacity, recover from catastrophic server failure, and provide enhanced support and upgrade processes.

Server synchronization can be configured in the following ways:

- **Synchronization to the back office**: In this configuration, the license server periodically sends information about the current state of license distribution or metered usage to the back office. Information about clients that have been served licenses will then be visible in the back office and usage-capture information is properly accounted for in the metered-license models. An offline synchronization process to the back office is also supported.
- **Synchronization to and from the back office**: In this configuration, in addition to sending license-distribution and metered-usage information to the back office, the license server can recover from catastrophic failure by synchronizing its state back from the data stored in the back office.
- **Synchronization to another FlexNet Embedded local license server**: (Local license servers only) In this configuration, the current FlexNet Embedded local license server sends its license-distribution data to another local license server, enabling server failover should the main license server fail or be shut down. This configuration is independent of synchronization with the back office.

The following sections describe how to enable and configure the different types of server synchronization:

- Online Synchronization to the Back Office
- Offline Synchronization to the Back Office
- Synchronization From the Back Office
- Failover Using Synchronization to FlexNet Embedded

The section Notes about Policy Settings for Synchronization provides references for more information about the `producer-settings.xml` file.

Notes about Policy Settings for Synchronization

The various types of synchronization processes described in the next sections are controlled by specific policy settings, as defined for the license server in the `producer-settings.xml` file, which you generate and provide with the license server.

Use the following references for more information about the policy settings in the `producer-settings.xml` file:

- For a description of the settings specified in these sections, see Appendix A, Reference: Policy Settings for the License Server.
- For instructions on generating the `producer-settings.xml` file, see License Server Configuration Utility in Chapter 7, Producer Tools.
- Some of these settings are editable by the license server administrator at the customer site. To determine which settings the administrator can edit, see Appendix A, Reference: Policy Settings for the License Server.

  Using the `/configuration` REST endpoint, the administrator can display and edit these settings (see APIs to Manage Instance Configuration in Chapter 4, License Server REST APIs). Additionally, the license server administration tools that are included in your license server software package—and that you can make available to the customer—provide facilities to view and update these settings. See the *FlexNet Embedded License Server Administration Guide* for details.

Online Synchronization to the Back Office

When synchronization to the back office is configured, the FlexNet Embedded local license server automatically sends a synchronization message to the back office on a periodic basis through an active internet connection with the back office. The type of data submitted in the synchronization message depends on the value specified for the policy setting `lfs.syncTo.IncludeAll` (see Table 5-1 for more information). When the back office receives and processes the synchronization message, it sends back a synchronization acknowledgment. The synchronization acknowledgment contains status information and any error information regarding the processing. Additionally, the server log records the event, along with any related errors or warnings.

After a successful synchronization session, information about client devices that have been served licenses is accessible to the back office. Consult your FlexNet Operations documentation for details.

Note ▪ *This synchronization process described here is "online"—that is, it uses direct internet access as a means of transmitting data to and from the back office. As an alternative to this type of data transmission, the FlexNet Embedded local license server supports an offline synchronization process that does not require the server to have direct internet access. See Offline Synchronization to the Back Office.*

Configuring Online Synchronization

The following policy settings in the `producer-settings.xml` file control scheduled online synchronization to the back office. For information about additional settings that might be available, see Appendix A, Reference: Policy Settings for the License Server.

Table 5-1 ▪ Policy Settings for Online Synchronization

Policy Setting	Description
`lfs.url`	The URL for the back office to which the license server sends the synchronization messages.
`lfs.syncTo.enabled`	Determines whether synchronization to the back office is enabled. Must be set to `true` for scheduled synchronization.
`lfs.syncTo.IncludeAll`	Determines whether historical data is collected and sent to the back office as part of the synchronization. If set to `false`: This mode collects only the current state for each active client device at the point of synchronization and uploads this data to the back office. If set to `true`: This mode collects all historical client actions (license distribution and metered usage on client devices, based on the server's deduction records) in the synchronization history and uploads this data to the back office as part of the synchronization.
`lfs.syncTo.repeats`	The amount of time between synchronization sessions to the back office. The value can be specified with an optional unit-suffix letter—`s`, `m`, `h`, `d`, or `w`—indicating seconds, minutes, hours, days, or weeks. If no suffix is used, the server assumes the value is in seconds. (Default is `5m`; minimum is `10s`.)

On-Demand Synchronization

The normal synchronization process to the back office is run automatically at a set interval. However, situations might arise where when the license server administrator needs to trigger an online synchronization before the next scheduled one. For example, the server might be accumulating an unusually large amount of metered-usage data or the enterprise is anticipating a high volume of network traffic during of the next scheduled synchronization. Running an extra synchronization might be a way to avoid problems getting license data sent to the back office.

Using the `/sync_message` REST endpoint, the administrator can run a synchronization to the back office at any time without altering the normal synchronization schedule. See Generating an Offline Sync Message in Chapter 4, License Server REST APIs.

Configuring On-Demand Synchronization

The following policy settings in the `producer-settings.xml` file control on-demand synchronization to the back office. For information about additional settings that might be available, see Appendix A, Reference: Policy Settings for the License Server.

Table 5-2 ▪ Policy Settings for On-Demand Synchronization

Policy Setting	Description
`lfs.url`	The URL for the back office to which the license server sends the synchronization messages.
`lfs.syncTo.IncludeAll`	Determines whether historical data is collected and sent to the back office as part of the synchronization. If set to **`false`**: This mode collects only the current state for each active client device at the point of synchronization and uploads this data to the back office. If set to **`true`**: This mode collects all historical client actions (license distribution and metered usage on client devices, based on the server's deduction records) in the synchronization history and uploads this data to the back office as part of the synchronization.

Offline Synchronization to the Back Office

The traditional process of synchronizing license data from the license server to the back office relies on the license server having direct internet access to transmit data to the back office. For security reasons, such as data privacy, license servers might have limited or no external network connections, making it difficult to synchronize data about concurrent-license distribution and metered usage on the server to the back office. The offline synchronization feature supports the ability to download this data from the license server on to another device that has an external network connection and then transmitting the data to the back office.

The following sections describe offline synchronization to the back office:

- Configuring Offline Synchronization
- Tools

- Offline Synchronization Process
- Keeping Enterprises in Compliance When Offline Synchronization Is Used

Configuring Offline Synchronization

The following policy settings in the `producer-settings.xml` file control offline synchronization to the back office. For information about additional settings that might be available, see Appendix A, Reference: Policy Settings for the License Server.

Table 5-3 ▪ Policy Settings for Offline Synchronization

Policy Setting	Description
`lfs.url`	The URL for the back office to which the license server sends the synchronization messages.
`lfs.syncTo.enabled`	Determines whether synchronization to the back office is enabled. Must be set to **`false`** for offline synchronization.
`lfs.syncTo.IncludeAll`	Determines whether historical data is collected. Must be set to **`true`**: All historical and current client actions (license distribution and metered usage on client devices, based on the server's deduction records) are collected. This data is retained on the license server until the offline synchronization tools are run (see Tools).

Tools

The license server provides these two tools to perform the offline synchronization operations. The tools can be found in the `enterprise` directory of the FlexNet Embedded local license server installation directory.

- "serverofflinesynctool.bat" (or "serverofflinesynctool.sh")
- "backofficeofflinesynctool.bat" (or "backofficeofflinesynctool.sh")

"serverofflinesynctool.bat" (or "serverofflinesynctool.sh")

This utility downloads the client records to be synced to the back office to a temporary storage location on (or accessible by) the device that has direct internet access to the back office. It is also used to upload and process the synchronization acknowledgment sent by the back office, purge old downloaded records, and force a download of all synchronization records when necessary.

If administrative security is enabled for the license server, the utility command (shown in the steps that follow) must include an `-authorize` option to specify the credentials needed to run the utility, as shown:

```
serverofflinesynctool.bat -authorize name password -url http://LicenseServerHostName:port/api/1.0/
    ...
```

***Note** ▪ (Linux only) If you supply a password directly on the command line you may need to escape some characters with a backslash in order to prevent the shell from interpreting them. The safest approach is to surround the whole password with single quotes, and backslash any single quote that is part of the password.*

For more information about administrative security and setting up authorization credentials, see Administrative Security in Chapter 6, Advanced License Server Features.

"backofficeofflinesynctool.bat" (or "backofficeofflinesynctool.sh")

This utility uploads the client records to be synchronized from the temporary location to the back office. It also receives and saves the synchronization acknowledgment received from back office.

Offline Synchronization Process

The following describes how to perform the offline synchronization process using `serverofflinesynctool` and `backofficeofflinesynctool`.

Step 1: Download Records

The first step is to download the transaction records to be synchronized from the license server using the `serverofflinesynctool.bat` (in Windows) or `serverofflinesynctool.sh` (in Linux) tool:

```
serverofflinesynctool.bat -url http://licenseServerHostName:port/api/1.0/sync_message/offline
    -generate path
```

where:

- *`licenseServerHostName:port`* is the server name and port for the license server URL from where the records are being downloaded (default port is 7070)
- *`path`* is the path on the local machine where the records will be temporarily stored

Additionally, if administrative security is enabled, you must include your authorization credentials in the command. See "serverofflinesynctool.bat" (or "serverofflinesynctool.sh") for details.

This command can be run on a system with an external network connection or on the hosting license server. If you run the command on the hosting license server, the files containing the data to be synchronized need to be copied to a machine with external network connectivity. Once the download completes, a message stating the number of transaction records downloaded is displayed:

```
OfflineSync utility started.
Sync completed for 3 device records.
```

If there are no new transaction records to download, the message displays the following:

```
OfflineSync utility started.
No new data is available.
```

Step 2: Synchronize to the Back Office

Next, use the `backofficeofflinesynctool.bat` (in Windows) or `backofficeofflinesynctool.sh` (in Linux) tool to synchronize the data to the back office:

```
backofficeofflinesynctool.bat -url https://siteID.flexnetoperations.com/flexnet/deviceservices -out
    filename path
```

where:

- *siteID* is your organization's DNS name or the specific site ID supplied by Revenera (see More About the Base URL).
- *filename* is the name of the file to which the synchronization acknowledgment will be written to (for example, `sync_ack.bin`).
- *path* is the path on the local machine where the transaction records to be synchronized were stored by the `serverofflinesync` tool. To specify a specific file, provide the path and file name.

A sync acknowledgment message is returned:

```
Successfully sent sync data and received a sync acknowledgment.
```

The sync acknowledgment received from the back office is written to the output file (default is `syncack.bin` in the current directory).

```
MessageType="Server sync acknowledgment"
MessageTIme="Jan 13, 2019 10:53:46 AM"
LastSyncTime="Jan 13, 2019 10:52:45 AM"
SourceIDType=String
SourceID=BACK_OFFICE
SourceIds=BACK_OFFICE
TargetID=A088B436F208
TargetIDType=Ethernet
```

Step 3: Update the Synchronization Time

The synchronization acknowledgment needs to be processed on the license server to update the last time of synchronization so that it knows that the data has been synchronized to the back office. When executing the tool, use the same *path* value used to download and synchronize the data to the back office so that the tool can remove the old synchronized data file after it processes the response. (Additionally, if administrative security is enabled, you must include your authorization credentials in the command, as described in "serverofflinesynctool.bat" (or "serverofflinesynctool.sh").)

```
serverofflinesynctool.bat -url http://licenseServerHostName:port/api/1.0/sync_ack/offline
    -process filename path
```

The server responds with the following message:

```
OfflineSync utility started.
Purging file 20140613T105312.fnesync
```

Force a Download of All Synchronization Records

The `-force` option can be used with the `serverofflinesync` tool if you encounter an error similar to the following:

```
"Mismatched or out of order time stamp in sync message": "Expected sync time Jan 24, 2019 11:04:09
AM, got Jan 24, 2019 11:05:00 AM")
```

Checking the synchronized records in the back office reveals that those records processed after a mismatch are not synchronized. To correct the problem, you must force a download of all transaction records to account for any records missed during synchronization. The `-force` option allows you to start the record collection and download from the last successful synchronization time.

Task ***To force a download of all records since the last time of synchronization***

1. Process the synchronization acknowledgment on the license server (as described in Step 3: Update the Synchronization Time). This will update the records up to the point of the mismatch. Only those records that were synchronized to the back office are deleted.
2. Recover the records missing from synchronization. However, if you try to download the records again, the server thinks it has already downloaded all the records and responds with `No new data is available`. To remedy the problem, use the `-force` option with **serverofflinesynctool**. (If administrative security is enabled, you must also include your authorization credentials in the command, as described in “serverofflinesynctool.bat” (or “serverofflinesynctool.sh”).)

   ```
   serverofflinesynctool.bat -url http://licenseServerHostName:port/api/1.0/sync_message/offline
       -generate path -force
   ```

   This will dump *all* the records since the `lastSyncTime` specified in the last synchronization acknowledgment.
3. Run the `backofficeofflinesynctool` utility to upload the data to the back office.
4. Once the data has been successfully synchronized, process the acknowledgment on the server to update the synchronization date and delete the synchronized data.

Keeping Enterprises in Compliance When Offline Synchronization Is Used

When FlexNet Embedded local license servers use the offline synchronization to send data to the back office, the producer’s ability to enforce timely synchronization can be difficult. You must rely on the enterprise to run an offline synchronization at scheduled times to keep license-distribution and meter-usage information on the back office up to date. A missed synchronization can cause an enterprise to be out-of-compliance with your business policies or, in the case of metered usage, can result in an enterprise exceeding their usage limit for a given time period.

The license server provides ways to monitor the current synchronization status at an enterprise and take appropriate action when synchronization is out of compliance.

Synchronization Status Reports

You can use the following two methods to obtain the current synchronization status for an enterprise customer. Each method reports the following information:

- Last time metered usage was synchronized to the back office
- Total number of transaction records in trusted storage that have not been synchronized

Run the “status” Command in the FlexNet License Server Administrator Command-line Tool

Use the FlexNet License Server Administrator command-line tool (included in the `enterprise` directory in your license-server software package) to run the `-status` command, as shown in this example:

```
flexnetlsadmin.bat -server licenseServer_baseURL -status
```

The output includes the synchronization status, as highlighted in this sample excerpt. (In this case, 42 records are pending synchronization, and the last synchronization occurred 7 days, 4 hours, 38 minutes and 28 seconds ago.)

```
Version              : 2020.01.0
Build Version        : 262400
Server               : https://localhost:7070/api/1.0/instances/~
State                : Ready
Backup Server        : Not configured
BackOffice Server    : http://localhost:8080/request
Records Pending Sync : 42
Last Sync            : 7d 4h38m28s
```

Note ▪ *The time format for the “Last Sync” value is* `nYnMnD nHnMnS`*, where Y=years, M=months, D=days, H=hours, M=minutes, and S=seconds. If any leading elements are missing, they are assumed to have a value of zero. If a sync has never been performed, "sync to back office pending" is displayed.*

See the “Using the FlexNet License Server Administrator Command-line Tool” chapter in the *FlexNet Embedded License Server Administration Guide* for details about the `-status` command.

Use the “instances/*instanceID*” REST API

Use the `/instances/instanceID` REST API with `syncinfo=true` query argument to display the properties `recordsPendingSync` and `lastSyncTime` (listed in Greenwich Mean Time), which report the server’s synchronization status:

```
http://licenseServer_baseURL?syncinfo
```

The following shows a sample response body:

```
{
  "id": 1,
  "maintenanceEndTime": "1970-01-01T00:00:00.000Z",
  "suspended": false,
   ...
  "publisherName": "demo",
  "failOverRole": "MAIN",
  "lastSyncTime": "1970-01-01T00:00:00.000Z",
  "lastUpdateTime": "2018-08-24T15:52:20.000Z",
  "lastUpdateHostid": {
    "hostidValue": "7C7A77BA777B",
    "hostidType": "ETHERNET"
  },
  "identityName": "demo-med-rsa",
  "recordsPendingSync": 2
}
```

For more information about this REST API, see Chapter 4, License Server REST APIs.

Disabling Synchronization to the Back Office

To disable synchronization to the back office, the `lfs.syncTo.enabled` and `lfs.syncTo.IncludeAll` settings in `producer-settings.xml` must both be set to `false`. In this scenario, no synchronization data is collected. Client data is deleted from the license server as soon as the client expires.

Synchronization From the Back Office

Another type of synchronization is synchronization of license data *from* a back-office server. Synchronization from a back office enables a FlexNet Embedded local license server's distribution state to be restored from the back office's distribution information after catastrophic server failure. (This configuration thus requires synchronization *to* the back office also to be enabled prior to the server failure.)

The following sections describe synchronization from the back office:

- Configuring Synchronization From the Back Office
- Synchronization Process

Configuring Synchronization From the Back Office

The following policy settings in the `producer-settings.xml` file control the synchronization from the back office. For information about additional settings that might be available, see Appendix A, Reference: Policy Settings for the License Server.

Table 5-4 ▪ Policy Settings for Scheduled Synchronization

Policy Setting	Description
`lfs.syncFrom.enabled`	Determines whether license-recovery from the back office is enabled. If recovery is enabled, the metered-usage data and license-distribution state for concurrent features is recovered from the back office when the license server initially starts up with a new or reset trusted storage. Must be set to `true`.

Synchronization Process

Once the license server is up and running, if trusted storage is empty, the server first sends a capability request to the back office to obtain its pool of licenses. The server then sends a *synchronization request* message to the back office to obtain the server's license-distribution and metered-usage data required to restore the deduction records on the server. (If synchronization from the back office is enabled, the server log contains an entry stating as such.)

Once the synchronization from the back office is complete, the server can start responding to capability requests from client devices. (Licensing data for active clients is once again visible using the `/clients` REST endpoint, described in APIs to Manage Clients in Chapter 4, License Server REST APIs.) Moreover, transaction records will be reflected in the back office with updated timestamps the next time synchronization to the back office is performed. These updated records indicate successful restore of the server's license-distribution and metered-usage state.

Failover Using Synchronization to FlexNet Embedded

The FlexNet Embedded local license server includes support for *license server failover*. Failover involves two local license servers—the *main license server* and the *back-up license server*—that work together to ensure that licenses in the enterprise remain available for serving to client devices should the main server fail.

The two license servers in a failover configuration must be of the same version.

In a failover configuration, the back-up license server periodically contacts the main FlexNet Embedded local license server, requesting that the main license server send its current client-usage information. The back-up license server then applies this information to mirror the state of the main server, thus synchronizing the two servers.

If a client device determines that the main license server has failed—by a failure to respond to the client's capability request, for example—it can communicate with the back-up license server, which then takes over responsibility for serving the pool of licenses until the main license server is restored, or until the configured maintenance interval has elapsed. You, as the software producer, configure this maintenance interval using the `server.backupMaintenance.interval` setting in the `producer-settings.xml` file.

The back-up license server does not synchronize to the back office or to the main license server. Therefore, data about clients served by the back-up license server during a failover period is not stored in the back office. However, once the main license server has come back online, license-distribution data collected from that point on is once again synchronized to the back office.

The following sections describe license server failover:

- Configuring License Server Failover
- Additional Failover Considerations

Configuring License Server Failover

The following process describes the responsibilities that you, as software producer, and your enterprise customer's license server administrator need to perform to enable support for license server failover. (Certain steps in this process describe defining specific license-server policy settings. See Notes about Policy Settings for Synchronization for information about synchronization settings in general.)

Step 1: Provide the Producer Settings File

You provide a `producer-settings.xml` file that defines the `server.backupMaintenance.interval` setting to define how long the back-up license server can serve licenses without the main license server being online. This file is installed with both license servers, but the setting is used by only the back-up license server. (Optionally, you can provide two separate files so that this setting is defined only in the file to be installed on the back-up license server.)

If necessary, also update the `fne.syncTo.repeats` license server policy using the **/configuration** API to set the synchronization interval between the back-up license server and the main license server. (The license server administrator does not have permission to edit this setting.) See APIs to Manage Instance Configuration.

Step 2: Install and Configure the License Servers

The license server administrator installs both the main and back-up license servers. As part of the installation process, the administrator configures both license servers with basically the same local settings, with some allowable differences. For example, the administrator can specify different `PORT` values for the license servers. Additionally, if specifying the `BACKUP_SERVER_HOSTID` setting, the administrator does so only on the main license server (see (Optional) Automatic Registration of the Failover Pair).

For more information about installing a license server and editing its local settings file—`flexnetls.settings` on Windows or `/etc/default/flexnetls-`*`producer_name`* on Linux, see Chapter 3, Installing and Running the License Server.

Step 3: Register the Main and Backup License Servers

Using the FlexNet Operations End-User Portal, the license server administrator registers the main license server with the back-up license server as a failover pair on the back-office server. (This registration is performed by specifying the hostid of the back-up license server on the portal's Create Server page for the main license server.) The hostid for each the main and the back-up license server must be of the same type.

Note **▪** *It may be necessary to enable support for server failover in FlexNet Operations.*

For more information about registering the failover pair from the Create Server page for the main license, the administrator can locate instructions in the online help system for the FlexNet Operations End-User Portal. (As producer, you can also navigate to **Manage Hosts | Add License Server** in the Producer Portal to register a failover pair.)

This step can be replaced with an automatic registration of the failover pair through the capability request sent from the main license server. See (Optional) Automatic Registration of the Failover Pair for details.

Step 4: Set Up an Entitlement for the License Servers

You need to set up an entitlement of the license rights for the main license server. If you map these rights to the failover pair through the main license server, both license servers automatically receive the entitled features when they start up. If you do not map the rights to the main license server, the license server administrator can use rights IDs to activate the licenses once the license servers are running.

Step 6: Start Up the License Servers

The license server administrator starts up the license servers. Any license rights already mapped to the failover pair through the main license server are automatically activated on the both license servers. If the administrator needs to activate licenses manually, see Step 9: Activate Licenses on the Servers.

The administrator can now use the administrator tool provided with the license server to update policy settings, as described next in Steps 7 and 8.

Step 7: Enable Failover Support on the Back-Up Server

The license server administrator makes changes to policy settings on the back-up license server, using the license-server administrator tool you provide, such as the FlexNet License Server Administrator command-line tool or your own administrator interface or tool:

1. Sets the following license server policy settings on the back-up server:
   - `fne.syncTo.mainUri` - Set to the URI of the main license server (in the format `http://`*`licenseServerHostName`*`:`*`port`*`/fne/bin/capability`).
   - `fne.syncTo.enabled` - Set to `true`.
2. (Optional) Sets other failover-related policy settings, such as other `fne.syncTo.*` settings or the `licensing.backup.uri` and the `licensing.main.uri` settings. (The format `http://`*`licenseServerHostName`*`:`*`port`*`/fne/bin/capability` must be used for the URI settings.)

For information about these policy settings, see Appendix A, Reference: Policy Settings for the License Server.

(Optional) Step 8: Edit Producer Settings on the Main Server

The license server administrator can configure one or both of the following license server policy settings on the main license server to include these URIs as reference information sent in capability responses to client devices. (The format `http://`*`server`*`:`*`port`*`/fne/bin/capability` must be used for the URIs.)

- `licensing.backup.uri`
- `licensing.main.uri`

For information about these policy settings, see Appendix A, Reference: Policy Settings for the License Server.

Step 9: Activate Licenses on the Servers

If no license rights were activated on the license servers at startup, the license server administrator performs these steps to provision the servers with licenses:

1. Starts up both license servers.
2. Sends one or more rights ID in a capability request to activate identical license rights on both license servers via the current (main) license server. (You need to let the administrator know which rights IDs are available to the license server.)

Step 10: Verify Failover Roles

The license server administrator can verify the failover role of each license server after the servers receive their license rights. (The failover roles are determined by the back office and included in the capability response.)

Verification can also be obtained using the FlexNet License Server Administrator command-line tool (for example, **`flexnetlsadmin -server`** ***`licenseServer_baseURL`*** **`-status`**). Additionally, the primary server log will state the failover role of the given license server.

(Optional) Automatic Registration of the Failover Pair

As described in Step 3: Register the Main and Backup License Servers, the license server administrator must register the main and back-up license servers as a failover pair in FlexNet Operations. One way to perform this registration is to do so manually through the FlexNet Operations End-User Portal.

Another way to perform this registration is to use the capability request sent to FlexNet Operations by the main license server at startup as a means of conveying the back-up license server's hostid to the back-office server. FlexNet Operations then uses the hostid information in the capability request to automatically register both license servers as a failover pair, thus avoiding the extra step of having to access the End-User Portal to perform the registration.

To set up this automatic registration process, the license server administrator does the following:

1. Shuts down both the main and back-up license servers.
2. On the main license server, updates the local license-server settings to include the `BACKUP_SERVER_HOSTID` setting. This value identifies the hostid of the back-up license server. It must be the same hostid type (such as "ETHERNET") as the type of hostid used by the main (current) license server. See Chapter 3, Installing and Running the License Server for details about updating local settings.

3. Starts up the main license server first.

   The main license server sends a capability request containing the back-up license server's hostid to FlexNet Operations, where both servers are then registered as a failover pair.

4. Starts up the back-up license server once the main license server has successfully communicated with FlexNet Operations. Both servers can now obtain license rights either through rights IDs sent in a capability request or through capability polling if licenses rights are subsequently mapped in FlexNet Operations to the registered failover pair.

   Note ▪ *The license server administrator should wait until the main license server has successfully started and communicated with FlexNet Operations before starting the back-up license server. Otherwise, FlexNet Operations might assign the back-up license server to the "main server" role, generating an error when FlexNet Operations then attempts to register the actual main license server.*

5. Configures the license servers as described in Step 7: Enable Failover Support on the Back-Up Server and (Optional) Step 8: Edit Producer Settings on the Main Server.

The license server administrator can view registration details for the main and back-up server by searching for the main license server (from the Search Servers page) in the FlexNet Operations End-User Portal. The View Server page for the main license server shows both license servers registered as a failover pair.

Additional Failover Considerations

The following lists additional considerations when enabling failover configuration:

- Since the back-up license server does not synchronize to the back office or the main license server, data about clients served by the back-up server during a failover period is permanently missing in the back office. Once the main server has come back online, the new license-distribution data can be synchronized to the back office.
- Metered-usage data is currently not supported in a failover scenario.
- License server failover has limited compatibility with JSON security in REST-driven licensing, mainly because the private and public keys required for this type of security are not part of the synchronization process from the main server to the backup server.

  For specific information about failover compatibility with JSON security, see Online REST API Reference Documentation Available. For more information about JSON security in REST-driven licensing in general, see API for Enabling JSON Security for the Cloud Monetization API in Chapter 4, License Server REST APIs.
- Failover functionality is dependent on the clocks of the main license server and back-up license server being accurate, synchronized with each other, and in the same time zone. If any of these requirements are not met, unpredictable behavior can occur.
- The failover configuration does not support reservations.
- The failover configuration does not support named license pools. During a failover period, feature counts will only be distributed from the default license pool.

Allocating Licenses Using Named License Pools

***Important** ▪ License server administrators can use either reservations or named license pools for license allocation. Reservations and named license pools cannot coexist alongside each other. If currently no license reservations are defined for a license server, review the information in Comparison of Named License Pools and Reservations to decide which approach is best for you. In general, reservations are conceptually similar to named license pools, but named license pools offer significantly more flexibility in controlling a server's license estate. See also Named License Pools vs. Reservations.*

***Note** ▪ In previous releases of the FlexNet Embedded license server, named license pools used to be referred to as "partitions".*

In a generic setup, each license server has a license pool which contains all the licenses available to its client devices. The license server distributes licenses from this license pool on a first-come-first-served basis to client devices requesting them. This license pool is also referred to as the *default license pool*.

However, in addition to the server's default license pool the license server administrator can create *named license pools*, which divide an organization's license estate into groups of licenses. By defining rules of access, administrators can allocate licenses to a group of client devices or users to help ensure that these entities have access to the features they need.

Thus, a license server serves licenses from its default license pool and any named license pools that the license server administrator creates. Each named license pool contains license counts for one or more features, which can be allocated to client devices or users. It is important to note that in a named license pool, licenses are not assigned to individual client devices or users. Instead, counts allocated to a named license pool can be accessed by a number of devices or users that meet certain criteria, such as, for example, be member of a particular business unit or have a certain hostid. Allocated license counts can be checked out by any device or user that fulfills the specified criteria.

The license server administrator can choose to allocate the entire pool of licenses, a portion of the licenses, or none of the licenses. Any license that is not allocated remains in the default license pool and is available to any device on a first-come-first-served basis.

The following sections provide information about named license pools on the FlexNet Embedded license server:

- Named License Pool Terminology
- Introduction to the Model Definition
- Model Definition Components
- Server Behavior When Distributing Feature Counts to Named License Pools
- Server Behavior when Assigning Features to Clients
- Named License Pools vs. Reservations
- Limitations of Named License Pools

Named License Pool Terminology

The following table provides brief definitions for the terminology used around the named license pool functionality:

Table 5-5 ▪ Named License Pool Terminology

Name	Definition
License Pool	When no named license pools have been defined, “license pool” means the license estate from which the license server serves licenses to its client devices. In setups where named license pools have been defined, the term “license pool” is used to refer to a nondescript license pool, that is, it can refer to a *default license pool* and/or *named license pool*.
Default License Pool	Every server has a default license pool which cannot be deleted. If no *named license pools* have been defined, the *default license pool* holds all licenses on the server. If one or more named license pools have been defined, the default license pool holds any licenses that have not been allocated to a named license pool.
Named License Pool	A pool that holds any licenses that have been allocated to it by rules of access. A named license pool is separate from the default license pool.
Model Definition	The model definition specifies the license pools and the rules of access that define how licenses are allocated to license pools (named or default).
Rules of Access	Rules of access define the criteria that allow or block certain client devices or users from obtaining licenses from specified license pools (named or default).
Conditions	Criteria used in rules of access that allocate licenses to license pools (named or default).

Introduction to the Model Definition

Named license pools are defined using a *model definition*. The model definition is configured and uploaded per local license server instance or Cloud Licensing Service (CLS) instance. The license server administrator can either use the REST API endpoints or the FlexNet License Server Administrator command-line tool to view or delete named license pools and license count allocations on the license server.

A model definition specifies the following:

- **License pools**—Named license pools are optional. If a model definition does not include any named license pools, all counts are placed in the default license pool, and rules can only allow or deny access to the default license pool.

If a model definition contains named license pools, each named license pool will hold one or more features. Within the license pools (named or default), counts are arranged in *feature slices* based on their feature ID, meaning each slice contains only counts from a single feature (which has a unique feature ID). For each feature, the license pool specifies the name of the feature, its version, and its feature count. The feature count can either be specified as a number or as a percentage of the total of the available feature count.

- **Rules of access allowing access to licenses.**—Rules allow client devices or users that fulfill certain conditions access to counts in specified license pools (named or default), in a given order. For example, the license server administrator could specify that only clients of a specific host type can receive licenses from a certain named license pool.
- **Rules of access denying access to licenses.**—The license server administrator can define criteria that block certain client devices or users from obtaining licenses. This means that no licenses will be allocated to entities meeting those criteria, even if sufficient licenses are available on that license server.

After the model definition has been uploaded to the license server, the definition is saved in the database. Named license pools and rules of access defined in the model definition persist even if the license server is restarted.

Note ▪ *On the local license server, the size of the model definition must not exceed 900 KB.*

Refer to the following section, Model Definition Components, for information about how to create named license pools and building rules of access.

For detailed information about the format and grammar of the model definition, see Model Definition Grammar for Named License Pools. This section also contains some sample definitions that help license server administrators write their own definition.

For information about managing (uploading, deleting, and viewing) the model definition, see Managing Named License Pools.

Model Definition Components

A model definition typically contains the following entities:

- Model
- Named License Pools
- Rules of Access and Conditions

A full description of the grammar and valid values can be found in Appendix D, Model Definition Grammar for Named License Pools. This appendix also includes a number of use cases and sample model definitions that help license server administrators write their own model definition.

This example shows a simple model definition. The individual components are described in the following sections.

```
model "test" {
    partitions {
        partition "techcomm" {
            "f1" 1.0 10%
            "f1" 2.0 5
            "f2" 1.0 3
        }

        partition "marketing" {
            "f1" 1.0 4
            "f1" 2.0 1
            "f2" 1.0 1
        }
    }

    on hostname("ABC") {
        deny
    }

    on hostid("ETHERNET/F01898AD8DD3") {
        use "techcomm"
        continue
    }

    on hostid("ETHERNET/5E00A4F17201") {
        use "marketing"
        continue
    }

    on any() {
        use "default"
        accept
    }
}
```

Model definition name

Named license pools containing features (including feature version and count)

Rules of access defining the conditions that must be met to be granted licenses from particular named license pools

Model

The top line of the model definition specifies the model name. The model name cannot be "default" or "reservations", because these names are reserved.

Example

```
model "test"
```

Named License Pools

The license server administrator can define one or more named license pools under the optional `partitions` element. If no named license pool is specified, all feature counts are placed in the default license pool. (The naming of the element stems from previous releases of the FlexNet Embedded license server, where named license pools were referred to as "partitions".)

For each named license pool, the administrator needs to specify a name followed by one or more features that the named license pool should contain. For each feature, they need to specify the feature name, feature version, and feature count. The feature count can be expressed as a number or as a percentage of the overall feature count that is available on the license server.

Example

```
partitions {
    partition "p1" {
```

```
            "f1" 1.0 50
            "f1" 2.0 25
            "f2" 1.0 100
        }
    }
```

The following section, Rules of Access and Conditions, describes the syntax and condition types used in rules of access.

For information about how the license server allocates feature counts to license pools, see Server Behavior When Distributing Feature Counts to Named License Pools.

Rules of Access and Conditions

Rules define criteria that allow or block certain client devices or users from obtaining licenses from specified license pools. This section covers the following topics:

- Syntax for Rules of Access
- Condition Types
- Advanced Keywords and Directives—Covers how to use advanced keywords to build more sophisticated rules of access.

For detailed information about the format and grammar of the model definition, see Model Definition Grammar for Named License Pools. This section also contains some sample definitions that help license server administrators write their own definition.

Syntax for Rules of Access

Rules of access are defined using the following simplified syntax:

```
on <condition> {
    [use "<license-pool-name>"]
    [accept|deny|continue]
}
```

where <condition> specifies one or more conditions that must be matched.

Example

If the condition dictionary("business-unit" : "engineering") evaluates to true—that is, the capability request's vendor dictionary contains the business-unit:engineering key-value pair—access to the named license pool engineering is granted:

```
on dictionary("business-unit" : "engineering") {
    use "engineering"
    accept
}
```

The following subsections provide more information about configuring rules:

- Conditions Syntax—Describes how to configure a condition in simple terms.
- AND, OR, and NOT Operators—Describes how to combine individual rules to create more complex conditions.

- Basic Keywords and Their Actions—Explains the keywords accept, deny, and continue.
- Default Behavior—Describes how feature counts are served when a feature request does not meet any of the conditions in the model definition.

Conditions Syntax

This section describes the syntax for defining conditions in simple terms. For detailed information, see Model Definition Grammar and Syntax—EBNF.

The following shows a simplified conditions syntax:

```
[NOT]<condition_type>(<parameter_list>) [AND|OR [NOT] <condition_type>(<parameter_list>)]
```

where <condition_type> is replaced with a condition type (hostid, hostname, hosttype, dictionary, or any), followed by a value or value/type pair that qualifies the condition.

Optionally, additional conditions—<condition_type>(<parameter_list>)—can be chained on using an AND, OR, or optional NOT operator.

AND, OR, and NOT Operators

The syntax supports the AND, OR, and NOT operators to combine conditions within a rule to create a more complex rule. The operators have synonyms that are familiar to C and Java programmers:

Operator	Synonym
AND	&&
OR	||
NOT	!

The syntax for combining conditions within a rule using these operators is illustrated in detail in section Model Definition Grammar and Syntax—EBNF. You may also find it helpful to review the provided sample rules.

Precedence Considerations

The AND and OR operators have left-to-right associativity; NOT has right-to-left associativity. NOT has a higher precedence than AND and OR.

Parentheses are not supported. An expression like this:

```
on hosttype("tv") and not hostname("xyz") || hostid("h1")
```

is interpreted as :

```
OR(
    AND(hosttype("tv"), NOT(hostname("xyz"))),
    hostid("h1")
)
```

Sample Rules of Access

The following sample rules in the Appendix C, Model Definition Grammar for Named License Pools, illustrate the use of operators:

- Use Case: Exclusive Use of Feature Counts for Business Unit With Exception of Specific Clients
- Use Case: Exclusive Use of Feature Counts for Business Unit and Specified Clients from other Business Units
- Use Case: Assigning Features Based on Combined hosttype and hostname Properties

Basic Keywords and Their Actions

The following table explains the keywords `accept`, `continue`, and `deny` and their actions. For additional keywords and directives, see Advanced Keywords and Directives.

Table 5-6 • Basic Keywords

Keyword	Action
accept	If the conditions of the rule are met and the rule ends with **accept**, the feature request is granted. No further rules are evaluated.
continue	The **continue** keyword causes rules on subsequent lines to be evaluated, allowing multiple rules that match a single request. The **continue** keyword therefore allows clients to accumulate counts from more than one license pool. For a use case example, see Use Case: Accumulating Counts from Multiple Named License Pools (“Continue” Action). This is the default behavior if no other keywords are present. The lack of an **accept** or **deny** action also means that the **on any** clause (see Default Behavior, below) will be in scope to be matched.
deny	If the conditions of the rule are met and the rule ends with **deny**, the feature request is refused. No counts are acquired and any previously held counts are returned to the server. No further rules are evaluated. If the conditions of the rule are not met and the rule ends with **deny**, the next rule is evaluated.

Default Behavior

When a feature request does not satisfy any of the rules in the model definition, an `on any()` condition is expected that either denies or allows access to a license pool. If no such `on any()` condition is provided, the default is that the feature will be served from the default license pool (if features are available). This behavior is equivalent to the following code:

```
on any() {
    use "default"
    accept
}
```

The examples in this book generally do not explicitly show this final `on any()` condition.

Condition Types

This section describes the following condition types:

- Hostid Condition

- Hostname (Device Name) Condition
- Hosttype (Device Type) Condition
- Vendor Dictionary Condition

These condition types can be combined using AND, OR, and NOT, to form more complex rules. For more information, see AND, OR, and NOT Operators.

Note that in this section, the syntax describing each condition has been simplified. For a detailed syntax description, refer to Model Definition Grammar and Syntax—EBNF.

Hostid Condition

When a client requests a license from the license server, the client includes the unique hostid in the request to which the license server binds the licenses sent in the response. The hostid condition enables the license server administrator to allocate licenses from a specified license pool to one or more client devices that are identified by a hostid.

The hostid is specified as a *value/type* pair (for example, 7200014f5df0/ETHERNET). If a hostid condition does not specify the hostid type, it is assumed that the hostid is of type string.

Syntax

```
hostid_condition = 'hostid', '(', parameter_list, ')' ;
```

Example

The following hostid condition allows any feature requests with hostid h1 or h2 to use features from the named license pool p1.

```
on hostid("h1", "h2") {
    use "p1"
    accept
}
```

Hostname (Device Name) Condition

You as the software producer can set a host name on a client device. (This host name is also sometimes referred to as a device name.) A host name is a human-readable “alias”—in contrast to the hostid—which can optionally be included in a capability request. Communicate the available host names to your enterprise customer’s license server administrator if you want to enable them to use the hostname rule.

Syntax

```
hostname_condition = 'hostname',  '(', parameter_list, ')' ;
```

Example

The following host name condition allows any feature requests with host name ABC to use features from the named license pool p2.

```
on hostname("ABC") {
    use "p2"
    accept
}
```

Note that the host name is case-sensitive.

Hosttype (Device Type) Condition

You as the software producer can set a host type on a client device. (This host type is also sometimes referred to as a device type.) A host type can optionally be included in a capability request. Communicate the available host types to your enterprise customer's license server administrator if you want to enable them to use the hosttype rule.

Syntax

```
hosttype_condition = 'hosttype', '(' parameter_list, ')' ;
```

Example

The following host type condition allows any feature requests with host type `device` to use features from the named license pool `p3`.

```
on hosttype("device") {
    use "p3"
    accept
}
```

Note that the host type is case-sensitive.

Vendor Dictionary Condition

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 license server and vice-versa. Basically, the vendor dictionary provides a means to send information back and forth between the client and server for any producer-defined purposes, as needed; FlexNet Embedded does not interpret this data.

Vendor dictionary data is stored as key–value pairs. The key name is always a string, while a value can be a string or a 32-bit integer value, or an array of those types (arrays are only supported when using the Cloud Monetization REST API). Keys are unique in a dictionary and hence allow direct access to the value associated with them.

Communicate the available vendor dictionary data to your enterprise customer's license server administrator if you want to enable them to use the vendor dictionary rule.

Syntax

```
vendor_dictionary_condition = 'dictionary', '(', keyword_parameter_list, ')' ;
```

Example

The following vendor dictionary condition allows any feature requests from the `engineering` business unit (as defined in the vendor dictionary) to use features from the named license pool `p4`.

```
on dictionary("business-unit" : "engineering") {
    use "p4"
    accept
}
```

Tip ▪ *For an example that demonstrates a vendor dictionary condition that uses an array, see Use Case: Making Feature Counts Available to Multiple Business Units.*

Advanced Keywords and Directives

The following table introduces advanced keywords and directives and their usage. For a description of the keywords `accept`, `continue`, and `deny`, see Basic Keywords and Their Actions.

Table 5-7 ▪ Keywords and Directives for Rules of Access

Keyword or Directive	Description	Refer to Section
`remainder`	Allocates any remaining license counts to a specified license pool. Useful in scenarios where integer rounding could cause left-over counts to remain in the default license pool when trying to configure license pools summing up to a count of 100%.	Allocating Remaining License Counts
`vendor string matches`	Allocates feature counts to license pools based on variables specified in the vendor string.	Allocating Counts Based on Vendor String Property
`max`	Limits the number of feature counts that a single user or device can consume.	Limiting Checkout of Feature Counts per User or Device
`without requested features`	Lets the license server specify the features that should be served to a client when a capability request lists no desired features.	Letting Server Specify Counts
[regular expressions]	Match patterns to reduce the number of license pools and conditions. Syntax: `~/REGEX/`	Using Regular Expressions

Allocating Remaining License Counts

The `remainder` keyword can be used to allocate any remaining license counts to a specified named license pool. This keyword is particularly useful in preventing the situation where integer rounding would cause left-over counts to remain in the default license pool when trying to configure named license pools summing up to a count of 100%.

Using the `remainder` keyword is equivalent to specifying `100%`, which allocates all available license counts. Any license pool later in the model definition than the one specifying the `remainder` keyword (or `100%`) in the model definition (including the default license pool) receives zero counts for that feature. For an example demonstrating its use, see the Use Case: Named License Pool Receiving Entire Remaining Feature Count.

Example

```
partitions {
    partition "p1" {
        "f1" 1.0 33%
    }

    partition "p2" {
        "f1" 1.0 33%
```

```
        }

        partition "p3" {
            "f1" 1.0 remainder
        }
    }
```

Allocating Counts Based on Vendor String Property

The directive **vendor string matches** enables license administrators to allocate feature counts to license pools based on variables specified in the vendor string. After use, feature counts are returned to their original license pool.

To use the directive, the producer must define a vendor string in the license model when they create a line item in the back office. The vendor string can hold arbitrary producer-defined license data. Data can range from feature selectors to pre-defined substitution variables. For more information, see Set Up the Selectors in FlexNet Operations, or Predefined Substitution Variables for License Strings in the FlexNet Operations User Guide.

The directive **vendor string matches** is located in the `partitions` section of the model definition. This directive is evaluated when the license server loads the model definition.

Syntax Example

```
model "exampleModel" {
    partitions {
        partition "engineering" {
            "f1" 1.0 75% vendor string matches "ProductName:Premium"
            "f2" 1.0 75% vendor string matches "ProductName:Premium"
        }
        partition "sales" {
            "f1" 1.0 25% vendor string matches "ProductName:Basic"
            "f2" 1.0 25% vendor string matches "ProductName:Basic"
        }
    }
```

For a use case example, see Use Case: Assigning Features Based on Vendor String Property.

Limiting Checkout of Feature Counts per User or Device

Use the **max** keyword to limit the number of feature counts that a single user or device can consume.

max is used in the `partitions` section of the model definition. The following shows an example model definition which limits the consumption of features f1 and f2 to 10 and 1 counts, respectively:

Syntax Example

```
model "exampleModel" {
    partitions {
        partition "engineering" {
            "f1" 1.0 100 max 10
            "f2" 1.0 100 max 10
        }
        partition "sales" {
            "f1" 1.0 5 max 1
            "f1" 1.0 5 max 1
        }
```

```
    }

    on dictionary("business-unit" : "engineering") {
        use "engineering"
        accept
    }

    on dictionary("business-unit" : "sales") {
        use "sales"
        accept
    }
}
```

If `max` is set to 0, access to that feature from the license pool is denied.

If a client requests a feature count that is greater than the value of `max`, the request is denied with the error `FEATURE_COUNT_INSUFFICIENT`.

To allow a partial checkout, set the “partial” attribute to true. In that scenario, a client is granted access to the requested number of features up to the value of `max`, even if a feature count exceeding `max` is requested.

Letting Server Specify Counts

There might be scenarios when the license server administrator wants the license server to specify the features that should be served to a client. The server cannot override a capability request containing desired features coming from the client. However, if a capability request lists no desired features, the `without requested features` directive lets the server allocate feature counts.

The administrator can use the `without requested features` directive to do the following:

- **Allocate specific feature counts.** To achieve this, list the feature counts that should be served. The following example shows the syntax:

  ```
  on hostname("myhost") {
      use "p1", "default"
      without requested features {
          "f1" 1.0 1
      }
      accept
  }
  ```

  For a use case example, see Use Case: Letting Server Specify Counts.

- **Allocate the entire feature count from a license pool.** To achieve this, combine `without requested features` with the `all from "license_pool_name"` instruction. Replace *`license_pool_name`* with the name of the named license pool whose entire list of features, as specified in the model definition, should be allocated. Note that if the specified license pool does not have sufficient counts to satisfy the request, counts are allocated from subsequent license pools listed in the rules of access (provided that the client has access to those license pools). The following example shows the syntax:

  ```
  on hostname("myhost") {
      use "p1", "default"
      without requested features {
          all from "p1"
      }
      accept
  }
  ```

For a use case example, see Scenario: Server-specified Counts.

Using Regular Expressions

Rules of access can include regular expressions to match patterns in conditions. This means that the number of license pools and conditions can be greatly reduced, simplifying management of license pools.

Regular expressions can be included in conditions using the following syntax: `~/REGEX/`

For a list of supported expressions, see https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/regex/Pattern.html.

Full vs Partial Equality

The regular expression is generally matched against the whole value of the property on which the condition is based (that is, hostid, hostname, hosttype, or vendor dictionary string). It is not necessary to add the `^` and `$` meta characters to signify the beginning and end of the expression.

Example

The following expression...

```
on hostname("~/[A-Z]{2}[0-9]{3}/") {
    ...
}
```

... matches the hostname **`AB123`**, but not **`ABCD1234`**.

However, when a regular expression is used in combination with the `vendor string matches` directive, the expression is used as a “contains” match.

Example

The following expression...

```
… vendor string matches "~/%%ProductName:[A-Z]{3}[0-9]{3}%%/
```

... matches **`%%ProductName:AAA111%%`** found anywhere inside the vendor string.

For a use case example, see Use Case: Using Regular Expression to Allocate License Counts.

Server Behavior When Distributing Feature Counts to Named License Pools

This section describes the distribution of feature counts to named license pools for these situations:

- When a New Model Definition Is Uploaded
- When the Feature Count Changes on the License Server
- When Clients Return or Renew Counts to the Server

When a New Model Definition Is Uploaded

When the model definition is uploaded to the license server, feature counts are placed into license pools. Counts are allocated to named license pools in the order that is specified in the model definition, starting with the topmost named license pool.

The following rules apply when allocating feature counts of the same name, but with different versions and expiry dates:

1. Features that match the version specified in the model definition are allocated first. Features with longer expiry dates are allocated before features with a shorter expiry date.
2. If no features are available that match the version specified in the model definition, features of the next higher version are allocated. Features with longer expiry dates are allocated before features with a shorter expiry date. (Features of a lower version are not allocated.)

Features holding a start date in the future will not be allocated to named license pools but will remain in the default license pool. A feature with a future start date will stay in the default license pool even beyond its start date until the model definition is reapplied to the license server.

Within named license pools, counts are arranged in feature slices based on their feature ID, meaning each slice contains only counts from a single feature (which has a unique feature ID). Named license pools that have the exact feature counts, including the exact feature versions, as specified in the model definition are considered *fully satisfied*.

When the available number of counts has been distributed, any named license pool for which insufficient counts are available will remain empty or will only be partially filled.

When the model definition on the license server changes, any counts that are currently in use by clients are recorded as used against the named license pool the counts originate from. If a named license pool is deleted, it stays active until all used counts have been returned to it. Only after that is the named license pool removed. This enables the license administrator to maintain an overview of all used feature counts.

Optimization Logic For Model Definition Changes

When the model definition changes, any named license pools that are already fully satisfied are skipped when feature counts are placed into named license pools. As a result, it is possible that existing features in fully satisfied named license pools are not replaced with newly available features that have a longer expiry date. Instead, these newly available features are placed into lower-ranked named license pools (provided that these are not fully satisfied) or the default license pool.

To ensure that higher-ranking named license pools receive features with longer expiry dates, it is recommended that you delete and re-upload the model definition. This forces a redistribution of feature counts across all license pools, regardless of the current allocation of feature counts.

When the Feature Count Changes on the License Server

When feature counts are added, reduced or expire on the license server, the server reapplies the model definition. The server redistributes the counts across license pools as described in When a New Model Definition Is Uploaded.

When Clients Return or Renew Counts to the Server

When there have been no changes to the model definition or to the licenses held by the server, licenses returned by clients are returned to the original slice from which they were acquired.

However, in scenarios where clients were holding usage prior to a model change or a change to the licenses held by the server, counts returned by clients may need to be redistributed to reflect the new state of the server.

The following sections describe more about feature count redistribution:

- Basic Redistribution Rules
- Order of Redistributing Counts

In these sections, *slices in deficit* refers to slices that have fewer counts than they should have, according to the model definition and the number of counts the server has been provisioned with ("`slice`" < "`computedCount`").

Basic Redistribution Rules

When clients had counts in use prior to a model change or prior to a change of feature counts on the license server, the following basic redistribution rules apply:

- A client cannot renew counts that it could not have acquired, based on the current model definition that is in force on the license server.
- Counts can only be reallocated to slices with a matching feature ID.

Order of Redistributing Counts

The server redistributes returned counts in the following order:

1. The server allocates feature counts to reduce or eliminate overage before redistributing counts.
2. Counts are redistributed to slices in deficit that are accessible to the client, in the order in which license pools that are accessible to the client are specified in the current model definition.
3. Counts are redistributed to slices in deficit, irrespective of whether these are accessible to the client. The server redistributes counts to license pools in the order in which they are specified in the current model definition.
4. Returned counts are only returned to overdraft after all slices in deficit have been adjusted to the counts they should have according to the model definition. Overdraft counts are only ever available from the default license pool.
5. Any remaining counts are returned to the default license pool.

Server Behavior when Assigning Features to Clients

When the license server receives a capability request from a client, it tries to serve the feature counts matching the requested version from the first named license pool that is configured in the model definition. If the request cannot be fulfilled from the first named license pool, or can only be partially fulfilled, the server will try to serve any remaining counts to the client from further feature slices and license pools in a specific hierarchy. The following rules apply, in order, when fulfilling capability requests:

1. Regular counts take priority over overdraft counts (overdraft counts are only available from the default license pool).
2. Counts are served from accessible named license pools in the order license pools are specified in the model definition (starting from the top).
3. Counts that match the requested version are served before counts of a higher version (lower versions are not accepted).
4. Counts with the shorter expiry date are served before counts with a later expiry date.

The following diagrams illustrate this hierarchy. Figure 5-1 shows the order in which feature counts are assigned from license pools.

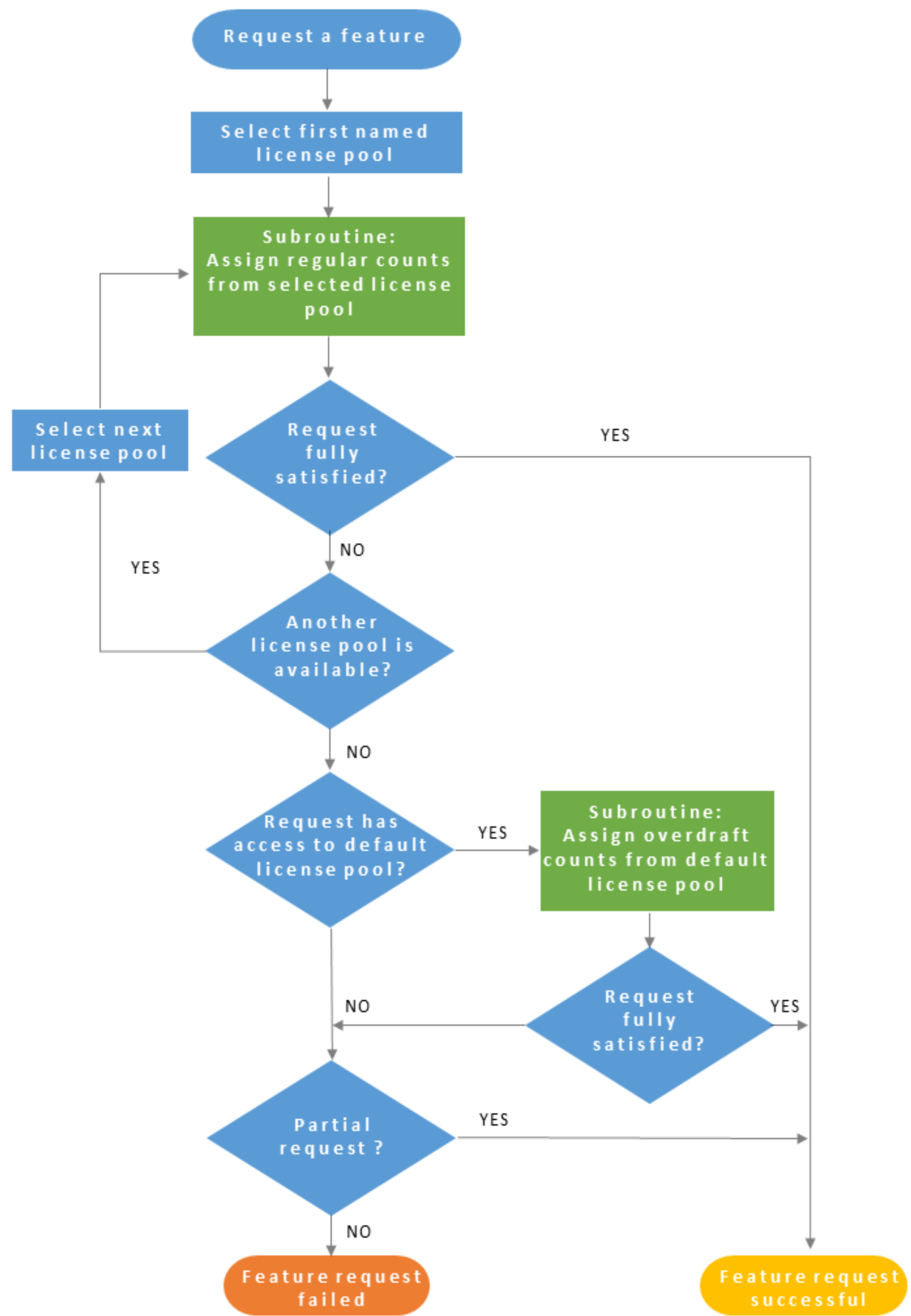


Figure 5-1: Flow of feature allocation

Figure 5-1 includes a subroutine (highlighted in green) that explains how counts are assigned if the request is served from more than one slice. This subroutine—see Figure 5-2—also comes into play when regular (non-overdraft) counts are exhausted, but the request has access to the default license pool which includes overdraft counts:

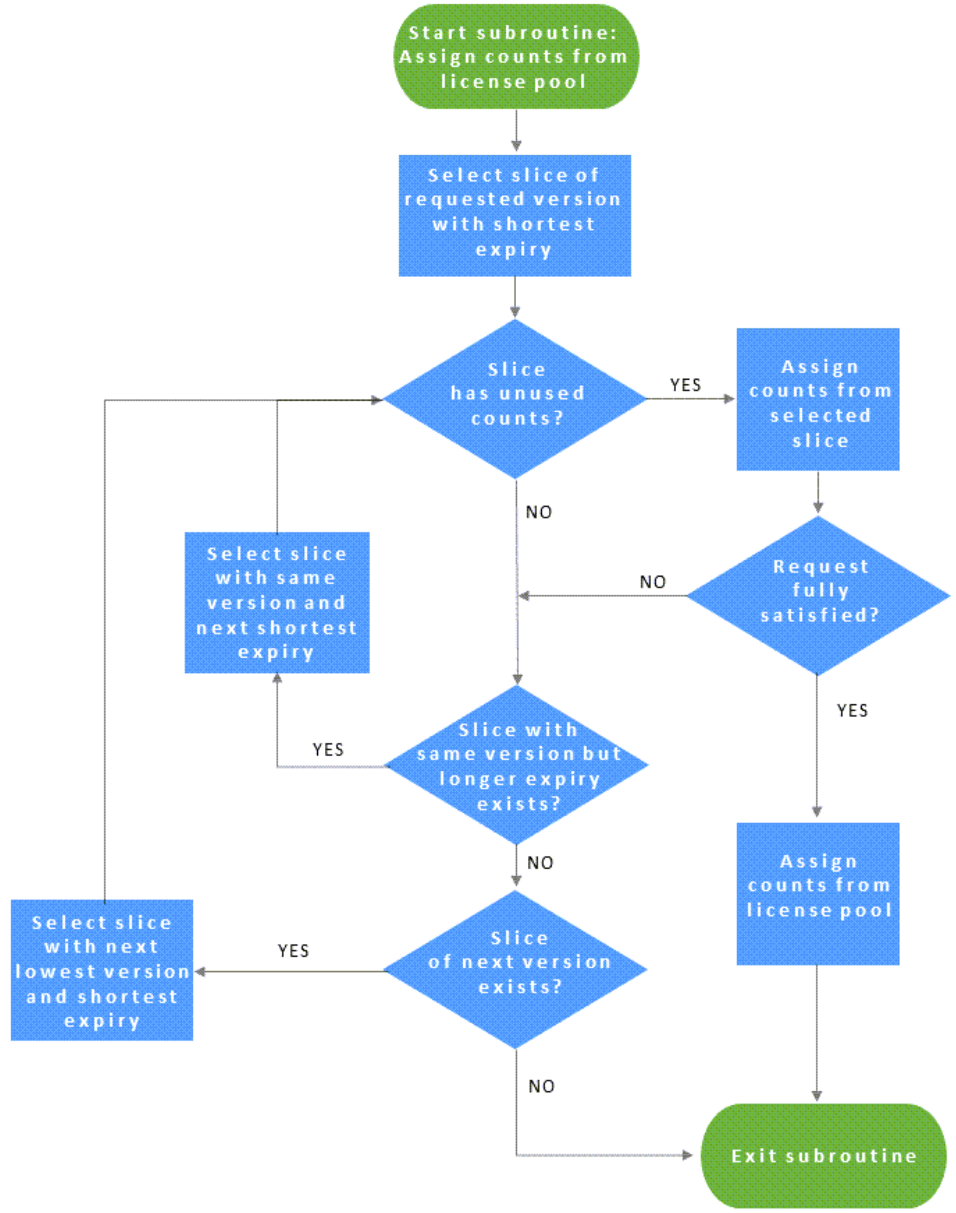


Figure 5-2: Flow of feature allocation in the subroutine

Named License Pools vs. Reservations

Named license pools are conceptually similar to reservations but offer significantly more flexibility in controlling a server's license estate. If currently no license reservations are defined for your license server, Revenera strongly recommends to use the Named License Pools functionality to allocate licenses.

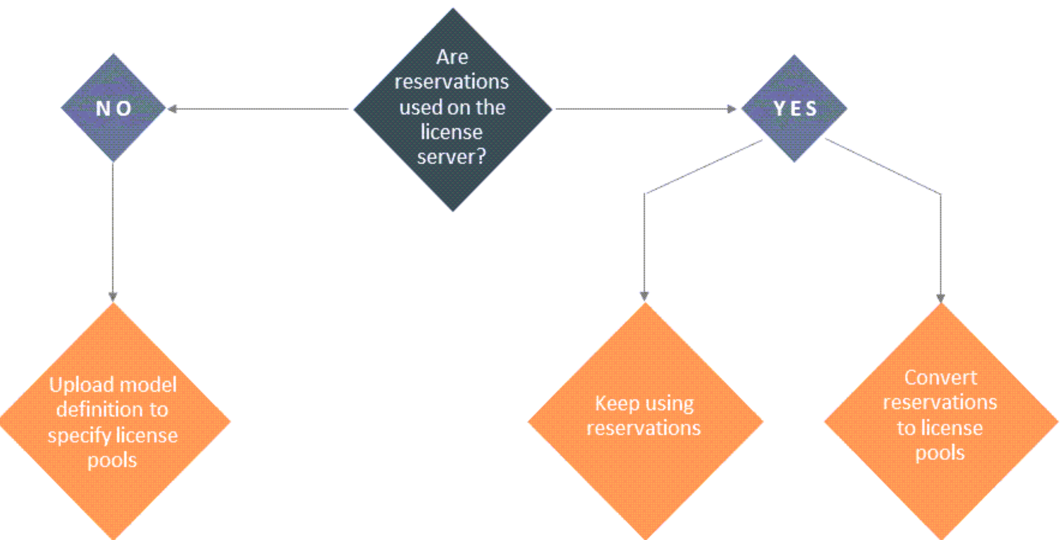
Read this section if one or more of the following statements apply to you:

- You want to know the pros and cons of using named license pools or reservations. See Comparison of Named License Pools and Reservations.

- You are currently using reservations and would like to find out how named license pools affect the behavior of reservations. See Impact of “Named License Pools” Functionality on Using Reservations.
- You have previously been using reservations and would like transition to named license pools. See Transitioning from Reservations to Named License Pools.
- You have transitioned to named license pools but want to use reservations again. See Returning to Reservations after Transitioning to Named License Pools.

If you have not been using reservations before and are now starting to use named license pools, you can skip this section.

The following diagram illustrates your options:



Comparison of Named License Pools and Reservations

If reservation groups or entries change frequently, it is recommended to use reservations. Otherwise, it it is recommended to use named license pools.

The following table lists the main differences between named license pools and reservations.

Table 5-8 ▪ Named License Pools vs. Reservations

Action	Named License Pools	Reservations
Counts are allocated based on...	• hostid • host name (device name) • host type (device type) • vendor dictionary data	• hostid

Table 5-8 ▪ Named License Pools vs. Reservations

Action	Named License Pools	Reservations
If the license server has been provisioned with less counts than are posted in a reservation or are specified in the model definition, respectively,...	...the license server allows access to feature counts based on the rules in the model definition until all available counts are in use by entitled clients.	...the license server rejects the entire reservation group.
Users' and devices' access to feature counts can be...	...allowed or denied.	...allowed.
Criteria that allow or block certain client devices or users from obtaining licenses can be configured using...	...an extensible model language.	...a file in JSON format.

Impact of "Named License Pools" Functionality on Using Reservations

The introduction of the Named License Pools functionality has no impact on the reservations functionality. A license server administrator can continue configuring reservations using the FlexNet License Server Administrator command-line tool, and the license server will distribute licenses according to the reservations that have been configured.

When reservations are configured, attempts to upload a model definition will fail.

Transitioning from Reservations to Named License Pools

If the license server administrator decides to move from using reservations to using named license pools, they need to perform the following steps:

1. Delete any reservation groups.
2. Create a new model definition which defines rules and named license pools (see Introduction to the Model Definition and Uploading the Model Definition)

The new model definition takes effect immediately after upload.

Before starting the transition, the license server administrator might want to view information about the reservations currently active on their license server. Procedural information is available under APIs to Manage Named License Pools in Chapter 4, License Server REST APIs.

A Note on Reserved Counts in Use

If any reserved license counts are in use by clients while transitioning from reservations to named license pools, such used counts are recorded as used against the default license pool. They remain in the default license pool until the client renews or returns these counts, after which point the license counts are allocated to their named license pool according to the model definition.

Returning to Reservations after Transitioning to Named License Pools

To return to using reservations, the license server administrator needs to delete the active model definition. After that, they can configure reservations using the FlexNet administrator tools supplied with the license server software package. For a description of the endpoints for managing reservations, see APIs to Manage Reservations in Chapter 4, License Server REST APIs.

For information about deleting the active model definition, see Deleting the Model Definition.

Limitations of Named License Pools

Note the following about named license pools:

- Named license pools support only concurrent features, but not metered features. Metered features will always remain in the default license pool. This matches the behavior of reservations, because metered features cannot be reserved.
- Using feature selectors in combination with named license pools may produce unexpected results. A feature selector is a key-value structure included as part of the feature's definition created in the back office.To find out more, see the *FlexNet Operations User Guide*.
- The license server failover configuration does not support named license pools. During a failover period, feature counts will only be distributed from the default license pool.

License Reservations

Important ▪ *A license server administrator can use either reservations or named license pools. Reservations and named license pools cannot coexist alongside each other. For more information, see Named License Pools vs. Reservations.For general information about named license pools, see Allocating Licenses Using Named License Pools.*

If currently no license reservations are defined for a license server, review the information in Comparison of Named License Pools and Reservations to decide which approach is best for you. In general, reservations are conceptually similar to named license pools, but named license pools offer significantly more flexibility in controlling a server's license pool.

The license reservation feature available with the FlexNet Embedded license server (the local license server or a CLS instance) provides a means for the license server administrator at a customer's enterprise to control the distribution of the server's license pool.

With no reservations defined, the license server distributes licenses on a first-come-first-served basis to client devices requesting them. However, with the use of reservations, the server can pre-allocate a certain number of licenses from the pool to specific entities (for example, client devices or users) to help ensure that these entities have access to the licenses they need. The license server administrator can choose to pre-allocate the entire pool of licenses, a portion of the licenses, or none of the licenses. Any license not reserved remains in the shared pool of licenses and is available on a first-come-first-served basis.

The following sections provide more information about reservations:

- Overview of Reservation Types

- Reservation Hierarchy
- Managing Reservations
- License Allocation on the Server
- Processing the Capability Request When Reservations Are Used
- Reservation Considerations and Limitations

Overview of Reservation Types

The FlexNet Embedded license server supports two types of license reservations: device-based and user-based. The reservations defined for a license server can be all of one type of reservation or a combination of both types. The following sections provide more information about these reservation types and their associated hostids:

- Device-based Reservations
- User-based Reservations
- About Main and Secondary Hostids

Device-based Reservations

A device-based reservation is assigned to a unique hostid identifying the client device on which your product (enabled with FlexNet Embedded licensing) is running. Whenever that device requests a specific feature count (called a *desired feature*), the license server checks the current reservations to determine whether a reservation for the requested feature exists for the specified hostid. The following happens:

- If a reservation is found and the reserved count is enough to satisfy the requested count, the requested count of the feature is granted from the reserved count and sent to the device.
- If no reservation is found or the reservation count is not enough to satisfy the requested feature count, the server attempts to obtain the remaining requested license count from the combination of reserved and shared (non-allocated) feature counts remaining in the license pool. If the sum of the available shared and reserved counts is insufficient to satisfy the request, the request for the desired feature is denied.

The client device can also send a request without specifying any desired features. In this case, the full count for all reserved features allocated to the device is served. However, if the device has no reserved features allocated, no new licenses are served.

Note about Special Capability Request Options

Capability request options—specifically the “incremental” option used to preserve the client device’s currently served features and the “partial” attribute assigned to a desired feature to allow partial checkout should the feature’s count fall short of the desired count—can affect the license server’s behavior in granting licenses from reserved and shared counts. See the appendix Effects of Special Request Options on Use of Reservations for more information.

User-based Reservations

A user-based reservation is assigned to a particular user instead of a client device. A user with reserved counts for a feature can access these counts from any client device. Thus, user-based reservations help to ensure that users who always need access to certain licenses can obtain them from any client device. (For example, users might need to access their licensed application from their laptop, tablet, or phone.)

Implementing a user-based reservation typically involves specifying a *secondary hostid* for the user in the capability request. When processing a request that contains a secondary hostid, the license server follows the same basic process described for Device-based Reservations, with the added step of searching for reservations that match the secondary hostid:

- If the capability request specifies desired features, the license server attempts to satisfy the desired count for a given feature first from reservations for the client device. If this count is insufficient, the server then searches reservations for the user (specified by the secondary hostid) for an additional count. If the sum of reserved device-based and user-based counts is still not enough, the server attempts to satisfy the remaining desired count from the pool of shared features.
- If the capability request contains no desired features, the license server serves the full count of all reserved features allocated to the client device, as well as the counts of reserved features allocated to the user. If both the device and the user have no reserved features allocated, no licenses are served.

As described in the previous section, certain options sent in the capability request can affect the license server's behavior in granting licenses from reserved counts. See Effects of Special Request Options on Use of Reservations for more information.

***Note** ▪ When a secondary hostid (typically used for user-based reservations) requests reserved feature counts, these counts are ultimately served to and returned from the client device from which they were requested. A secondary hostid (that is, a user) can use multiple client devices to request partial counts of its available reserved count for a given feature. Once the secondary hostid has checked out all its available reserved counts for the feature across one or more devices, the hostid's next request for reserved counts for this feature will not be satisfied unless the reserved counts served to any of these devices have been previously returned to the license server.*

About Main and Secondary Hostids

The capability request always includes a main hostid, but can also include one or more secondary hostids. The following provides more information about these two types of hostids.

Main Hostid

The *main hostid* sent in the capability request is the hostid to which the license server binds the licenses sent in the response. For security purposes, this hostid is traditionally the unique client-device hostid. The client-device hostid is also used in defining device-based reservations.

Secondary Hostid

One or more *secondary hostids* can also be sent in the capability request (and are identified as such in the request). In general, secondary hostids are sent in the capability request as a means of capturing extra data about the FlexNet Embedded clients that can later be seen in reports in FlexNet Operations. Another use for a secondary hostid is to provide a second key for locating reservations—typically, user-based reservations— on the license server.

Should multiple secondary hostids be sent in the capability request, the license server uses only the *first* secondary hostid listed in the request to check for reservations.

***Important ▪** Capability requests must not use a main hostid as secondary hostid or vice-versa. Capability requests that specify the same hostid as a main hostid in one request and as a secondary hostid in another are rejected.*

Reservation Setup

When setting up a license reservation on the license server, the enterprise license server administrator must provide a hostid to identify the entity—client device or user—to which the reservation belongs. However, no specification exists in the actual reservation definition to distinguish this hostid as main or secondary.

If a secondary hostid is included in the capability request, the license server's search for reservations to satisfy the request is handled differently from when only the main hostid is present. The only way the license server knows to treat the hostid for a given reservation as secondary during its reservation search is through the hostid's designation as secondary in the capability request. For more information about setting up the license server's reservations, see Managing Reservations. For a detailed description of how reservations are used to satisfy requested features, see Processing the Capability Request When Reservations Are Used.

As producer, you must inform the license server administrator of the specific hostid types (for example, ETHERNET, STRING, USER, or other) that can be used for the reservation definitions.

Reservation Hierarchy

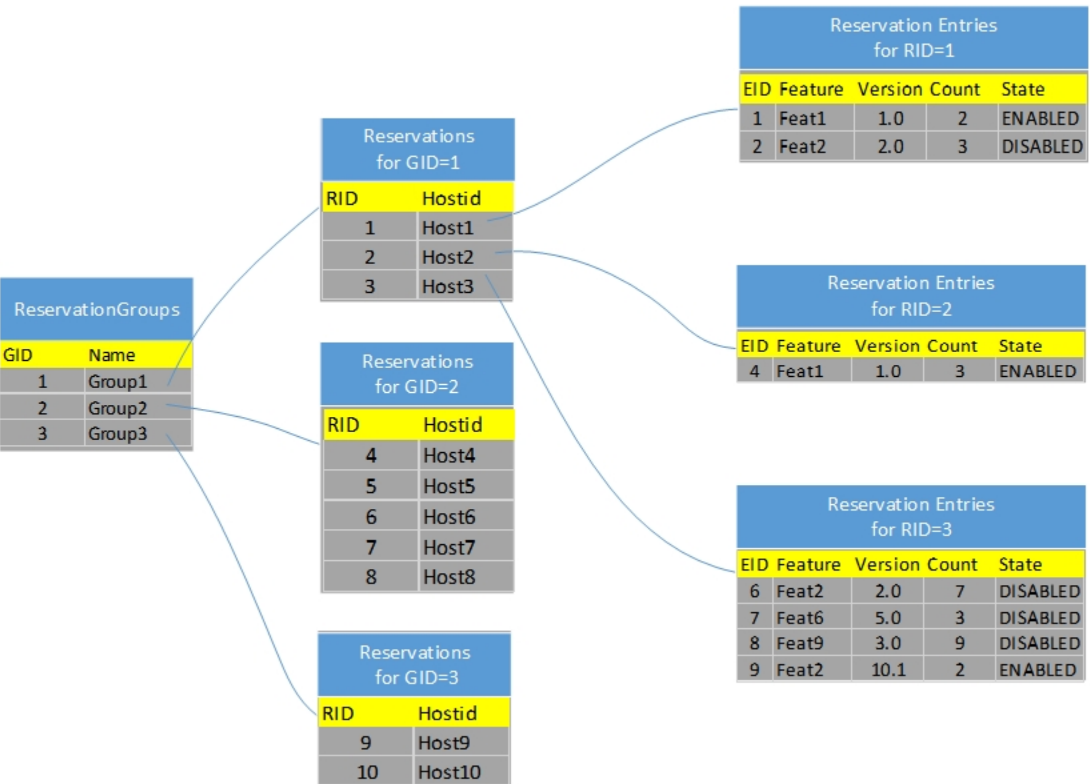
The reservations you post to the license server are ordered in a hierarchy, enabling more granularity in managing them. For example, *reservation entries* are the actual feature reservations assigned to a *reservation*, which is identified by the hostid for the specific client device or user to which the feature reservations apply. One or more reservations are assigned to a *reservation group*, which represents an entity to which the client devices and users identified by the reservations belong. This entity might be a department, a job role assigned to specific employees within the company, a physical location, or any other entity applicable to your company. This hierarchy enables you to view, add, or delete reservations at the group level, the reservation (hostid) level, or the reservation entry (feature) level.

When reservations are added to the license server, the following IDs are generated for the various hierarchal components that define a reservation:

- A *group ID* (GID) for a reservation group
- A *reservation ID* (RID) for the reservation, based on the hostid identifying a specific client device or user within the reservation group
- A *reservation-entry ID* (EID) for each reservation entry defined in a specific reservation

The following diagram illustrates the reservation hierarchy. For example, reading from right to left, trace the reservation entry **EID 1** for feature **Feat1**:

- **EID 1** belongs to the reservation **RID 1**, defined for the hostid **Host1**.
- **RID 1** is one of three reservations belonging to the reservation group **GID 1**, also named **Group1**.



Managing Reservations

Important ▪ *You can use either reservations or named license pools. Reservations and named license pools cannot coexist alongside each other. For more information, see Named License Pools vs. Reservations.For general information about named license pools, see Allocating Licenses Using Named License Pools.*

The following sections provide an overview of ways to manage reservations using the `/reservationgroups` REST endpoints. You can use these endpoints to write your reservation-management tool (or you can simply distribute one of the FlexNet administrator tools supplied with your license server software package). For a complete description of these endpoints, see APIs to Manage Reservations in Chapter 4, License Server REST APIs.

- Add a New Reservation Group
- Add Reservations to an Existing Group
- Enable or Disable Reservation Entries
- Delete Reservations

Important ▪ *If you were previously using named license pools, you must delete the model definition before creating reservations. See Deleting the Model Definition.*

Add a New Reservation Group

Important ▪ *If you were previously using named license pools, you must delete the model definition before creating reservations. See Deleting the Model Definition.*

The following shows an example of a request (in JSON format) that you post to define a new reservation group—in this case, a group named **support**, containing reservation entries for two reservations, one for hostid **7A7A77AAA77A** and one for hostid **Joe**:

Table 5-9 ▪ Add a New Reservation Group

```
{ "name" :  "support", "reservations" :
  [
    { "hostId": { "value" : "7A7A77AAA77A", "type" : "ETHERNET" },
      "reservationEntries": [
         { "featureName": "f1", "featureVersion": "1.0", "featureCount": 1 },
         { "featureName": "f2", "featureVersion": "1.0", "featureCount": 2 } ]},
    { "hostId": { "value" : "Joe", "type" : "USER" },
      "reservationEntries": [
         { "featureName": "f3", "featureVersion": "1.0", "featureCount": 3 }]}
]}
```

When this request is posted without any errors, the `/reservationgroups` REST endpoint shows the added group **support** with its GID **194**:

```
[ {
  "creationDate" : "2019-02-27T16:27:33.013Z",
  "ipAddress" : "10.20.42.164",
  "name" : "Support",
  "id" : 194

} ]
```

The `/reservationgroups/194/reservations` endpoint shows the two new reservations—one for hostid **7A7A77AAA77A** (RID **193**) and one for hostid **Joe** (RID **196**)—to the reservation group **support**.

```
[ {
  "id" : 193,
  "lastModified" : "2019-02-27T16:27:33.013Z",
  "ipAddress" : "10.20.42.164",
  "reservationEntries" : [ {
    "id" : 194,
    "state" : "ENABLED",
    "featureCount" : 1,
    "featureVersion" : "1.0",
    "featureName" : "f1"
  }, {
```

```
      "id" : 195,
      "state" : "ENABLED",
      "featureCount" : 2,
      "featureVersion" : "1.0",
      "featureName" : "f2"
    } ],
    "hostId" : {
      "type" : "ETHERNET",
      "value" : "7A7A77AAA77A"
    }
  }, {
    "id" : 196,
    "lastModified" : "2019-02-27T16:27:33.013Z",
    "ipAddress" : "10.20.42.164",
    "reservationEntries" : [ {
      "id" : 197,
      "state" : "ENABLED",
      "featureCount" : 3,
      "featureVersion" : "1.0",
      "featureName" : "f3"
    } ],
    "hostId" : {
      "type" : "USER",
      "value" : "Joe"
    }
  } ]
```


Errors that Cause Rejection of New Reservation Groups

When a new reservation group is processed during the posting phase, the *entire* group is rejected if any of following errors exist.

- A group name is repeated within the reservation posting. A group name must be unique within the *entire* reservation posting.
- A hostid is used in more than one reservation group. A given hostid can be used only once, not only within a given group, but within the entire reservation posting.
- A hostid is missing. A group reservation cannot be created without at least one reservation, defined by a hostid and its reservation entries. Additionally, reservation entries not associated with a reservation cause the entire reservation group to be rejected.
- A reservation has no reservation entries associated with it.
- A reservation entry is invalid. For example, the entry might be missing a feature name, version, or reserved count, or the specified reserved count is less than or equal to zero.
- The feature or feature version specified in a reservation entry is not found in the license pool.
- The feature specified in a reservation entry is expired or has not yet started.
- The feature specified in a reservation entry is a metered feature.
- The license server has an insufficient count of a given feature to satisfy the *total* requested count for that feature across all reservation entries within the entire group. For more explanation, see the later section License Allocation on the Server.

Add Reservations to an Existing Group

Within an existing reservation group, you can add the following:

- A new reservation with its reservation entries
- New reservation entries to an existing reservation

Add a New Reservation

Using the `/reservationgroups` REST endpoint you can add a new reservation with one or more reservation entries to an existing reservation group.

The following shows an example of a request (in JSON format) that you post to define a new reservation—in this case, for hostid **dev1**. The reservation contains three reservation entries, one each for features `f1`, `f2`, and `f3`.

Table 5-10 ▪ Add a New Reservation to an Existing Group

```
{ "hostId": { "value" : "dev1", "type" : "USER" },
   "reservationEntries": [
         { "featureName": "f1", "featureVersion": "1.0", "featureCount": 1 },
         { "featureName": "f2", "featureVersion": "1.0", "featureCount": 2 }
         { "featureName": "f3", "featureVersion": "1.0", "featureCount": 5 }
]}
```

The reservation is successfully added as long as no error is encountered. (A subset of errors that cause a reservation group to be rejected cause a reservation to be rejected as well. See Errors that Cause Rejection of New Reservation Groups for a error list.)

Especially note that, if the license server's feature counts can satisfy the requested feature counts for *all* reservation entries in the reservation, the reservation is added. However, if any one reservation entry cannot be satisfied by the current license server counts, the *entire* reservation is rejected. For more information, see License Allocation on the Server.

Add New Reservation Entries to an Existing Reservation

Using the `/reservationgroups/`*`GID`*`/reservation/`*`RID`*`/entries` REST endpoint, you can add new reservation entries to an existing reservation.

The following shows an example of a request (in JSON format) that you post to add a new reservation entries to an existing reservation:

Table 5-11 ▪ Add New Reservation Entries to an Existing Reservation

```
[
{ "featureName": "f1", "featureVersion": "1.0",
    "featureCount": 1 },
{ "featureName": "f2", "featureVersion": "1.0",
    "featureCount": 2 },
{ "featureName": "f3", "featureVersion": "1.0",
    "featureCount": 3 }
]
```

The reservation is successfully added as long as no error is encountered. (A subset of errors that cause a reservation group to be rejected cause reservation entries to be rejected as well. See Errors that Cause Rejection of New Reservation Groups for a error list.)

Especially note that, if the license server's feature counts can satisfy the requested feature counts for *all* the reservation entries being added to the reservation, the entries are added. However, if any of the requested feature counts cannot be satisfied by the current license server counts, the *entire* array of new reservation entries is rejected. For more information, see License Allocation on the Server.

Enable or Disable Reservation Entries

A new reservation entry is never added with a "disabled" status. However, when license rights on a the license server change, reservation entries can become disabled due to an insufficient feature count on the license server.

To view disabled reservation entries, use the `/reservationgroups/GID/reservations` REST endpoint. You have the option to delete disabled reservation entries or re-enable them should more licenses become available on the license server through a new capability response from the back office or through the deletion of other reservation entries. You can also disable reservation entries if you choose.

Use the `/reservationgroups/GID/reservations/RID/entries/EID/state` REST endpoint to enable or disable reservation entries.

Delete Reservations

Using the `/reservationgroups` REST endpoints, you can delete an entire reservation group, individual reservations within a given group, or individual reservation entries within a given reservation. See APIs to Manage Reservations in Chapter 4, License Server REST APIs, for details.

License Allocation on the Server

License allocation when using reservations is handled as follows for these situations:

- When Adding a New Reservation Group
- When Adding Reservations to an Existing Reservation Group
- When Feature Counts Change on the Server

When Adding a New Reservation Group

In the attempt to add a new reservation group, the license server determines the total feature counts required by all the reservation entries within the new group. If the totals cannot be satisfied by counts not reserved to other reservation groups on the license server, the entire new group is rejected.

Example Allocations When Adding a New Reservation Group

The examples described in this section use the following reservations to pre-allocate the licenses in the server's license pool. All these reservations are being created in the single new reservation group **payroll**.

Table 5-12 • Reservations Used in Example Allocations

```
{ "name" :  "payroll", "reservations" :
  [
    { "hostId": { "value" : "111A11AAA11A", "type" : "ETHERNET" },
      "reservationEntries": [
         { "featureName": "f1", "featureVersion": "1.0", "featureCount": 1 },
         { "featureName": "f3", "featureVersion": "1.0", "featureCount": 2 } ]},
    { "hostId": { "value" : "222A22AAA22A", "type" : "ETHERNET" },
      "reservationEntries": [
         { "featureName": "f3", "featureVersion": "1.0", "featureCount": 1 } ]},
    { "hostId": { "value" : "Joe", "type" : "USER" },
      "reservationEntries": [
         { "featureName": "f3", "featureVersion": "1.0", "featureCount": 3 }]}
]}
```

Example 1: Sufficient Feature Counts

In this example, a license server is allocated 1 count of f1 and 8 counts of feature f3 from the back office. When processing the new reservation group, the license server does the following:

- Determines that a total of 1 count of f1 is required (for device 111A11AAA11A).
- Determines that a total of 6 counts of f3 is required—**2** counts for device 111A11AAA11A, **1** count for device 222A22AAA22A, and **3** counts for user Joe.
- Reserves **6** counts of f3 and **1** count of f1.
- Keeps **2** unreserved counts of f3 available for any request on a first-come-first-served basis.

Example 2: Insufficient Feature Counts

If a license server is allocated 1 count of f1 and 4 counts of f3, it does the following when the reservations are posted:

- Determines that a total of 1 count of f1 is required (for device 111A11AAA11A).
- Determines that a total of 6 counts of f3 are required—**2** counts for device 111A11AAA11A, **1** count for device 222A22AAA22A, and **3** counts for user Joe.
- Rejects the entire reservation group since only 4 counts of f3 are available.

Example 3: Features Not in the License Pool

If a license server is allocated no (0) counts of f1 but is allocated 8 counts of f3, it rejects the entire reservation group since f1 is not available in the license pool.

When Adding Reservations to an Existing Reservation Group

If you are adding a reservation to an existing reservation group or are adding reservation entries to an existing reservation, the basic allocation rules used to add a reservation group apply:

- If you are adding a reservation to an existing reservation group, the license server must be able to satisfy the total feature counts requested for *all* reservation entries in the reservation. If total feature counts cannot be satisfied, the entire reservation is rejected.
- If you are adding a single reservation entry to an existing reservation, the license server must be able to satisfy the feature count requested for the new entry. If the feature count cannot be satisfied, the new reservation entry is rejected.
- If you are adding an array of reservation entries to an existing reservation, the license server must be able to satisfy the total feature counts requested for *all* the reservation entries in the array. If any of these total counts cannot be satisfied, the entire array of reservation entries is rejected. (Existing reservation entries remain unaffected.)

When Feature Counts Change on the Server

When feature counts are reduced or expire on the license server, the server automatically disables certain reservation entries to adjust to the new feature count. The server uses no particular order as it processes the reservation groups, reservations, or reservation entries to determine which reservation entries to disable.

Processing the Capability Request When Reservations Are Used

The following sections describe how the license server processes capability requests to grant licenses when reservations are available:

- Basic Process for Granting Licenses
- Example Scenarios: Basic Process for Granting Licenses

Basic Process for Granting Licenses

When reservations have been added and the license server receives a capability request from a client device, the server uses the following basic process in its attempt to grant the license count requested.

***Note** ▪ This process described here assumes that the capability request is not defined as "incremental" and that no desired feature is enabled for partial checkout in the request. For a description of the effect that these options have on granting licenses, see the appendix Effects of Special Request Options on Use of Reservations.)*

1. Validates that the capability request contains the appropriate content for processing.
2. Returns all licenses currently served to the client device. (Reserved licenses remain reserved. Shared licenses are returned to the pool.)
3. Searches for reservations assigned to the client device and to the user sending the request.

4. Determines whether desired features are specified in the request:

   If no desired features are requested, the server grants the client device whatever licenses have been reserved for it through device-based reservations and, if a secondary hostid is included in the capability request, through user-based reservations. If no reservations exist, no licenses are granted.

   If a desired feature is specified, the server determines whether the specified feature is available through a reservation. If not, it determines whether any shared counts are available. In summary, the server does the following:

   a. Checks for the device-based reserved count first.

   b. If device-based reserved count is not sufficient, checks the user-based reserved count (if the secondary hostid is sent in the capability request) for an additional count.

   c. If the sum of device-based and user-based reserved counts is not sufficient, checks the available shared counts to satisfy the remaining requested count.

   d. If enough reserved and shared counts are available, grants the licenses.

Example Scenarios: Basic Process for Granting Licenses

The following scenarios illustrate instances of the basic process (described in the previous section) that the license server uses to grant licenses when reservations are involved. The sections describing the scenarios in include the following:

- Feature Allocation on the License Server
- Starting State for Each Scenario
- Scenario 1—No Device or User Reservations Available
- Scenario 2—Device Reservations But No User Reservations Available
- Scenario 3—User Reservations But No Device Reservations Available
- Scenario 4—Device and User Reservations Available

For the sake of simplicity, the feature version is omitted in these scenarios. Anytime feature F1 is mentioned, assume that its version is 1.0.

Feature Allocation on the License Server

The license server has 10 counts of feature F1:

- 3 reserved counts for device D1
- 2 reserved counts for user U1
- 5 shared counts available

Starting State for Each Scenario

The starting state for each scenario is a follows:

- Device D1 has been served 4 counts (all 3 reserved counts, 1 shared count).
- Device D2 has been served 3 counts (3 shared counts).

- User U1 currently has no served counts.
- 1 shared count remains available.

Scenario 1—No Device or User Reservations Available

When user U7 (who has no reservations) sends a capability request from device D2 (which also has no reservations), the 3 counts of F1 currently served to D2 are returned to the shared counts. (Four shared counts are now available.)

The following describes possible scenarios of what happens next:

- If the request has no desired features, no counts are served since neither D2 nor U7 has reserved counts.
- If the request specifies 4 counts of F1, the remaining 4 available shared counts are served.
- If the request specifies 5 counts of F1, no counts of F1 are sent in the response since only 4 shared counts are available.

Scenario 2—Device Reservations But No User Reservations Available

When user U7 (who has no reservations) sends a capability request from device D1, the 4 counts of F1 currently served to D1 (3 reserved counts for D1 and 1 shared count) are returned. (Two shared counts are now available.)

The following describes possible scenarios of what happens next:

- If the request has no desired features, the 3 counts of F1 reserved for D1 are served.
- If the request specifies 4 counts of F1, the 3 counts reserved for D1 plus 1 shared count are served.
- If the request specifies 6 counts of F1, the 3 counts reserved for D1 and the remaining 2 shared counts are insufficient to satisfy the request, so no counts of F1 are sent in the response.

Scenario 3—User Reservations But No Device Reservations Available

If user U1 sends a capability request from device D2 (which has no reservations), the 3 counts of F1 currently served to D2 are returned to shared counts. (Four shared counts are now available.)

The following describes possible scenarios of what happens next:

- If the request has no desired features, the 2 counts of F1 reserved for U1 are served.
- If the request specifies 4 counts of F1, the 2 counts reserved for U1 plus 2 shared counts are served.
- If the request specifies 7 counts of F1, the two counts reserved for U1 and the remaining 4 shared counts are not sufficient to satisfy the request, so no counts of F1 are sent in the response.

Scenario 4—Device and User Reservations Available

If user U1 sends a capability request from device D1, the 4 counts of F1 currently served to D1 (3 reserved and 1 shared) are returned. (Two shared counts are now available.)

The following describes possible scenarios of what happens next:

- If the request has no desired features, all reserved features—3 counts of F1 for D1 and 2 counts for U1—are served.

- If the requests specifies 4 counts of `F1`, the 3 counts reserved for `D1` and 1 of the 2 counts reserved for `U1` are served.
- If the request specifies 8 counts of `F1`, the 3 counts reserved for `D1`, the 2 counts reserved for `U1`, and the 2 remaining shared counts are not sufficient to satisfy the request, so no counts are sent in the response.

Reservation Considerations and Limitations

Note the following about reservations:

- A license server administrator can use either reservations or named license pools on a license server. Reservations and named license pools cannot coexist alongside each other. For more information, see Named License Pools vs. Reservations.For general information about named license pools, see Allocating Licenses Using Named License Pools.
- Reservations support only concurrent features; metered features cannot be reserved.
- Capability requests that specify the same hostid as a main hostid in one request and as a secondary hostid in another are rejected.
- If you need to reset trusted storage on a license server, note that all server data, including reservations, are deleted. Therefore, make sure that the license server administrator backs up all reservation information to a file before the trusted-storage reset is performed. The reservations can then be restored once the reset has taken place. The administrator can use the FlexNet License Server Administrator command-line tool to perform both the backup and the restoration. See the *FlexNet Embedded License Server Administration Guide* for details.
- The license server failover configuration does not support reservations.

6

Advanced License Server Features

This chapter provides background information about special features that you can enable for your FlexNet Embedded local license server and that might require additional setup to be performed by you or the license server administrator at the enterprise customer site.

Some of these features are also supported by the CLS instance, and are noted as such in this chapter.

- Producer-defined Hostids
- Secure Anchoring
- Clone Detection Reporting
- Binding-Break Detection with Grace Period
- Trusted Storage Backup and Restoration
- Outgoing HTTPS
- Incoming HTTPS
- Proxy Support for Communication with the Back Office
- Extended Hostids
- Trusted-Storage Reset
- License Checkout: Support for Special Capability-Request Options
- Usage Capture and Management
- Administrative Security

Producer-defined Hostids

When the FlexNet Embedded local license server communicates with the back office, FlexNet Operations, to obtain its pool of licenses, the back office uses the server's hostid to identify that particular server. This kind of standard server hostid is typically the server system's Ethernet address or can be an attached dongle ID or VM UUID.

In addition to these standard hostid types, you can optionally specify a *producer-defined hostid* value to identify the server to the back office.

The following sections describe producer-defined hostids:

- Configuring Support for the Producer-defined Hostid
- Providing the Shared Library for Hostid Retrieval
- Using the Producer-defined Hostid

Configuring Support for the Producer-defined Hostid

To use a producer-defined hostid, you must enable support for this feature on the back office and the license server.

Enable Support on the Back Office

Check your FlexNet Operations documentation for information about enabling support for producer-defined hostids.

Enable Support on the License Server

To enable support for producer-defined hostids on the license server, you must set the `server.publisherDefinedHostId.policy` to **strict** in the policy settings file (`producer-settings.xml`) file:

```
server.publisherDefinedHostId.policy=strict
```

By default, this parameter is set to `disabled`.

You use the FlexNet Embedded local license server configuration utility to edit this setting. For more information, see License Server Configuration Utility in Chapter 7, Producer Tools.

Providing the Shared Library for Hostid Retrieval

Use the following process to create the shared library that retrieves the producer-defined hostid:

- Step 1: Implement Hostid Retrieval Code
- Step 2: Build the Shared Library
- Step 3: Distribute the Shared Library

If you are also configuring the license server to use the binding-break detection feature, the same shared library you create to retrieve binding elements for the detection feature is also used to retrieve your producer-defined hostid. See Using the Same Library to Retrieve Binding Elements for more information.

Step 1: Implement Hostid Retrieval Code

The license server software package provides the following example C interface header file and code implementation for retrieving the producer-defined hostid. These files are located in directories under the `example` directory in the software package.

- `ExamplePublisherDefinedHostId.c` (found in the `publisher_defined_hostid` directory)
- `PublisherDefinedExtensionExports.h` (found in the `include` directory)

You can use these examples as the basis for your implementation; or you can create your own implementation in C, C++, or asm. As long as you keep the interface to the shared library (as provided in the examples) intact, the license server will be able to load the shared library.

The following shows the interface required in your implementation. Refer to the example code for a complete sample implementation.

```
FlxServerExtensionRef FlxServerExtensionCreate( void );

const char * FlxServerExtensionGetHostId( FlxServerExtensionRef extension );

void FlxServerExtensionDelete( FlxServerExtensionRef extension );

int FlxServerExtensionGetLastError( FlxServerExtensionRef extension )
```

***Note** • The FlexNet Embedded local license server does not support the Windows COM interface for the shared library.*

Hostid Specifications

Instead of returning a hard-coded value, your code typically reads a system characteristic—for example, a hardware serial number—and returns that value. When using a producer-defined hostid, try to use a value that cannot easily be spoofed.

The return value for `FNE_SERVER_PUBLISHER_HOSTID_STRING` must follow the standard hostid-value rules, such as not containing any white space. The value also cannot contain commas. The following special characters are acceptable in the hostid: underscore (_), period (.), hyphen (-), ampersand (&), dollar sign ($), “at” sign (@), hash (#), colon (:).

If the return value is null or an empty string, an error is generated. In this situation, the license server cannot serve licenses because no valid active hostid exists. Make sure the return value is a valid hostid.

Step 2: Build the Shared Library

Use gcc/clang on Linux or Visual Studio on Windows to build the shared library. The compiler version you use is your choice.

It is critical that the shared library uses the name listed here for the specific platform:

Table 6-1 ▪ Required Name for the Shared Library per Operating System

OS Platform	Name
Windows (JVM 64-bit)	`ServerExtension64.dll`
Windows (JVM 32-bit)	`ServerExtension.dll`
Linux (JVM 64-bit)	`libServerExtension64.so`
Linux (JVM 32-bit)	`libServerExtension.so`

Step 3: Distribute the Shared Library

Once you have built the shared library for retrieval of the producer-defined hostid, create a subdirectory with your producer name in the same directory where the `flexnetls.jar` will reside in the enterprise customer's license server installation. Then copy the shared library there. For example, if the `flexnetls.jar` file will reside in the root installation directory `c:\FNELicenseServer` and your producer name is "companyA", create the directory `c:\FNELicenseServer\companyA`. Place the shared library there.

Using the Same Library to Retrieve Binding Elements

If you are also using the binding-break detection feature (as described in Binding-Break Detection with Grace Period), the same shared library you create to retrieve binding elements for the detection feature is also used to retrieve your producer-defined hostid. To create a shared library that supports the combined functionality, incorporate this interface in your code:

```
FlxServerExtensionCreate

FlxServerExtensionGetHostid

FlxServerExtensionGetBindingIdentity

FlxServerExtensionGetLastError
```

Use the sample C code implementations found in the `example` directory in the license server software package as a guide for creating code that retrieves both a producer-defined hostid and the binding elements.

Using the Producer-defined Hostid

Keep these points in mind when using producer-defined hostids:

- Using a producer-defined hostid results in other hostid types (Ethernet, dongle, or VM UUID) being unavailable for the license server.
- When using the manual testing tools provided in (`bin\tools` in your license server software package) to generate capability responses that contain producer-defined hostids, use the syntax `PUBLISHER_DEFINED=`*`value`* in the text license file used as input to `capresponseutil`, along with the hostid-type switch `-idtype publisher_defined`.

Secure Anchoring

The FlexNet Embedded local license server 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 the license server. While default anchoring stores the anchoring information in the anchor file, secure anchoring uses additional techniques to store anchor information on the target system.

The following sections describe secure anchoring:

- Prerequisites
- Enabling a Secure Profile

Prerequisites

The operating systems supported by the FlexNet Embedded local license server 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, the license server seamlessly defaults to a reduced set of anchoring techniques for that system.

Additionally, you need the binary file containing your producer-identity data for the license server (for example, `ClientServerIdentity.bin`). This identity data is generated—along with your identity data files for the back office and the FlexNet Embedded client—by the back office itself or by the example `pubidutil` utility provided in your license server software package (see Publisher Identity Utility in the Chapter 7, Producer Tools.)

Enabling a Secure Profile

After you have obtained the file containing the server-identity binary data, 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 `server-medium`.) A typical command is:

```
secureprofileutil -profile server-medium IdentityClientServer.bin IdentityClientServerSecure.bin
```

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.

Clone Detection Reporting

FlexNet Embedded provides a mechanism that detects when your licensed product or an enterprise local license server (that is, the FlexNet Embedded local license server) might be running on a cloned machine—virtual or physical—and provides this information to FlexNet Operations, where you can generate a report about the clone suspects.

Note that this clone detection feature simply reports on potential clones; it does not take any action on clone activity.

For the most part, you can ignore FlexNet Embedded clients or license servers that this feature infrequently flags as clone suspects. For example, if a FlexNet Embedded client accidentally gets disconnected during license checkout from the license server, the client might be marked as a clone suspect the next time it connects with the back office or license server, even though no cloned environment exists. However, if a client or license server is persistently identified as a clone suspect, you might want to investigate further with the customer and take action if necessary.

The following sections describe this clone detection feature:

- About the Clone Detection Process
- Requirements
- Generate the Clone Detection Report in FlexNet Operations
- Report Alternatives
- Considerations and Limitations

About the Clone Detection Process

The clone detection mechanism is found in the following components:

- The license server (FlexNet Embedded local license server or CLS license server) to detect served FlexNet Embedded clients running on cloned devices
- The back office (FlexNet Operations) to detect FlexNet Embedded clients and FlexNet Embedded local license servers running on possible cloned devices

If a possible cloned device is detected, the FlexNet Embedded client or the FlexNet Embedded local license server running on that device is flagged as a "clone suspect". Information about this clone suspect is ultimately stored in the back office, as described next.

License-Server Detection of Possible Cloned Client

If the license server detects that a possible client device to which it has served licenses is a clone, it records a "clone suspect" flag in the client record on the server. This flag is then included in the synchronization message to the back office, where this "clone suspect" information is recorded for reporting purposes.

Back-Office Detection of Possible Cloned Client or Local License Server

If the back office detects a possible cloned device running a FlexNet Embedded local license server or a FlexNet Embedded client that obtains its licenses directly from the back office, it stores this "clone suspect" information for reporting purposes.

Requirements

Make sure that the following requirements are met to enable clone detection reporting:

- Capability Exchange Required for Clone Detection
- Synchronization Enabled from the License Server to the Back Office
- Back Office Enabled to Record Clone Detection Information

Capability Exchange Required for Clone Detection

Clone detection occurs only when a capability exchange takes place. (These exchanges require communications between the FlexNet Embedded client or the license server and the back office—that is, FlexNet Operations.) By design, the FlexNet Embedded local license server always obtains its licenses from online or offline capability exchanges with the back office. However, you must ensure that a FlexNet Embedded client obtains its licenses through online or offline capability exchanges with the back office or with a license server, not through another means such as a license file. (For information about the API used on the client to retrieve its clone detection status from a capability response, refer to the *Using the FlexNet Embedded APIs* chapter in your FlexNet Embedded client SDK user guide.)

Synchronization Enabled from the License Server to the Back Office

To ensure that the back office receives information about possible client device clones at your enterprise customer sites, configure the license server to perform online or offline synchronization to the back office. (Note that online synchronization ensures that any captured "clone suspect" information is sent on a timely basis to the back office.) For information about performing synchronization, see Online Synchronization to the Back Office in Chapter 5, More About Basic License Server Functionality.

Back Office Enabled to Record Clone Detection Information

As a prerequisite for generating the back office reports describing possible clones, enable the recording of "clone suspect" information in the back office. To do so, in the FlexNet Operations Producer Portal, navigate to the Embedded Devices tab of the System Configuration page, and select the **Clone Detection** option. If necessary, refer to the Producer Portal help system available with FlexNet Operations for additional details.

If this option is not enabled, "clone suspect" information is not recorded in the back office and therefore not available for reports.

Generate the Clone Detection Report in FlexNet Operations

FlexNet Operations Web Services provides the API `generateCloneDetectionReport` to extract the information stored in the back office about the detected clone suspects and generate a report listing this information. (Each clone entry provides the device ID, client type—client, served client, or license server—and other details.) For details about accessing this API and integrating it in your business code to obtain the report as needed, refer to the *FlexNet Operations Web Services Guide* and the *FlexNet Operations Web Services Integration Guide*.

Report Alternatives

Aside from the generation of reports in the back office, the clone detection feature provides other ways to help you view the detected clone suspects:

- Obtain Clone Status in Capability Response
- View History Event Through Producer Portal

Obtain Clone Status in Capability Response

The capability response sent to a FlexNet Embedded client (whether from a license server or the back office) includes the client's current clone detection status. FlexNet Embedded provides functionality that you can incorporate in the client code to retrieve this status from the response. See the *Using the FlexNet Embedded APIs* chapter in your FlexNet Embedded client SDK user guide for more information.

View History Event Through Producer Portal

When the clone detection mechanism determines a clone suspect, the information is stored in the back office as a history event for the specific FlexNet Embedded client (called "device"), served client, or license server. If the client or license server is a clone suspect, its history, as accessed through the Producer Portal, lists the following event:

```
The capability request might have been out of order
```

For security purposes, this event description does not explicitly state that the given client or license server is a clone suspect, as histories for license servers and non-served clients can be viewed through the End User Portal as well.

Considerations and Limitations

Note the following about clone detection reporting:

- This feature is compatible with capability requests defined with the *request* and *undo* operation types only. It is not compatible with the *report* operation type.
- Should a potential clone be detected, this feature does not take any action to deny capability exchanges with the "suspect" FlexNet Embedded client or license server, nor does it send you any alerts. The only way to view clone information is through back-office reports (see Generate the Clone Detection Report in FlexNet Operations) or through your retrieval of this information from individual client installations (see Report Alternatives). It is your responsibility to decide how to act on the gathered clone detection information.

Binding-Break Detection with Grace Period

During normal licensing operations at an enterprise customer site, FlexNet Operations provides the FlexNet Embedded local license server with features that are bound to a *unique device hostid* identifying the machine on which the server is running. (This hostid is typically the server system's Ethernet address or can be a VM UUID or the ID of an attached dongle ID.) In turn, the license server distributes these "bound" features to FlexNet Embedded clients requesting them. Should the device hostid of the license server change, the binding becomes broken, and the server is considered to be in a state of a "hard binding break" and unable to serve licenses.

Tip *▪ For step-by-step instructions that walk you through setting up binding on a local license server, see the appendix Workflow Example for Producer-Defined Binding. The appendix also simulates a binding break and explains how to repair the break.*

Problem with Maintenance Events

Maintenance situations can arise when the license server administrator needs to change the environment in which the license server is running. For example, the administrator might need to replace or upgrade the hardware on which the server runs or, in a virtual environment, restore a machine image if the primary image is corrupted or move the virtual machine across hardware to optimize performance. Because a change in environment can cause the hostid binding to break, the license server administrator most likely attempts to maintain the same hostid in the new environment to avoid any workflow disruptions. However, when the same hostid is maintained but no environment change is detected, you are at risk for license-leakage.

Solution

FlexNet Embedded provides an additional binding-break detection feature which locks license rights of a local license server to the machine on which the server is running. This feature helps to ensure compliance by preventing license rights being transferred to another system.

Specifically, the binding-break detection does the following:

- Perceives an environment change in the license server even when the binding hostid remains the same. It does this through the use of system attributes that you select as binding elements.
- Enables you to configure a grace period before any action is taken against licensing in the changed environment. This approach gives the license server administrator time to plan for a repair, as well as provides you with a level of protection against any license leakage.
- Reports the binding-break status in the server's log, in capability responses sent to FlexNet Embedded clients, in capability requests sent to the back office, and through the license server's binding REST API.

Tip ▪ *The binding-break detection feature is particularly useful for license servers running in an air-gapped environment. For license servers that run with back-office support (FlexNet Operations), Revenera recommends to use the clone detection feature. For more information, see* *Clone Detection Reporting**.*

Separate from Normal Binding-Break Checks

This producer-defined binding-break detection feature is separate from FlexNet Embedded's normal check for breakage in the server's device hostid binding. Whether the producer-defined binding-break detection feature is enabled or disabled, the normal checks for breaks in the device hostid binding on the server continue.

For More Information

For more information about configuring and using the binding-detection feature, see the following:

- Enabling the Binding-Break Detection Feature
- Providing the Shared Library to Retrieve the Binding Elements
- How the Binding-Break Detection Works
- Repairing the Broken Binding
- Considerations and Limitations

Enabling the Binding-Break Detection Feature

To enable producer-defined binding-break detection, you need to add a policy setting called `binding.break.policy` to the `producer-settings.xml` file for the license server. This policy uses one of the following values to define the action taken when the binding-break detection feature perceives a break:

- **hard**—An immediate hard binding break is imposed—that is, the license server can no longer serve licenses.
- **soft**—A soft binding break is in effect, allowing the license server to continue to serve licenses despite the break.
- **interval**—A soft binding break with a grace period is in effect, allowing the license server to continue to serve licenses until the grace period expires. When the expiration occurs, a hard binding break goes into immediate effect, and licenses can no longer be served.

By default, this policy is not automatically included in `producer-settings.xml`; hence, the binding-break detection feature is disabled unless you add the policy.

Example Policy

The following shows an example `binding.break.policy` entry to add to `producer-settings.xml`. This policy setting specifies that an initial soft binding break with a grace period of 86400 seconds (1 day) is put into effect once a binding break is detected. (For the grace period, you can also specify an equivalent value in minutes, hours, or weeks, as in `1d` or `1440h` for 1 day.) When the grace period expires, a hard binding break goes in effect.

```
binding.break.policy=86400
```

For details about providing a value for this policy, see Reference: Policy Settings for the License Server appendix. For instructions on adding the policy to `producer-settings.xml`, see License Server Configuration Utility in Chapter 7, Producer Tools.

Once you add the `binding.break.policy` setting, it is not visible in the actual `producer-settings.xml` file for security purposes.

Providing the Shared Library to Retrieve the Binding Elements

Use the following process to create the shared library that retrieves the system attributes to be used as binding elements:

- Step 1: Determine the Binding Elements
- Step 2: Implement Code to Retrieve the Binding Elements
- Step 3: Build the Shared Library
- Step 4: Distribute the Library with the License Server

The license server calls this shared library to retrieve the binding information whenever the server starts up or attempts to reset the binding per instructions from the back office (see Repairing the Broken Binding).

If you are also configuring the license server to use a producer-defined hostid, the same shared library you create to retrieve your producer-defined hostid is also used to retrieve your binding elements. See Using the Same Library to Retrieve a Producer-defined Hostid for more information.

Step 1: Determine the Binding Elements

The choice of system attributes that you select as binding elements in the binding-break detection feature should be attributes that *are* likely to change if the license server's environment changes, but that are *not* easily retained between environments.

The recommendation is to use a a combination of system attributes (the Ethernet address, VM type, the SMBIOS UUID, and such). If any one of these attributes change, the binding data sent to the license server reflects this change, and a binding break is detected.

Step 2: Implement Code to Retrieve the Binding Elements

You need to implement code that retrieves the system attributes that you want to use as binding elements. You then build the code as a shared library that you distribute with the license server.

The license server software package provides the following example C interface header file and code implementation for retrieving the system attributes and returning them as binding data to the server. These files are located in directories under the `example` directory in the software package:

- `ExamplePublisherDefinedBinding.c` (found in the `publisher_defined_binding_identity` directory)
- `PublisherDefinedExtensionExports.h` (found in the `include` directory)

You can use these examples as the basis for your implementation; or you can create your own implementation in C, C++, or asm. As long as you keep the interface to the shared library intact (as provided in the examples), the license server will be able to load the shared library.

The following shows the basic interface required in your implementation. Refer to the example code for a complete sample implementation.

```
FlxServerExtensionRef FlxServerExtensionCreate( void );

unsigned int FlxServerExtensionGetBindingIdentity(
                    FlxServerExtensionRef extension,
                    FxServerBindingContextRef context,
                    FlxProvideBindingInfoFn provideBindingInfo );

void FlxServerExtensionDelete( FlxServerExtensionRef extension );

int FlxServerExtensionGetLastError( FlxServerExtensionRef extension )
```

***Note** ▪ The FlexNet Embedded local license server does not support the Windows COM interface for the shared library.*

Binding Data Specifications and Considerations

Note the following:

- The binding data returned to the license server must be in byte array format and not exceed 512 bytes.
- If the initial return value for the shared library is null or an empty string, no binding break is triggered. However, if in subsequent server startups, the library starts to return binding data, the license server is considered to be in a state of binding breakage.

- If the shared library is unable to retrieve binding information, a hard binding break goes into effect.

Step 3: Build the Shared Library

Use gcc/clang on Linux or Visual Studio on Windows to build the shared library. The compiler version you use is your choice.

It is critical that the shared library uses the name listed here for the specific platform:

Table 6-2 ▪ Required Name for the Shared Library per Operating System

OS Platform	Name
Windows (JVM 64-bit)	`ServerExtension64.dll`
Windows (JVM 32-bit)	`ServerExtension.dll`
Linux (JVM 64-bit)	`libServerExtension64.so`
Linux (JVM 32-bit)	`libServerExtension.so`

Step 4: Distribute the Library with the License Server

Once you have built the shared library for retrieval of the binding elements, create a subdirectory with your producer name in the same directory where the `flexnetls.jar` will reside in the enterprise customer's license server installation. Then copy the shared library there. For example, if the `flexnetls.jar` file will reside in the root installation directory `c:\FNELicenseServer` and your producer name is "companyA", create the directory `c:\FNELicenseServer\companyA`. Place the shared library there.

Using the Same Library to Retrieve a Producer-defined Hostid

If you are also using a producer-defined hostid (as described in Some of these features are also supported by the CLS instance, and are noted as such in this chapter.), the same shared library you create to retrieve the producer-defined hostid is also used to retrieve the binding elements for producer-defined binding-break detection. To create a shared library that supports the combined functionality, incorporate this basic interface in your code:

```
FlxServerExtensionCreate

FlxServerExtensionGetHostid

FlxServerExtensionGetBindingIdentity

FlxServerExtensionGetLastError
```

Use the sample C code implementations found in the `example` directory in the license server software package as a guide for creating code that retrieves both the binding elements and a producer-defined hostid.

How the Binding-Break Detection Works

The following provides an overview of how the producer-defined binding-break detection approach works after you have configured the license server with the appropriate binding-break policy and provided the shared library:

- At Server Startup
- During Normal Server Operations
- During the Grace Period for the Binding Break
- When the Grace Period for the Binding Break Expires
- When the Binding Is Restored
- Reporting a Binding Break to the Back Office

At Server Startup

The first time that the license server starts up, it calls the shared library to retrieve the binding data from server environment and persists this data in trusted storage.

Each time thereafter when the server starts up, the binding data is retrieved and compared with the data persisting in trusted storage. If no change exists in the binding data, the license server continues to run normally. However, if a change in binding data is detected, the binding break policy you set goes into effect (see Enabling the Binding-Break Detection Feature).

You, the enterprise license server administrator, and the FlexNet Embedded clients can use various methods to view the license server's current binding status.

About the "binding" REST API

As producer, you can access the FlexNet Embedded local license server `/binding` REST API to view the binding policy and current binding status for a given enterprise license server at any time. You might want to incorporate this API in your custom license server administrator tool so that the enterprise license server administrator can view this information as well.

The `-binding` command option in the FlexNet License Server Administrator command-line tool also accesses this API. For details, see the "Using the FlexNet License Server Administrator Command-line Tool" chapter in the *FlexNet Embedded License Server Administration Guide*.

The following sections shows the information retrieved by the `/binding` API, depending on the license server's binding status. For more information about this REST API, see Chapter 4, License Server REST APIs.

During Normal Server Operations

When no binding break is detected at server startup, the `/binding` REST API returns an "ok" status with results similar to this example, depending on the setting for the binding-break policy:

```
{
  "bindingPolicy" : "86400 seconds",
  "bindingStatus" : "ok"
}
```

See About the “binding” REST API for information about how the /binding REST API is used in the binding-break detection feature.

No other binding status information is reported to the back office or FlexNet Embedded clients.

During a Soft or Hard Binding Break

If a binding break is detected at server startup and the license server is configured with a “hard” or “soft” binding break policy, the license server administrator and FlexNet Embedded clients are alerted of the break status in the following ways.

View License Server Log

The license server administrator can view the current binding status in the log displayed when the license server starts.

The following example log shows the status when a soft binding break occurs. (The license server continues to serve licenses and operate normally).

```
server>flexnetls.bat
 15:44:45,848 INFO  Starting FlexNet License Server 2018.12 (build 244007)
 15:44:45,850 INFO  Copyright (C) 2013-2018 Flexera.
 15:44:45,851 INFO  All Rights Reserved.
 15:44:45,852 INFO  Running as a console session
 15:44:61,852 WARN  Soft binding break detected at startup.
```

If a hard binding break occurs, the log shows the following status instead, as the license server is no longer able to serve licenses:

```
ERROR: Features will not be available due to a hard binding break.
```

Access “binding” REST API

The /binding REST API shows the server’s binding status as “soft break” or “hard break”, depending on the binding-break policy:

```
{
  "bindingPolicy" : "soft",
  "bindingStatus" : "soft break"
}
```

See About the “binding” REST API for information about how you can use the /binding REST API to alert enterprise license server administrators about the current binding status.

View Capability Response on the Client

The capability response sent to a FlexNet Embedded client includes the appropriate status code. If a soft binding break occurs, the response includes the licenses requested by the client (since the server can continue to serve licenses) and provides the following status code specifying the "soft" value:

```
FLX_MS_CODE_SERVER_BINDING_BREAK_DETECTED, "soft"
```

If a hard binding break occurs, the response includes no requested features (since the server can no longer serve licenses) and shows the provides the following status code specifying the "hard" value:

```
FLX_MS_CODE_SERVER_BINDING_BREAK_DETECTED, "hard"
```

Your client code can retrieve this status information from the response to send the client an alert. Upon receiving the alert, the client can contact the license server administrator about breakage status.

During the Grace Period for the Binding Break

If a binding break is detected when the license server starts up and you have configured a grace period for the binding-break policy, the server immediately enters into a soft binding break state and continues to serve licenses until the grace period ends.

The license server administrator and client devices are alerted of the server's initial "soft break" status in the various ways described in this section.

If, during the grace period, the current binding data once again matches the binding data in trusted storage, the license server is removed from the "soft break" state and resumes normal operations. See When the Binding Is Restored.

View License Server Log

The license server administrator can view the current binding status in the log that is displayed when the license server starts, as shown in this example:

```
server>flexnetls.bat
 15:44:45,848 INFO  Starting FlexNet License Server 2018.12 (build 244007)
 15:44:45,850 INFO  Copyright (C) 2013-2018 Flexera.
 15:44:45,851 INFO  All Rights Reserved.
 15:44:45,852 INFO  Running as a console session
 15:44:61,852 WARN  Soft binding break detected at startup.
```

Access REST API

The `/binding` REST API lists the server's binding policy and the current binding status, along with the grace period expiration time (in ISO-8601 format), as shown in this example:


```
{
  "bindingPolicy" : "86400 seconds",
  "bindingStatus" : "soft break",
  "gracePeriodExpiry" : "2019-02-15T16:59:52Z"
}
```


See About the "binding" REST API for information about how you can use the `/binding` REST API to alert enterprise license server administrators about the current binding status.

View Capability Response on the Client

The capability response sent to clients includes the licenses requested by the given client (since the server can continue to serve licenses during the grace period) and provides the following status code, where *n* represents the grace period's expiration timestamp in epoch (also called "Unix") format:

```
FLX_MS_CODE_SERVER_BINDING_BREAK_DETECTED, "<n seconds>"
```

Your client code can retrieve this status information from the response to send the client an alert. Upon receiving the alert, the client can contact the license server administrator about breakage status.

When the Grace Period for the Binding Break Expires

When a grace period expires, the license server goes into a hard binding break and can no longer serve licenses. At this point, the license server administrator and FlexNet Embedded client devices are informed of the expiration of the server's grace period through various means.

View License Server Log

The license server administrator can view the current binding status in the log that is displayed when the license server starts:

```
server>flexnetls.bat
 15:44:45,848 INFO  Starting FlexNet License Server 2018.12 (build 244007)
 15:44:45,850 INFO  Copyright (C) 2013-2018 Flexera.
 15:44:45,851 INFO  All Rights Reserved.
 15:44:45,852 INFO  Running as a console session
 ERROR: Features will not be available due to a hard binding break.
```

Access REST API

The `/binding` REST API lists the server's binding policy and shows the binding status as a "hard break" due to the expired grace period (in ISO-8601 format), as shown in this example:

```
{
  "bindingPolicy" : "86400 seconds",
  "bindingStatus" : "hard break",
  "gracePeriodExpiry" : "2019-02-15T16:59:52Z"
}
```

See About the "binding" REST API for information about how you can use the `/binding` REST API to alert enterprise license server administrators about the current binding status.

View Capability Response on the Client

The capability response sent to clients includes no requested features (since the server can no longer serve licenses) and provides the following status code, specifying that grace period has expired. The `0` value (always provided with this status code) indicates that the server has entered into a hard break from the soft break.

```
FLX_MS_CODE_SERVER_BINDING_GRACE_PERIOD_EXPIRED, "0"
```

Your client code can retrieve this status information to send the client an alert. Upon receiving the alert, the client can contact the license server administrator about breakage status.

When the Binding Is Restored

If, during the grace period or after its expiration, the current binding information once again matches the binding information in trusted storage (for example, through a broken-binding repair or restoration of the original machine), the license server is restored to a normal state and can resume normal licensing operations.

At this point, the status information described in At Server Startup is again in effect.

For information about how to repair a broken binding, see Repairing the Broken Binding.

Reporting a Binding Break to the Back Office

When the license server experiences a binding break, subsequent capability requests originating from the server conveys information about the breakage. If the server is running under a grace period, this information includes the expiration date in the epoch (Unix) timestamp format. The back office can then determine how to handle breakage (for example, by sending a “reset binding” flag in the capability response to the server, as described in Repairing the Broken Binding).

Viewing Breakage Events on FlexNet Operations Portals

When a capability request sent to the back office indicates that the license server is experiencing a binding break, you can view the breakage event in the FlexNet Operations Producer Portal on the server’s View Host Requests and View Host History pages. Likewise, the license server administrator can view the event in the server’s history in the End User Portal. For more information, refer to the help system for either FlexNet Operations portal.

Repairing the Broken Binding

Once alerted about the binding break, the license server can send a capability request containing the binding-break status to the back office, which, in return, sends a “reset binding” flag in the capability response to the server. When the server processes the response, the server’s trusted storage is reset. When the license server is restarted, the new binding data is persisted in trusted storage.

This reset process should have no effect on the existing device hostid used by the license server.

Considerations and Limitations

Note the following about the producer-defined binding-break detection feature:

- This feature is for the FlexNet Embedded local license server only.
- Checks for binding breaks occur only when the license server starts up.

Trusted Storage Backup and Restoration

The FlexNet Embedded local license can be enabled to back up its trusted storage at designated times. Should trusted storage become corrupted, the license server administrator can restore the last backup without having to contact FlexNet Operations. The following sections describe the trusted-storage backup and restoration process and other implications:

- Performance Implications
- Enabling Backups of Trusted Storage
- Overview of the Backup Process
- Running a Trusted Storage Restoration

Performance Implications

Regular trusted-storage backups can negatively affect the local license server's performance. Revenera therefore recommends to disable trusted-storage backups (setting `database.backup-enabled=false`) in setups where the license server exhibits performance issues. In scenarios where no trusted-storage backup is available, license administrators can rely on the back office (FlexNet Operations) to be the source of truth for license counts.

Enabling Backups of Trusted Storage

To enable backups of trusted storage on the license server, set the `database.backup-enabled` policy to `true` in `producer-settings.xml`. (By default, this policy is `false`.) For instructions on updating policies in `producer-setting.xml`, see License Server Configuration Utility in Chapter 7, Producer Tools.

Overview of the Backup Process

The following provides an overview of the backup process for trusted storage on the license server:

- **One.** The license server automatically initiates a trusted-storage backup whenever the following events occur:
    - A capability response from the back office is processed.
    - A quantity of capability requests from FlexNet Embedded clients have been processed. (This quantity is based on anchor thresholds you define, as described in Appendix A, Reference: Policy Settings for the License Server.)
- **Two.** Once the backup is initiated, the license server suspends the processing of capability responses and requests until the backup completes.
- **Three.** If the backup is successful, the backed-up trusted storage data is stored in the file `tsBackup/flexnetls_licenses.mv.db.ts`, located in the directory containing trusted storage. (A failed backup can be an indicator that trusted storage is corrupt.)

  The license server maintains only one copy of the backed-up data at any given time.

Running a Trusted Storage Restoration

If trusted storage is determined to be corrupt, the license server administrator can use these steps to restore the last successful backup of trusted storage.

Task ***To restore trusted storage from the backup***

1. Stop the license server.
2. Use the `-restore-database` or `-restore-service-database` argument with the command that runs the license server script to restore trusted storage.

   The *path* value used in this command is the path to the trusted-storage backup file `flexnetls_licenses.mv.db.ts`, located under `tsBackup` in the trusted storage directory. Note that the explicit path values listed in this step use the default value (`${base.dir}`) for the license server policy

`server.trustedStorageDir`, which defines the trusted storage directory. You might need to adjust the path based on this policy definition on your license server. (See Appendix A, Reference: Policy Settings for the License Server.)

If more than one user log-in will be doing a restore, you should set `server.trustedStorageDir` to a path other than `${base.dir}`.

On Windows

- In console mode, enter:

  `flexnetls.bat -restore-database path`

  where *path* is `C:\Users\user_name\flexnetls\producer_name\tsBackup\flexnetls_licenses.mv.db.ts`.

- As a service, enter:

  `flexnetls -restore-service-database path`

  where *path* is `C:\Windows\ServiceProfiles\NetworkService\flexnetls\producer_name\tsBackup\flexnetls_licenses.mv.db.ts`.

On Linux

- In Linux console mode, enter:

  `./flexnetls -restore-database path`

  where *path* is `/var/opt/flexnetls/producer_name/tsBackup/flexnetls_licenses.mv.db.ts`.

- As a service (Systemd or SysV), use the same command as used for console mode. Note that the database can be restored only if a custom installation location is defined.

3. Restart the license server.

Best practice for license server administrators is *never* to delete any files in directory containing the corrupted trusted storage; they should allow the restoration process handle the fix. However, if for some reason administrators need to remove the corrupted data, they should delete only `flexnetls_licenses.mv.db`.

One basic method that license server administrators can use to verify that trusted storage has been restored to its previous uncorrupted state is to run an administrator tool you provide to check the feature counts in trusted storage.

Outgoing HTTPS

The FlexNet Embedded local license server is capable of HTTPS communication with FlexNet Operations—that is, *outgoing* HTTPS.

The initial step in this process is for the producer to determine whether to include a truststore file in the server installation and what configuration is needed on the license server.

For a secure connection, a truststore file containing root and intermediate certificates is required.

- If you are using the Revenera-hosted FlexNet Operations as your back office, the back-office server certificate is signed by a known certificate authority. In this case, little or no configuration is needed on the server. See Default Server Configuration.

- If you are using the "on-premises" version of FlexNet Operations as your back office and you are providing your own truststore file which has been signed by an unknown (private) certificate authority, you need to perform some configuration steps. Some configuration is also required if you changed the password found in the Java `cacerts` file. See Enabling Server for HTTPS When Another Certificate Is Used for instructions.

Default Server Configuration

In most scenarios, HTTPS will work "out of the box". A secure connection to the back office requires access to the appropriate root and intermediate certificates for connection validation. The default behavior is to use the truststore shipped with the Java runtime environment (JRE). This is the "cacerts" file found in the `lib/security` directory of the JRE installation. The file is encrypted, and is installed with a password of "changeit".

It is considered best practice to change the password to guard against malware that is silently adding bogus root certificates. Note that this will make JRE updates harder, as the action will have to be repeated. If you choose to do this, then the password should be supplied in `local-configuration.yaml`. You can do this in an obfuscated form by using the `java -jar flexnetls.jar -password` command. If you change the password, inform the license server administrator of the new password.

When defining the `lfs.url` setting in the `producer-settings.xml` file to identify the back-office URL, ensure that the URL begins with **`https://`**. (See the Reference: Policy Settings for the License Server appendix for more information about the `lfs.url` setting and about generating the `producer-settings.xml` file.)

No additional configuration should be needed for Revenera-hosted FlexNet Operations instances and instances where FlexNet Operations is hosted on AWS.

Important ▪ *Ensure that you are using an up-to-date version of Java 8. Older versions of Java 8 do not contain the requisite AWS root and intermediate certificates. If used, you will encounter HTTPS errors. For more information about Java prerequisites, consult the FlexNet Embedded License Server Release Notes.*

If any SSL-related errors appear in the server log, refer to the following section, Enabling Server for HTTPS When Another Certificate Is Used.

Enabling Server for HTTPS When Another Certificate Is Used

Use the following steps to enable the license server for HTTPS communications when you have the "on premises" version of FlexNet Operations installed that is using a private certificate authority whose root certificates are not found in the `cacerts` file.

- Step 1: Provide the Client Certificate
- Step 2: Provide Access to Client Certificate
- Step 3: Specify the Correct Back-office URL in Policy Settings

Step 1: Provide the Client Certificate

As producer, generate a client certificate file containing the same root certificate (and possibly the certificate chain) that FlexNet Operations used to sign its server certificate. This certificate file is created using certificate-management tools included in your Java installation. For information, see the *FlexNet Operations Installation Guide* (only available for publishers using an on-premises implementation of FlexNet Operations).

Include this client certificate file as a deliverable in the license server installation at the enterprise.

Step 2: Provide Access to Client Certificate

There are different ways to provide access to the client certificate, depending on whether the license administrator is newly installing the local license server or whether they have an existing installation.

- New License Server Installations—updating the `local-configuration.yaml` file
- Existing License Server Installations—creating or updating the “client” configuration file

New License Server Installations

Update the `local-configuration.yaml` file to override the default server configuration. On Linux systemd systems, this file is generated in the `/opt/flexnetls/producer` directory during installation. For SysV installs, the license server software package includes an example `local-configuration.yaml` file.

On Windows, the `local-configuration.yaml` file is located in the same directory as `flexnetls.jar`.

You might need to change the password or the path to the truststore file, or both.

Edit the following settings in `local-configuration.yaml` so that they look similar to this:

```
https-out:
  # Set to true to enable
  enabled: true
  # Path to truststore containing server certificate.
  truststore-path: path-to-your-truststore
  # Truststore password. You can obfuscate this with java -jar flexnetls.jar -password your-
  password-here
  truststore-password: your-password-here
  # Switch off if you're having host validation problems (not recommended)
  host-verify: true
  # Set to true if you're using self-signed certificates (not recommended)
  self-signed: false
```

The password is not required to be stored in plain text; you can obfuscate it first. The following shows an example session to obfuscate the password:

```
$ java -jar flexnetls.jar --password=abracadabra
abracadabra => OBF:1ri71v1r1v2n1ri71shq1ri71shs1ri71v1r1v2n1ri7
```

Be sure to set the `truststore-password` value to the entire obfuscated string, including the `OBF:` prefix.

For detailed instructions about editing the `local-configuration.yaml` file, see Edit “local-configuration.yaml” (Windows) and Edit “local-configuration.yaml” (Linux).

Existing License Server Installations

In older installations of the local license server, a “client” configuration file was used that provides access to the client certificate file you are providing (step 1). This method is no longer recommended; instead, Revenera recommends you update the `local-configuration.yaml` file (see New License Server Installations). If you have an existing configuration file, you can transfer the HTTPS values from the “client” configuration file to `local-configuration.yaml`, as described in Transferring HTTPS Values from “client” Configuration File to YAML File, below.

If you choose to use a “client” configuration file, include it in the license server installation at the enterprise. If the license server administrator creates the file, provide the administrator with instructions on how to create the file, including what path and password to use for the certificate file.

Task ***To create or update the “client” configuration file***

1. The minimum content of this file includes the following:

   ```
   truststore.path=path-to-client-certificate
   truststore.password=password
   ```

   The password is not required to be stored in plain text; it can be obfuscated first. Here is an example session:

   ```
   $ java -jar flexnetls.jar --password=abracadabra
   abracadabra => OBF:1ri71v1r1v2n1ri71shq1ri71shs1ri71v1r1v2n1ri7
   ```

   Set the `truststore-password` value to the entire obfuscated string, including the `OBF:` prefix.

2. The license server administrator at the enterprise uses one of these steps to tell the license server where the “client” configuration file is located:

   - In Windows, the administrator edits the `flexnetls.settings` file to include this option to specify the path to the configuration file:

     ```
     HTTPS_CLIENT_CONFIG=path-to-configuration-file
     ```

     In Linux, the administrator edits the following variable in the `/etc/default/flexnetls-producer_name` file, replacing the comment with the path to the configuration file:

     ```
     HTTPS_CLIENT_FILE=    #empty, can replace by -https-client-configuration path
     ```

     See Manage the License Server Service on Linux in Chapter 3, Installing and Running the License Server, for complete instructions on editing this local settings file.

Transferring HTTPS Values from "client" Configuration File to YAML File

Instead of using the "client" configuration file, you can use the `https-out` setting in `local-configuration.yaml`. The following table shows how to transfer the values from the "client" configuration file to `local-configuration.yaml`. Note that the settings `truststore-path` and `truststore-password` in `local-configuration.yaml` use a hyphen instead of a dot.

"client" Configuration File Content	local-configuration.yaml File Content

`truststore.path=path-to-client-certificate`
`truststore.password=password`
	`https-out:`
	`# Set to true to enable`
	`enabled: true`
	`# Path to truststore containing server certificate.`
	`truststore-path: path-to-your-truststore`
	`# Truststore password. You can obfuscate this with java -jar flexnetls.jar -password your-password-here`
	`truststore-password: your-password-here`
	`# Switch off if you're having host validation problems (not recommended)`
	`host-verify: true`
	`# Set to true if you're using self-signed certificates (not recommended)`
	`self-signed: false`

Step 3: Specify the Correct Back-office URL in Policy Settings

As producer, ensure that the correct URL for your "on premises" version of FlexNet Operations is defined for the `lfs.url` setting in the `producer-settings.xml` file. The URL must begin with `https://`.

See the Reference: Policy Settings for the License Server appendix for more information about the `lfs.url` setting and about generating the `producer-settings.xml` file.

Incoming HTTPS

In most cases, communication between client devices and the license server need not be secure. However, in deployments where security is required, the local license server can be configured to support exchanges with the client devices over a secure HTTPS connection.

Best practice for the enterprise customer is to offload the HTTPS protocol work to a separate device that specializes in this type of processing. However, if this practice is not feasible or desirable, the enterprise's license server administrator can use these steps to configure the server to support incoming HTTPS connections.

- Step 1: Obtain Certificate
- Step 2: Enable Access to "server" Certificate
- Step 3: Define Scope of HTTPS Communications

Step 1: Obtain Certificate

The license server administrator requires a “server” certificate from a certificate authority (CA) such as Verisign. This certificate must be in Java keystore or PKCS#12 format. The latter should have a filename extension of '.pfx' or '.p12'. The certificate must be secured by a password. The certificate provider will have instructions on how to request it in this format (or convert to it.)

Step 2: Enable Access to “server” Certificate

Access to the “server” certificate can be provided in different ways, depending on whether the license server administrator is newly installing the local license server or whether they have an existing installation.

- New Installations—updating the `local-configuration.yaml` file
- Existing Installations—creating or updating the “server” configuration file

New Installations

For new installations of the local license server, the administrator specifies the location of the keystore file holding the “server” certificate in the `local-configuration.yaml` file using the `https-in` parameter. On Linux systemd systems, the `local-configuration.yaml` file is generated in the `/opt/flexnetls/ producer` directory during installation. For SysV installs, the license server software package includes an example `local-configuration.yaml` file.

On Windows, it is located in the same directory as `flexnetls.jar`.

The license server administrator should edit the following settings in `local-configuration.yaml` so that they look similar to this:

```
https-in:
  # Set to true to enable
  enabled: true
  # HTTPS listening port
  port: 1443
  # Path to keystore
  keystore-path: path-to-your-keystore
  # Keystore password. You can obfuscate this with
  java -jar flexnetls.jar -password your-password-here
  keystore-password: your-password-here
```

Note the following:

- The password is not required to be stored in plain text; the license server administrator can obfuscate it first. The following shows an example session to obfuscate the password:

  ```
  $ java -jar flexnetls.jar --password=abracadabra
  abracadabra => OBF:1ri71v1r1v2n1ri71shq1ri71shs1ri71v1r1v2n1ri7
  ```

  The `keystore-password` value must be the entire obfuscated string, including the `OBF:` prefix.

- The standard HTTPS port is 443 (as defined by the HTTPS protocol)

- Best practice on Linux is to use a port number greater than 1024 to avoid issues with possibly having to obtain extra privileges to use a lesser port number. Should port 443 be required at a customer enterprise, you can always use iptables to redirect the port from 443.
- Ensure that a firewall is not blocking the license server.

Existing Installations

In older installations of the local license server, a “server” configuration file was used that provides access to the “server” certificate file that the administrator obtained in step 1. This method is no longer recommended; instead, Revenera recommends that the license server administrator update the `local-configuration.yaml` file (see New Installations). If they have an existing configuration file, they can transfer the HTTPS values from the “client” configuration file to `local-configuration.yaml`, as described in Transferring HTTPS Values from “client” Configuration File to YAML File, below.

Task ***To create or update the “server” configuration file***

1. The minimum content of this file includes the following:

```
keystore.path=path-to-server-certificate
keystore.password=password
https.port=1443
```

   Note the following:

   - The password is not required to be stored in plain text; the license server administrator can obfuscate it first. The following shows an example session to obfuscate the password:

     ```
     $ java -jar flexnetls.jar --password=abracadabra
     abracadabra => OBF:1ri71v1r1v2n1ri71shq1ri71shs1ri71v1r1v2n1ri7
     ```

     The `keystore-password` value must be the entire obfuscated string, including the `OBF:` prefix.

   - The default HTTPS port is 443.
     - Best practice on Linux is to use a port number greater than 1024 to avoid issues with possibly having to obtain extra privileges to use a lesser port number. Should port 443 be required at a customer enterprise, you can always use iptables to redirect the port from 443.
     - On Windows, ensure that a firewall is not blocking the license server.

2. Use the appropriate step to tell the license server where the “server” configuration file is located:

   - In Windows, edit the `flexnetls.settings` file to include this option to set the path to the configuration file:

     ```
     HTTPS_SERVER_CONFIG=path-to-configuration-file
     ```

   - In Linux, edit following variable in the `/etc/default/flexnetls-`*`producer_name`* file, replacing the comment with the path to the configuration file:

     ```
     HTTPS_SERVER_FILE=    #empty, can replace by -https-server-configuration path
     ```

See Editing Local Settings Post-Installation in the Installing and Running the License Server chapter for complete instructions on editing the local settings file.

Transferring HTTPS Values from "server" Configuration File to YAML

Instead of using the "server" configuration file, you can use the `https-in` setting in `local-configuration.yaml`. The following table shows how to transfer the values from the "server" configuration file to `local-configuration.yaml`. Note that the settings `keystore-path` and `keystore-password` in `local-configuration.yaml` use a hyphen instead of a dot.

<table>
<tr><th>"client" Configuration File Content</th><th>local-configuration.yaml File Content</th></tr>
<tr><td><pre>keystore.path=path-to-server-certificate
keystore.password=password
https.port=1443</pre></td><td><pre>https-in:

  # Set to true to enable
  enabled: true

  # HTTPS listening port
  port: 1443

  # Path to keystore
  keystore-path: path-to-your-keystore

  # Keystore password. You can obfuscate this with
  java -jar flexnetls.jar -password your-password-here
  keystore-password: your-password-here</pre></td></tr>
</table>

Step 3: Define Scope of HTTPS Communications

At this point, the license server administrator should have a license server that is operating in both HTTP and HTTPS modes. The next step is to define the scope of HTTPS usage for the license server. For example, perhaps the administrator wants to run the FlexNet Embedded local license server entirely in HTTPS mode. To do so, the administrator must switch off the HTTP listening port by changing the `PORT` value to zero (0) in the `flexnetls.settings` file (in Windows) or the `/etc/default/flexnetls-`*`producer_name`* file (in Linux).

See Editing Local Settings Post-Installation in Chapter 3, Installing and Running the License Server, for complete instructions on editing this local-settings file.

Proxy Support for Communication with the Back Office

For security purposes, some corporate networks require networking proxies as an intermediary between their internal systems and the Internet. To address this requirement, the FlexNet Embedded local license server supports outbound communication with FlexNet Operations through an HTTP proxy. The license server depends on Java Runtime to provide networking services.

Note *▪ The local license server supports both authenticating and non-authenticating HTTP proxies.*

The following sections explain how to set up the license server to use this proxy:

- Configuring the License Server for Proxy Support
- Obfuscating the Proxy Password

Configuring the License Server for Proxy Support

You and the license server administrator for the enterprise customer perform these tasks to configure the license server for HTTP communications with the back office through a networking proxy:

- Producer: Provide Appropriate Information in the Producer Settings File
- License Server Administrator: Configure the License Server

Producer: Provide Appropriate Information in the Producer Settings File

As producer, ensure that the `lfs.url` setting, identifying the back-office URL, is included in the `producer-settings.xml` file delivered with the license server product to the enterprise customer.

See the Reference: Policy Settings for the License Server appendix for more information about the `lfs.url` setting and about generating the `producer-settings.xml` file.

License Server Administrator: Configure the License Server

Use this information to configure the license server for HTTP communications with the back office through a networking proxy:

- Proxy Configuration Basics
- Supported Proxy Parameters
- Parameter Format

Proxy Configuration Basics

The license server administrator for the enterprise customer must define a set of proxy parameters as local settings on the license server (`flexnetls.settings` on Windows or `/etc/default/flexnetls-`*`producer_name`* on Linux). These parameters are passed to the Java Runtime system to identify the HTTP proxy.

The administrator defines these parameters in the `EXTRA_SYSPROPERTIES` setting, which must be "uncommented" if it is currently "commented" in the this file.

For complete instructions on editing the local settings file, refer to the Installing and Starting the License Server and Manage the License Server Service on Linux sections of Chapter 3, Installing and Running the License Server.

Supported Proxy Parameters

The following proxy parameters are supported:

- `http.proxyHost`
- `http.proxyPort`
- `http.proxyUser`
- `http.proxyPassword`

Consider the following:

- If not included, the `http.proxyPort` parameter defaults to `80`.

- The `http.proxyUser` and `http.proxyPassword` parameters are optional. They are required only if the proxy requires authentication.
- Best practice is to use an obfuscated password instead of a plain-text value for `http.proxyPassword`. See Obfuscating the Proxy Password for encryption details.

Parameter Format

Follow these instructions when adding the proxy parameters to the `EXTRA_SYSPROPERTIES` setting in the local settings file:

- Each parameter specified for `EXTRA_SYSPROPERTIES` must use the following Java system syntax:

  `-D`*`property`*`=`*`value`*

- The parameters are separated from each other with a space, and the entire set of parameters are enclosed in double quotations. The following shows an example `EXTRA_SYSPROPERTIES` setting with proxy parameters specified:

  ```
  EXTRA_SYSPROPERTIES="-Dhttp.proxyHost=10.90.3.133 -Dhttp.proxyPort=3128 -Dhttp.proxyUser=user1a
  -Dhttp.proxyPassword=user1apwd35"
  ```

  Even if you specify only a single parameter, it must be enclosed in double quotations:

  ```
  EXTRA_SYSPROPERTIES="-Dhttp.proxyHost=10.90.3.133"
  ```

Obfuscating the Proxy Password

Best practice for the license server administrator is to obfuscate the proxy password and use this encrypted value for the `http.proxyPassword` parameter.

The following is an example session showing both the command used to obfuscate a plain-text password (in this case, `user1apwd35`) and the resulting obfuscated string:

```
$ java -jar flexnetls.jar --password=user1apwd35 user1apwd35 =>
OBF:1ri71v1r1v2n1ri71shq1ri71shs1ri71v1r1v2n1ri7
```

The administrator would then set the `http.proxyPassword` value to the entire obfuscated string, including its `OBF:` prefix, as shown in this example:

```
-Dhttp.proxyPassword=OBF:1ri71v1r1v2n1ri71shq1ri71shs1ri71v1r1v2n1ri7
```

Extended Hostids

The FlexNet Embedded local license server typically resides with an enterprise customer, serving its pool of licenses within that enterprise. When the license server communicates with the back office (FlexNet Operations) to obtain its pool of licenses, the back office uses the server’s hostid to identify that particular server. This kind of standard server hostid is typically the server system’s Ethernet address or can be a VM UUID or the ID for an attached dongle.

However, there are some cases in which the same license server will be conveyed by an operative of the producer to multiple enterprises to distribute features to multiple enterprises. Specifically, one such case is a field engineer running a notebook with the server installed, taking the notebook to multiple sites to distribute licenses to enterprise customers. In such cases, it is undesirable to use the same server hostid at each location, as each enterprise's pool of licenses is expected to be different.

For this sort of case, the license server supports *extended hostids*. An extended hostid is the standard hostid value extended with the hostid type (Ethernet/MAC address, dongle, or VM UUID) and an engineer-defined suffix.

***Note ▪** Refer to the "FlxHostIdType.h" in the FlexNet Embedded client toolkit for a definition of all the possible hostid type enumerations.*

The following sections describe extended hostids:

- Scenario Using Extended Hostids
- Enabling Support for Extended Hostids
- Setting and Using Extended Hostids

Scenario Using Extended Hostids

Using an extended hostid, field engineers can each assign a unique suffix for each enterprise customer visited, which keeps the license server instances distinct as far as the back office is concerned. For example, Field Engineer 1 might have a notebook with Ethernet address 11111111. While at Customer A, Engineer 1 might use suffix "E1CA", resulting in extended hostid of "11111111-3-E1CA". (The "3" is FlexNet Embedded's predefined identifier for the Ethernet address hostid type.) When the license server contacts the back office, this extended hostid is registered and associated with the pool of licenses for Customer A. The engineer then distributes licenses to Customer A's client systems, and the server's synchronization process will report this distribution information to the back office.

After finishing synchronization of Customer A's data and before continuing to the next customer, the field engineer must reset trusted storage using the license server's `reset-database` option (or a similar custom option). (The `reset-database` option works only when all transaction records in trusted storage have been synchronized to the back office.)

At Customer B, Engineer 1 might then use suffix "E1CB", resulting in extended hostid of "11111111-3-E1CB". This second extended hostid is separately registered in the back office and associated with the new customer's pool of licenses. Once again, synchronization to the back office communicates this customer's license distribution to the producer.

Using this technique, the same server host can appear as distinct license servers in the back office, enabling each enterprise customer's entitlements and license checkouts to remain separate.

Enabling Support for Extended Hostids

To enable support for extended hostids on the license server, ensure that the `server.extendedHostId.enabled` setting is set to `true` (the default value) in the policy settings file (`producer-settings.xml`) file:

```
server.extendedHostId.enabled=true
```

Also, the `reset-database` option used to reset trusted storage between customers works only when all transaction records from the previous customer have been synchronized to the back office. However, if you want your field engineer or operative to be able to force a reset of the trusted storage when unsynchronized records still exist in trusted storage, set the `server.forceTSResetAllowed` setting to `true` (`false` is the default). See Enabling a Forced Trusted-Storage Reset in the Trusted-Storage Reset section for complete instructions.

The license server configuration utility is used to edit these policy settings. For more information, see License Server Configuration Utility in Chapter 7, Producer Tools.

Setting and Using Extended Hostids

On the host license server, use the following procedure to set a specific extended hostid (for a specific customer), obtain the licenses entitled to that hostid, and distribute them to the hostid's clients. Once license distribution for the extended hostid is synchronized to the back office, you must reset trusted storage on the host license server in preparation to use a different extended hostid (for another customer).

Extended hostid suffixes should contain only the following characters: a-z, A-Z, and 0-9. Use of other characters can result in errors from third-party applications.

Task ***To set and use an extended hostid on the host license server***

3. On the host license server, specify the extended hostid by providing its suffix:

    - If the license server is running on Windows, open the `flexnetls.settings` file in a text editor, and enter a value for the `EXTENDED_SUFFIX` parameter, as shown in this example:

    ```
    EXTENDED_SUFFIX=HOSTID1
    ```

    - If the license server is running on Linux, configure the `/etc/default/flexnetls-`*`producer_name`* file to include this setting:

    ```
    EXTENDED_SUFFIX=HOSTID1
    ```

    See Manage the License Server Service on Linux in Chapter 3, Installing and Running the License Server, for complete instructions on editing and reapplying the server's local-settings file in a Windows or Linux environment.

4. To verify that the extended hostid is currently in effect, access the `/hostids` REST API (see Viewing Instance hostids in Chapter 4, License Server REST APIs).
5. Send a capability request to the back office to obtain the licenses entitled to the extended hostid, and distribute these licenses to the hostid's clients.
6. To ensure that the current license distribution for the extended hostid is available in the back office, wait for the license server to synchronize (or force a synchronization using the `/sync` REST API).
7. Use the procedures in Determining Whether Records Pending Synchronization Exist in the *Trusted-Storage Reset* section to ensure that synchronization is complete. (Trusted-storage reset, described in the next step, fails if any unsynchronized transaction records remain in trusted storage.)
8. Reset trusted storage using the procedures described in Resetting Trusted Storage in the *Trusted-Storage Reset* section.
9. Repeat the previous steps to set and use another extended hostid.

Trusted-Storage Reset

The FlexNet Embedded local license server provides a `reset-database` option to reset trusted storage on the license server. Because this option removes all licenses and stored information from trusted storage, you should use this option cautiously, such as for testing purposes, deployment of extended hostids (see Extended Hostids), or repairing a corrupted trusted storage.

Caution ▪ *In general, the enterprise license server administrator should not have direct access to the "reset-database" option. You can provide this access as necessary, such as to repair trusted storage, and under your direction.*

This section covers the following procedures for using the `reset-database` option:

- Determining Whether Records Pending Synchronization Exist
- Resetting Trusted Storage
- Enabling a Forced Trusted-Storage Reset

Determining Whether Records Pending Synchronization Exist

By default, the `reset-database` option to reset trusted storage will not work if any transactions records are pending synchronization. Before running the reset, use the following procedure to determine whether trusted storage contains unsynchronized records. If such records do exist, you need ensure they are synchronized before resetting trusted storage.

As an alternative, you can set a license server policy that allows you to force a trusted-storage reset when transaction records are pending synchronization. See Enabling a Forced Trusted-Storage Reset for details.

Task ***To determine whether trusted storage contains transaction records pending synchronization***

1. Perform either action to determine whether unsynchronized records exist and to obtain the time of the last synchronization:

    - Using the FlexNet License Server Administrator command-line tool (included in the `enterprise` directory in your license-server software package), run the `-status` command shown in this example:

      ```
      flexnetlsadmin.bat -server http://localhost:7070 -status
      ```

      The output includes the synchronization status, as highlighted in this sample excerpt. (In this case, 42 records are pending synchronization, and the last synchronization occurred 7 days, 4 hours, 38 minutes and 28 seconds ago.)

      ```
      Version              : 2020.01.0
      Build Version        : 262400
      Server               : https://localhost:7070/api/1.0/instances/~
      State                : Ready
      Backup Server        : Not configured
      BackOffice Server    : http://localhost:8080/request
      ```

```
Records Pending Sync : 42
Last Sync            : 7d 4h38m28s
```

Note ▪ *The time format for the "Last Sync" value is* `nYnMnD nHnMnS`*, where Y=years, M=months, D=days, H=hours, M=minutes, and S=seconds. If any leading elements are missing, they are assumed to have a value of zero. If a sync has never been performed, "sync to back office pending" is displayed.*

See the "Using the FlexNet License Server Administrator Command-line Tool" chapter in the *FlexNet Embedded License Server Administration Guide* for details about the `-status` command.

- Use the `/instances/instanceID?syncinfo` REST API. The response body includes `lastSyncTime` (listed in Greenwich Mean Time) and `recordsPendingSync` to show the synchronization status, as depicted in this sample output excerpt:

```
{
  "id" : 1,
  "pendingDeletion" : false,
 ...
  "lastUpdateTime" : "2019-08-24T16:44:36.000Z",
  "failOverRole" : "MAIN",
  "lastSyncTime" : "2020-02-12T20:24:16.093Z",
  "lastUpdateHostid" : {
    "hostidType" : "ETHERNET",
    "hostidValue" : "7C7A9999999B"
  },
  "identityName" : "demo-rsa",
  "recordsPendingSync" : 2,
}
```

For more information about this REST API, see Chapter 4, License Server REST APIs.

2. If transaction records are pending synchronization, either wait for the next scheduled synchronization (which you can determine from the `lastSyncTime`), or use the `/sync` REST API to force a synchronization. For information about synchronization, see Synchronization Operations in Chapter 5, More About Basic License Server Functionality.

3. Once synchronization occurs, check for the following message in the log file:

```
Sync completed for instance instanceID.
```

Also, you might want to redo Step 1 to confirm that *no* transaction records are pending synchronization.

Resetting Trusted Storage

Use the appropriate procedure to reset trusted storage on the license server.

On Windows

The following procedure resets trusted storage on a license server installed as a service in a Windows environment.

Task ***To reset trusted storage for a license server running as a service on Windows***

As an administrator, open a command window, and execute the following:

```
flexnetls.bat -reset-database
```

On Linux

The following procedure resets trusted storage on a license server installed as a service in a Linux environment.

Task ***To reset trusted storage when running the license server as a service on Linux***

Stop the service, run the following command, and then start the service:

```
./flexnetls -reset-database
```

See Chapter 3, Installing and Running the License Server, for details about stopping and starting the license server service on Linux.

Enabling a Forced Trusted-Storage Reset

By default, if unsynchronized transaction records exist in the license server's trusted storage, any attempt to reset trusted storage using the `reset-database` option will fail with an error message. However, you can configure the license server to permit a forced trusted-storage reset in emergency cases when a reset is needed but records have not been synchronized.

Task ***To enable a forced trusted-storage reset***

1. Make a copy of your current policy settings file, `producer-settings.xml`.
2. Generate a new policy settings file that sets `server.forceTSResetAllowed` to **`true`**. Use the license server configuration utility to generate the new file. For more information about this process, see License Server Configuration Utility in Chapter 7, Producer Tools.
3. In the `server` directory of the license server, replace the current `producer-settings.xml` file with the new one.

With `server.forceTSResetAllowed` set to **`true`**, you or your operative (or, in special cases, the enterprise license server administrator) can force a trusted-storage reset even though transaction records are pending synchronization.

For security purposes, once trusted storage has been reset, best practice is to restore the previous version of the policy settings file (in which `server.forceTSResetAllowed` is set to **`false`**) on the license server.

License Checkout: Support for Special Capability-Request Options

The FlexNet Embedded local license server and the CLS instance supports the following special capability-request options:

- Incremental Capability-Request Processing
- Granting All Available Quantity for a Feature
- Feature-Selector Filtering
- Vendor Dictionary Data
- Capability Preview

Incremental Capability-Request Processing

When the license server processes a regular capability request from the FlexNet Embedded client, requesting desired features, the server considers the licenses currently served to the client as "returned" and sends a capability response with the desired features, if they are available. When the FlexNet Embedded client processes the capability response, the client's existing licenses are removed and the new licenses added.

If the client wants to maintain its existing licenses when it requests new features, it must either explicitly include the existing features as desired features in the capability request or mark the request as "incremental" (without specifying existing features). The following describes how the license server processes an incremental capability request:

- Processing Overview
- Incremental Request Examples
- Considerations and Limitations

For more information about setting up a capability request marked as "incremental", refer to either of the following:

- *Performing a JSON Capability Exchange* chapter in the *Cloud Monetization API User Guide* (when using the Cloud Monetization API, a form of REST-driven licensing)
- "Licenses Obtained from a License Server" section in the *Using the FlexNet Embedded APIs* chapter in your FlexNet Embedded client SDK user guide (when using the language-based APIs delivered in a FlexNet Embedded client SDK).

Processing Overview

When the license server processes a capability request marked as "incremental", it attempts to renew all licenses currently served to the client and include them in the capability response along with the desired features that are available. (When processing the request, the license server attempts to renew existing features first, then processes the desired features.) If the license server determines that a certain existing feature cannot be renewed (that is, it has expired or is no longer available), it is not included in the response. Ultimately then, the

license server's response to an incremental request 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 following sections provide examples of how the incremental capability request is processed.

Example 1: Renew Existing Features and Add New Features

For example, if the license server receives a capability request for 1 desired count of the `highres` feature from a client that has previously been served 1 count of the `survey` license, the following happens:

- If the capability request is a regular request (that is, not marked as "incremental"), the license server generates a capability response with 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 license server generates a response with 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

An incremental capability request allows the client to increment the count of a renewed feature. For example, if the license server receives a capability request for 2 desired counts of `survey` from a client that has been previously served 1 count of `survey`, the following happens (assuming that the existing feature is renewable and the requested new counts are available):

- If the capability request is not marked as "incremental", the capability response from the license server includes simply the requested 2 desired counts of `survey`.
- If the request is marked as "incremental", the license server generates a response with 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

An incremental capability request allows the client to renew all existing features but reduce the count of (or not renew at all) selected existing features. To set up such a request, you must explicitly include the existing feature whose count you want to reduce (or that you do not want to renew) as a desired feature with a negative count in the request.

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 license server receives 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 license server generates a response with 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 request processing:

- An incremental capability request is compatible with only the “request” operation type and concurrent features.
- Incremental capability requests are compatible with the use of reservations on the license server. For more information, refer to License Reservations in Chapter 5, More About Basic License Server Functionality, and to the appendix Effects of Special Request Options on Use of Reservations.
- Incremental capability requests are compatible with the use of named license pools on the license server. For more information, refer to Allocating Licenses Using Named License Pools in Chapter 5, More About Basic License Server Functionality.
- 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.)

Granting All Available Quantity for a Feature

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:

- How the Request is Processed
- Considerations and Limitations

For more information about setting up a capability request that marks features as "partial", refer to either of the following:

- *Performing a JSON Capability Exchange* chapter in the *Cloud Monetization API User Guide* (when using the Cloud Monetization API, a form of REST-driven licensing)
- "Licenses Obtained from a License Server" section in the *Using the FlexNet Embedded APIs* chapter in your FlexNet Embedded client SDK user guide (when using the language-based APIs delivered in a FlexNet Embedded client SDK).

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. For more information about reservations, refer to License Reservations in Chapter 5, More About Basic License Server Functionality, and to the appendix Effects of Special Request Options on Use of Reservations.
- Partial-checkout features are compatible with the use of named license pools on the license server. For more information, refer to Allocating Licenses Using Named License Pools in Chapter 5, More About Basic License Server Functionality.
- These features are compatible with incremental capability requests (see Incremental Capability-Request Processing).
- They are compatible with the capability requests defined with the *request* operation type only. They are not compatible with the *report* or *undo* operation type.

Feature-Selector Filtering

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. For example, as producer, you might also want to filter a given feature by the client's region and department because you vary the cost and availability of this feature by these attributes.

To filter a feature on one or more attributes in addition to the basic criteria, you 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. Once the license server obtains this feature, the client code can include the feature selectors, along with feature's basic criteria, 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.

The following sections describe more about feature-selector filtering:

- Set Up the Selectors in FlexNet Operations
- Provision the Feature on the License Server
- Include Feature Selectors in the Capability Request
- Considerations and Limitations

Set Up the Selectors in FlexNet Operations

You define the feature selectors in FlexNet Operations, using the `Vendor String` property for the license model that will be associated with the feature. The following description highlights important aspects about this feature-selector setup process. For complete instructions on setting up a license model, associating it with the product (to which the feature belongs), and entitling the product to an enterprise customer, see the *FlexNet Operations User Guide*, in particular the section "*FlexNet Operations Getting Started Guide for Usage Management*".

Each feature selector entered for the `Vendor String` property must be in *key*:*value* format, where *value* must be a string, not an integer. Additionally, each *key*:*value* string must be enclosed with "%%" to set it up as a token. You can tokenize individual *key*:*value* pairs (for instance, `%%key:value%%`), especially when these pairs are interspersed among other vendor strings defined for the property; or you tokenize the entire set of the *key*:*value* pairs as shown:

```
%%key1:value1,key2:value2,key3:value3%%
```

As an example, you might want to define two feature selectors, one each for the customer's region and department. For the `Vendor String` value, you can enter explicit feature selectors:

```
%%REGION:Global,DEPARTMENT:Acct%%
```

Alternatively, you can set up user-defined functions or use "customized attribute" substitution in FlexNet Operations to populate the selector values. For example, you might define `REGION` as a customized license attribute and `DEPARTMENT` as a customized entitlement-line attribute. In this case, the values for both attributes are specified when a feature (as part of a product) is entitled. To set up the `Vendor String` value to capture these values, provide the proper substitution functionality, similar to this:

```
%%REGION:{REGION},DEPARTMENT:{EntitlementLineItem.DEPARTMENT}%%
```

When the `REGION` and `DEPARTMENT` values are selected during the entitlement process for the enterprise customer, the selector values are also populated in the `Vendor String` property.

Whether feature selectors are defined explicitly or populated through functions for `Vendor String`, this information for the property is used to generate a "features selector" dictionary when the feature is provisioned on the license server.

You can implement more complex feature-selector filtering using an MVEL script.

Provision the Feature on the License Server

Once the entitled feature is provisioned on the license server, you can invoke the license server's `/features` REST API to view the "feature selectors" dictionary for the feature, as well as the vendor-string value used as a basis for the dictionary. For example, if `Vendor String` was populated with **EMEA** for `REGION` and **Acct** for `DEPARTMENT` for the license model in the back office for feature `f2`, the information for the feature on the license server might look like this:

```
{
  "id" : 1,
  "selectorsDictionary" : {
    "REGION" : "EMEA",
    "DEPARTMENT" : "Acct",
  },
  "used" : 0,
  "featureName" : "f2",
```

```
  "expiry" : "2020-12-31",
  "vendorString" : "%%REGION:EMEA, DEPARTMENT:Acct%%",
  "issued" : "2018-08-31",
  "type" : "CONCURRENT",
...
}
```

Include Feature Selectors in the Capability Request

The client application code generates a capability request that includes the feature selectors. The license server serves the requested feature (or tracks its metered usage) only when the selectors in the request match *all* those in the “selectors dictionary” for the feature in the license server’s trusted storage. (Based on the example, the capability request would need to specify both `EMEA` for `REGION` and `Acct` for `DEPARTMENT` as the feature selectors in order to obtain feature `f2`.)

For more information about including feature selectors in the capability request, refer to the appropriate guide:

- *Performing a JSON Capability Exchange* chapter in the *Cloud Monetization API User Guide* (when using the Cloud Monetization API, a form of REST-driven licensing)
- “Licenses Obtained from a License Server” section in the *Using the FlexNet Embedded APIs* chapter in your FlexNet Embedded client SDK user guide (when using the language-based APIs delivered in a FlexNet Embedded client SDK).

Considerations and Limitations

Note the following about filtering with feature selectors:

- If feature selectors are included in the capability request, they must match all selectors in the “selectors dictionary” for a given 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 comma is not supported within the key or value.
- The *key* element in the feature selector is case-sensitive and supports UTF-8 characters.
- The *value* element in the 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.
- Using feature selectors in combination with named license pools may produce unexpected results.

- Feature selectors are compatible with the capability-request operation types “request” and “preview”, but are *not* compatible with “report” and “undo”.

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 license server and vice-versa. Basically, the vendor dictionary provides a means to send information back and forth between the client and server for any producer-defined purposes, as needed; FlexNet Embedded does not interpret this data.

Vendor dictionary data is stored as key–value pairs. The key name is always a string, while a value can be a string or a 32-bit integer value, or an array of those types (arrays are only supported when using the Cloud Monetization REST API). Keys are unique in a dictionary and hence allow direct access to the value associated with them. The maximum size for a vendor dictionary is 7168 bytes after the dictionary’s conversion to base64.

For more information about including a vendor dictionary in capability requests, refer to either of the following:

- *Performing a JSON Capability Exchange* chapter in the *Cloud Monetization API User Guide* (when using the Cloud Monetization API, a form of REST-driven licensing)
- “Licenses Obtained from a License Server” section in the *Using the FlexNet Embedded APIs* chapter in your FlexNet Embedded client SDK user guide (when using the language-based APIs delivered in a FlexNet Embedded client SDK).

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.

The client application can request the availability of specific desired features or can request a preview of all available features (and their versions). Additionally, feature selectors can be included in the preview request to enable the license server to filter the available features (see Feature-Selector Filtering).

The following sections describe what the license server returns in the preview capability response:

- Feature Counts Returned in the Preview
- Calculation of Available Counts for Preview

For a list of capability request attributes currently not supported in a preview capability request, see Considerations and Limitations.

For more information about setting up a preview capability request”, refer to either of the following:

- *Performing a JSON Capability Exchange* chapter in the *Cloud Monetization API User Guide* (when using the Cloud Monetization API, a form of REST-driven licensing)
- “Licenses Obtained from a License Server” section in the *Using the FlexNet Embedded APIs* chapter in your FlexNet Embedded client SDK user guide (when using the language-based APIs delivered in a FlexNet Embedded client SDK).

Feature Counts Returned in the Preview

The license server returns a preview response containing 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 to which the client is entitled to according to the model definition, or, if reservations are used, 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 pre-allocated according to the model definition or reserved for 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 Considerations and Limitations for factors that can affect the calculation of these two counts.

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

Calculation of Available Counts for Preview

In general, the license server selects those features to include in the preview capability response that meet the “desired feature” or “all” specification and that match the feature-selector criteria specified in the capability request. However, the server does not include the following in the response:

- Expired features
- Features not yet started
- Features with no counts available for the requesting client

The following scenarios illustrate how the license server computes the preview counts for features:

- Basic Preview Example
- Preview Example for Processing Metered Features
- Preview Example for Processing Overdraft Counts

In the following scenarios, “reserved” refers to pre-allocated feature counts, but no distinction is made whether features are reserved using reservations or using the named license pools functionality.

Basic Preview Example

Suppose the license server has the following concurrent features:

Table 6-3 ▪ Basic Preview Example: Current Feature State on License Server

Feature on License Server	Activated Count	Current State
f1	10	Feature attributes do not match client1 feature selectors
f2	10	Expired
f3	10	Not yet started
f4	10	All counts reserved for client2
f5	10	5 counts served to client3
f6	10	• 1 count reserved for client1 • 1 count reserved for client2 • 1 count served to client3

The following sections describe how the license server handles a capability preview:

- For All Available Features
- For a Specific Desired Feature

For All Available Features

If client1 sends a preview capability request for *all* features and counts, the license server computes the following counts for client1 (assuming that feature `f2` through `f6` have met the feature-selector criteria in the request):

Table 6-4 ▪ Basic Preview Example: Initial “All Features” Counts Computed for Client

Feature on License Server	Computed Count	Computed Maximum Count	Current State
f1	-	-	Rejected—feature does not meet client1 feature-selectors criteria
f2	-	-	Rejected—feature expired
f3	-	-	Rejected—feature not yet started
f4	0	0	All counts reserved for client2

Table 6-4 ▪ Basic Preview Example: Initial "All Features" Counts Computed for Client (cont.)

Feature on License Server	Computed Count	Computed Maximum Count	Current State
f5	5	10	5 counts currently served to client3
f6	8	9	• 1 count reserved for client2 • 1 count currently served to client3

However, the license server sends only the following features in the preview capability response for client1. The f4 feature is not included because the count has a zero (0) value.

Table 6-5 ▪ Basic Preview Example: Actual "All Features" Counts Returned in Preview Response

Feature in Preview Response	Count	Maximum Count
f5	5	10
f6	8	9

For a Specific Desired Feature

If client2 sends a preview capability request for **3** counts of f5, the license server determines that the 5 counts currently available for f5 are enough to satisfy client2's request:

Table 6-6 ▪ Basic Preview Example: Initial Desired Feature Counts Computed for Client

Feature on License Server	Computed Count	Maximum Computed Count	Current State
f5	5	10	5 counts currently served to client3

Therefore, the license server includes the 3 counts of f5 in the preview capability response for client2:

Table 6-7 ▪ Basic Preview Example: Actual Desired Feature Counts Returned in Preview Response

Feature in Preview Response	Count	Maximum Count
f5	3	10

However, if client2 had requested **6** counts of f5, the license server would include *no* features in the preview capability response since the current count of 5 for f5 is not enough to cover the requested preview count of 6 for client2.

Preview Example for Processing Metered Features

When processing a preview capability request, the license server treats any currently served metered feature as permanently consumed, a factor that affects the maximum count for the metered feature.

For example, suppose the license server has 10 counts of the concurrent feature `f1` and 10 counts of the metered feature `m1`. Currently 2 counts of each feature is served to client1.

Table 6-8 ▪ Metered Feature Preview Example: Current Feature State on License Server

Feature on License Server	Activated Count	Current State
f1	10	2 counts served to client1
m1 (metered)	10	2 counts served to client1

The following sections describe how the license server handles a capability preview:

- For All Available Features (When Metered Features Exist)
- For Desired Metered Features

For All Available Features (When Metered Features Exist)

When the client requests a preview of all features, the license server always computes the count and maximum count as identical for a metered feature (even if the feature is defined as “undoable” or “reusable”).

For example, if client2 sends a preview capability request asking for *all* features (and specifying no features selectors), the license server sends the following features in the preview capability response to client2:

Table 6-9 ▪ Metered Feature Preview Example: Actual “All Features” Counts Returned in Preview Response

Feature in Preview Response	Count	Maximum Count
f1	8	10
m1 (metered)	8	8

While the maximum count for the concurrent feature `f1` is based on the eventual return of the served 2 counts, the maximum count for the metered feature `m1` assumes that 2 counts currently served to client1 have been permanently consumed and therefore remains 8.

For Desired Metered Features

If client2 sends a preview request for 2 counts of concurrent feature `f1` and 2 counts of metered feature `m1`, the license server sends the following features in the preview capability response to client2:

Table 6-10 ▪ Metered Feature Preview Example: Actual Desired Feature Counts Returned in Preview Response

Feature in Preview Response	Count	Maximum Count
f1	2	10
m1 (metered)	2	8

The 8 counts currently available for each `f1` and `m1` are enough to satisfy client2's request, so the license server sends the desired 2 counts for each feature in the preview capability response. However, because the 2 counts of metered featured `m1` currently served to client1 are considered permanently consumed, the maximum count for `m1` remains 8.

Preview Example for Processing Overdraft Counts

The count and maximum count for a feature includes overdraft counts.

For example, suppose a license server has 10 regular counts and 10 overdraft counts for concurrent feature `f1`. Currently, 15 counts of `f1` are served to client1:

Table 6-11 ▪ Preview Example for Overdraft Counts: Current Feature State on License Server

Feature on License Server	Regular Count	Overdraft Count	Current State
f1	10	10	15 counts server to client1 (10 regular, 5 overdraft)

If client2 sends a preview capability request for all features (and specifying no feature selectors), the license server sends the following in the preview capability response for client2:

Table 6-12 ▪ Preview Example for Overdraft Counts: Actual Counts Returned in Preview Response

Feature in Preview Response	Count	Maximum Count
f1	5	20

The 5 counts are from the remaining overdraft count; the maximum count includes the 10 regular counts and the 10 overdraft counts.

Considerations and Limitations

Note the following about the capability preview feature:

- As described previously, only features with non-zero counts are included in the preview 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.
- The capability preview functionality currently does not support the following attributes in a preview capability request and generates an error during request generation should either attribute exist:
    - The “partial” attribute for features (described in Granting All Available Quantity for a Feature)
    - The “incremental” attribute for the capability request (described in Incremental Capability-Request Processing)

  The client does not allow a preview capability request to be generated when either of these attributes are enabled in the request.

Usage Capture and Management

The FlexNet Usage Capture and Management feature uses metering licensing to support various pay-for-use and pay-for-overage license models. This feature requires FlexNet Usage Management—a separately purchased component of FlexNet Operations—for metered-licensing setup and usage management and FlexNet Embedded for usage capture. Additionally, a license server—either the FlexNet Embedded local license server or a CLS license server—is required to channel the captured usage data from the FlexNet Embedded client device to FlexNet Operations.

Synchronization Required

The license server uses a synchronization process to send the captured usage data to FlexNet Operations.

The CLS license server performs an automatic (and almost immediate) synchronization once it receives the usage data from the client.

The FlexNet Embedded local license server, however, must be configured to perform synchronization (either online or offline) to send usage data to FlexNet Operations. To support online synchronization, you must ensure that, in the `producer-settings.xml` file, the online synchronization feature is enabled and that the `lfs.url` setting is included (and defined with the URL for FlexNet Operations). To support offline synchronization, you must provide the appropriate synchronization tools (such as `serverofflinesynctool` and `backofficeofflinesynctool`) to the enterprise customer. You or the enterprise can determine the synchronization configuration, such as the synchronization interval (for online synchronization), page size, and other parameters that meet your usage-capture and usage-management requirements. For more information about the license server’s synchronization to FlexNet Operations, see Online Synchronization to the Back Office in Chapter 5, More About Basic License Server Functionality.

For More Information

Additional information about usage-capture and management is found in the following resources:

- For instructions about setting up the metered-license model, metered features, and other associated components used to create the metered-license entitlement that enterprise customer activates on their FlexNet Embedded license server, see the section "*FlexNet Operations Getting Started Guide for Usage Management*" in the *FlexNet Operations User Guide*.
- For information about viewing usage reports and exporting usage data to other entities, such as to your billing system, see the *FlexNet Operations User Guide*, in particular the section "*FlexNet Operations Getting Started Guide for Usage Management*".
- For examples of usage-capture implementations in the FlexNet Embedded client code, refer to the appropriate FlexNet Embedded SDK user guide specific to your programming language.
- For information about the CLS license server, see the *FlexNet Operations User Guide*, in particular the section "*FlexNet Operations Getting Started Guide for Cloud Licensing Service*".

Administrative Security

The FlexNet Embedded license server uses a set of REST APIs to manage the license server, as described in Chapter 4, License Server REST APIs. The license server offers a facility that secures access to these REST APIs to prevent unauthorized queries or access to administrative operations on the local license server or a CLS instance. When this administrative security is enabled on the license server, authorization credentials are needed to access REST API endpoints, thus protecting access to data and administrative operations on the license server.

For either a local license server or CLS instance, administrative security enables the license server administrator to restrict access to certain license server operations, as required by enterprise policies. For a CLS instance, this security is especially critical if you want to provide the enterprise with a REST API client to administer the license server. Because REST APIs are accessed through the Internet, the CLS instance's REST endpoints are insecurely exposed to the public when administrative security is not enabled. With security enabled, however, a CLS instance can be safely managed using the Revenera administration tools or your custom REST API client to manage their license servers.

Additionally, you can provide a single REST API administration client that works with both a secured CLS instance and a local license server.

Overview: Deploying a License Server with Security Enabled

The following provides an overview of the process used to deploy a license server with administrative security enabled.

Table 6-13 • Process for Deploying a License Server with Administrative Security Enabled

Deployment Task	CLS Instance	Local License Server
Enable and configure administrative security	Performed by the Revenera Cloud operator using REST APIs.	Accomplished when you generate the `producer-settings.xml` file with security policies defined. See Enabling Administrative Security on a Local License Server.

Table 6-13 ▪ Process for Deploying a License Server with Administrative Security Enabled (cont.)

Deployment Task	CLS Instance	Local License Server
Create the default administrator account	Automatically created when the CLS instance is created (manually or through auto-provisioning).	Created when you generate the `producer-settings.xml` file with security policies defined. See Enabling Administrative Security on a Local License Server.
Set up the default administrator credentials	Automatically generated when CLS instance is created: ● **User name:** `admin` ● **Password:** Randomly generated value The password must be reset before accessing the instance. See the “Reset default password” entry in this table.	Provided by you: ● **User name:** `admin` ● **Password:** Value you specify when using `flexnetlsconfig` to generate the `producer-settings.xml` file. See Enabling Administrative Security on a Local License Server.
Provide administration tool	Provided by you when you deploy the license server. The tool can be any of the following: ● FlexNet License Server Administrator command-line tool ● Your custom REST API administration client See About License Server Administrator Tools.	
Reset default administrator password	Performed by the enterprise license server administrator using the FlexNet Operations End-User Portal. (For more information, see the portal’s help system.)	Performed by the enterprise license server administrator using one of the tools listed in the previous table entry.
Create enterprise user accounts (optional)	Performed by the enterprise license server administrator using the FlexNet License Server Administrator command-line tool or a REST API client you provide. See About License Server Administrator Tools.	

About Administrative Security

The following sections provide more information about administrative security:

- Secured Functionality on the License Server
- User Management Functionality
- User Roles Defining Administrative Privileges
- Policies to Configure Administrative Security
- Credential Requirements

Secured Functionality on the License Server

When security is enabled on the license server, authorization credentials are required to access the REST APIs that perform the following administrative operations:

- Resetting the administrator password
- Creating and managing other enterprise user accounts
- Adding and deleting a model definition
- Adding and deleting license reservations
- Setting or resetting license server policies, including security policies
- Suspending and resuming the license server
- Manual initiating a binary exchange between the license server and the back office, such an online or offline activation or synchronization event

Additionally, “read” operations might also require credentials, depending on how you configure security on the license server. Refer to the next section Policy Affecting “read” Security for more information.

For more information about the `/authorization` API used to authenticate credentials needed to access to secured REST APIs, see Authorizing Access to Secured REST APIs in Chapter 4, License Server REST APIs.

Operations Exempt from Security

The following operations are always exempt from license-server security measures (that is, no credentials are ever needed to perform these operations):

- Binary exchanges between the license server and FlexNet Operations that are initiated by the license server, such capability-polling exchanges and synchronization at set intervals
- Binary exchanges between the license server and license-enabled clients

Policy Affecting “read” Security

When administrative security is enabled on the license server, it uses the `security.anonymous` license-server policy to determine whether users need to provide credentials simply to “read” information—such as current reservations, features, licenses, or status—on the license server. See Policies to Configure Administrative Security for more information.

User Management Functionality

When you deploy a license server with administrative security enabled, a default license-server administrator account is automatically created when the license server starts up. This account is assigned ROLE_ADMIN, enabling it to create, edit, and delete other user accounts as needed, using the `/users` REST API. For more information about the `/users` API, see APIs to Manage User Accounts in Chapter 4, License Server REST APIs.

For more information about the roles assigned to the default administrator account and to other user accounts that the administrator can create, see the next section, User Roles Defining Administrative Privileges.

User Roles Defining Administrative Privileges

The default license-server administrator account—created when the license server starts up—is assigned ROLE_ADMIN, ROLE_RESERVATIONS, ROLE_DROPCLIENT, and ROLE_READ, giving the administrator full rights to administer the license server. Any user account that the administrator creates thereafter can be assigned one or more roles of these roles, including ROLE_ADMIN to create another administrator. The following table describes these roles.

Keep in mind that any of these roles can perform capability exchanges and synchronization operations, as this functionality is exempt from security measures.

Table 6-14 ▪ User Roles Used to Define Administrative Privileges on the License Server

Role	Privileges
ROLE_READ	Privileges to perform “read” operations (for example, to query features, licenses, named license pools, reservations, the model definition, or server status). When no other role is assigned to a user account, ROLE_READ is assigned by default as the only role. Additionally, depending on the security configuration, either every account is automatically given “read” rights, or you are required to assign ROLE_READ explicitly to each account to give it “read” rights. See Policy Affecting “read” Security.
ROLE_RESERVATIONS	Privileges to add and delete reservations.
ROLE_DROPCLIENT	Privileges to delete client records on the license server.
ROLE_ADMIN	Administrator privileges to update license server policies (local license server only), create and manage other enterprise user accounts, upload and delete a model definition, and perform other administrative tasks, such as suspend or resume the license server.
ROLE_PRODUCER	Privileges given to a producer account (by convention, the account named "producer"). Required to supply or delete checkout filters, and change configuration values not editable by the administrator account (for example, `lfs.syncTo.enabled`).

Policies to Configure Administrative Security

The following describes the license server policies that enable and configure administrative security on the license server you are about to deploy. These policies are viewed and edited through the `/configuration` REST API.

- `security.enabled` to enable or disable security on the license server. (This policy is editable on a local license server only.) See Enabling Administrative Security on a Local License Server.

The following policies are in effect only when security is enabled:

- `security.anonymous` to indicate whether “read” operations require credentials to determine whether users need to provide credentials simply to “read” information—such as current reservations, the model definition, named license pools, features, licenses, or status—on the license server:

    - When `security.anonymous` is set to **true**, user accounts, including administrator accounts, are automatically granted "read" rights (ROLE_READ); no credentials are needed to perform "read" operations. This policy lessens the administrative burden to manage user accounts.
    - When this policy is set to **false**, the license server administrator must explicitly assign ROLE_READ to a user account, including an administrator account, to allow that account to perform "read" operations. (The exception occurs when no role is assigned to an account, in which case ROLE_READ is automatically assigned as the only role.) Users are then required to provide credentials to perform any "read" operation. If an account is not authorized for ROLE_READ, no "read" access is given.
- `security.token.duration` to limit token validity. This policy sets a time limit on how long an authorization "token" is in effect before it expires, requiring a user to re-enter credentials. Note the following:
    - This policy is relevant to those custom administrator tools where credentials are entered once and then automatically applied to each subsequent operation requiring authorization. Once the token expires, the user must re-enter credentials.
    - The policy is not relevant for tools like the FlexNet License Server Administrator command-line tool, where users must manually re-enter credentials every time they perform an operation requiring authorization.
- `security.ip.whitelist` to grant to one or more secure machines (whose IP addresses you specify for this policy) unrestricted access to the license server administrative interface.
- `security.http.auth.enabled` to control whether HTTPS is required to access secured REST APIs:
    - When set to **true**, this policy allows access to REST APIs using either the HTTPS or HTTP protocol.
    - When set to **false**, the policy enforces the use of the HTTPS protocol to access REST APIs. An error is generated when HTTP is used.

  This policy has no effect on those operations exempt from license-server security measures, as described in Operations Exempt from Security.

More Information About the Policies

Refer to the following for more information about security polices:

- For a local license server, you enable this security by setting policies in the producer settings file (see License Server Configuration Utility in Chapter 7, Producer Tools). For a CLS instance, follow the procedures described in the FlexNet Operations documentation.
- For a description of the `/configuration` REST API, see APIs to Manage Instance Configuration in Chapter 4, License Server REST APIs.
- For details about policies and their defaults, refer to Appendix A, Reference: Policy Settings for the License Server.
- The administrator of a local license server can edit these policies once the server is deployed. The policy-editing operation is described in detail in the *FlexNet Embedded License Server Administration Guide*.

Credential Requirements

The following lists the requirements for the user name and password needed for the default license-server administrator account (and any other user account).

User Name

The user name (up to 64 characters) is case-sensitive but requires no special characters.

User Password

The password for a user account is case-sensitive and must meet the following criteria:

- At least 8 characters (with a maximum of 64 characters)
- At least one digit
- At least one upper-case character
- At least one special character (for example, ^ * $ - + ? _ & = ! % { } / # @ and such)
- No whitespace

About License Server Administrator Tools

The license server administrator needs a tool that authenticates the administrator's credentials, providing access to the REST APIs needed to perform administrative functions. You can provide any of these tools:

- FlexNet License Server Administrator command-line tool. For information about using this tool, see the "Using the FlexNet License Server Administrator Command-line Tool" chapter in the *FlexNet Embedded License Server Administration Guide*.
- Your own custom REST API administration client. For information about creating a tool that supports administrative security, see Writing a REST API Client for a Secured License Server in Chapter 4, License Server REST APIs.

Enabling Administrative Security on a Local License Server

Follow these step to enable and configure administrative security on a local license server you are about to deploy.

Task ***To enable and configure administrative security on a local license server***

1. Generate the `producer-settings.xml` file to enable administrative security:
    - Be sure to set the `security.enabled` policy to **`true`** and update other security policies as needed. See Policies to Configure Administrative Security for more information.
    - When providing the **`flexnetlsconfig`** command to generate the file, include the `-admin-password` option to specify the password for the default license-server administrator account. The password must meet the criteria listed in Credential Requirements.

    Refer to License Server Configuration Utility in Chapter 7, Producer Tools, for instructions on how to use the `flexnetlsconfig` utility to update, generate, and distribute the `producer-settings.xml` file.
2. When you deploy the license server, be sure you also distribute the following:

- The license server administration tool. See About License Server Administrator Tools.
- The user name (`admin`) and password for the default license-server administrator account. Inform the administrator that these credentials are case-sensitive and that password needs to be reset using the administration tool.

7

Producer Tools

The following tools are found in the `bin/tools` directory of the FlexNet Embedded local license server software package. These tools help you to test and prepare the license server for production use by simulating an end-to-end license server process. For example, you can set up a test scenario that uses a test back-office server utility to activate license rights on the license server and then runs utilities in which “example” client devices check out licenses from the license server.

This tool set also includes utilities needed to configure the license server for production.

Tools Used to Test

- Publisher Identity Utility used to generate identity data for client product, back office, and license server
- Print Binary Utility used to display binary files or convert them to source code for embedding in the solution
- Identity Update Utility used to inject filters for hostid detection on the client
- License Conversion Utility used to create signed licenses
- Trial File Utility used to create trial licenses
- Capability Server Utility used to simulate a back-office server
- Capability Request Utility used to request and process license rights on client devices
- Capability Response Utility used to create a capability response that can be processed on the license server or client

Tools Used to Prepare the License Server for Production

- Secure Profile Utility
- License Server Configuration Utility

Publisher Identity Utility

The Publisher Identity utility, pubidutil, enables you to create producer-specific identification data in binary format. The back-office identity data is used by your back-office tools for digitally signing license rights and notification messages; the client–server identity is used by the FlexNet Embedded local license server; and the client identity data is used by your FlexNet Embedded or FlexNet Connect code to validate your license rights or notification messages, respectively, and to perform other validation operations.

Purpose

The Publisher Identity utility pubidutil enables producers to create API-compatible producer-specific identification data in binary format. This binary data is passed as a buffer to the API function FlxIdentityCreate for the creation of an Identity object. The Identity object is then associated with other API functions (such as FlxLicenseSourceCreateFromTrustedStorage in FlexNet Embedded or FlxConnectCreate in FlexNet Connect) to validate and secure your product for licensing and notification operations.

The Publisher Identity utility is provided to enable each producer to create unique identity files used to digitally sign the following:

- FlexNet Embedded license files, trial rights, and capability responses
- Notification messages sent to and from the notification-enabled client and the FlexNet Connect notification server

Three files are generated: one for the producer's back office use that contains all public and private key information (by default called IdentityBackOffice.bin), a second used by the license server (by default called IdentityClientServer.bin), and a third, used by the client code, that contains only the public keys 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 by clicking on it. It takes optional arguments that specify where the identity files will be written and whether it should run in console mode (no GUI). An argument is also available to list the supported RSA encryption strengths.

```
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 directory of the toolkit. With no arguments, it will bring up a graphical UI which can be used to generate the identity files:

```
pubidutil
```

***Note** ▪ When running "pubidutil" in console mode to generate identity files for FlexNet Connect functionality only, you are not required to include the "-clientServer" argument since the client-server identity is not used by FlexNet Connect.*

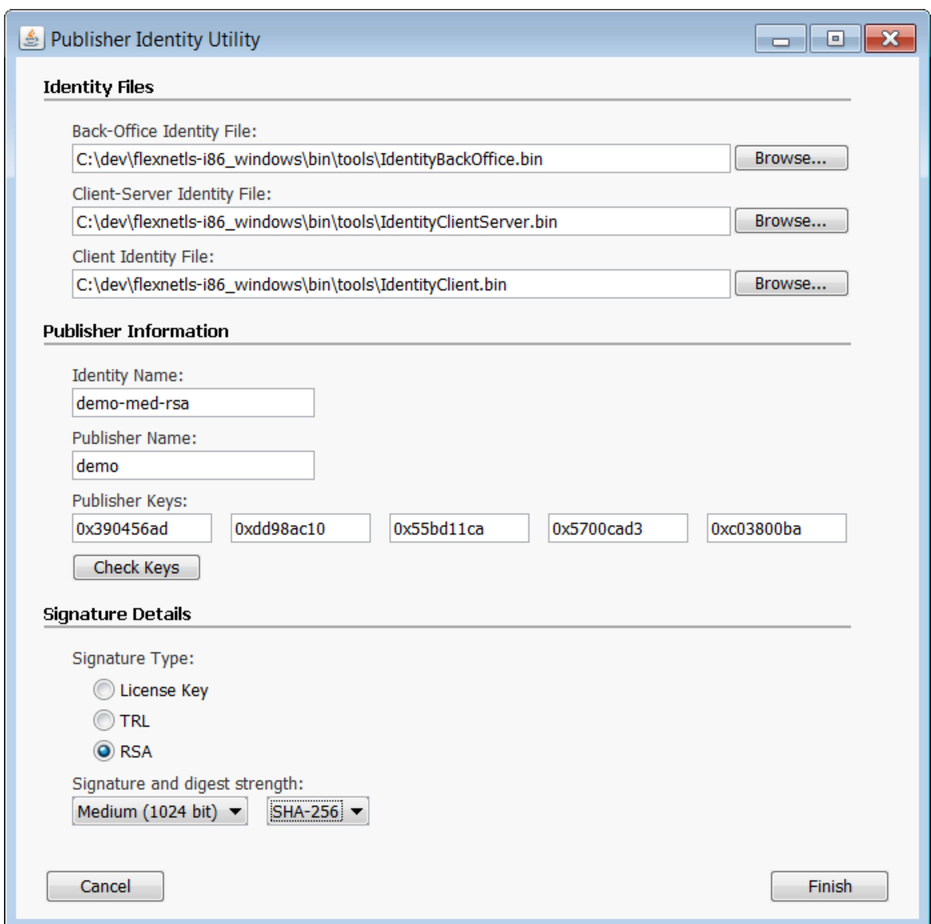


Figure 7-1: Pubidutil in GUI Mode

To avoid the use of the GUI, use the `-console` switch (running `pubidutil -?` gives additional usage information):

```
pubidutil -console
```

You will prompted for all of the required information at the command line.

If you specify existing identity files, you can modify the previous settings and then regenerate the identity files. Naturally, if you update your producer identity files, it will likely be necessary to update the client-identity buffer data in your license-enabled or notification-enabled code to correspond with the back-office identity data used in your back-office server to sign licenses.

***Caution** ▪ If you intend to use FlexNet Operations as the back-office tool, you must generate the identity information with the back-office tool and then export the client and client–server identity files for use with the FlexNet Embedded Client C SDK. 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 host types, for example).
- **Publisher Name**—Enter the producer name provided by Revenera, or "demo" for an evaluation license server software package.
- **Publisher Keys**—Enter the producer keys (five hexadecimal numbers) provided by Revenera.
- **Signature Type**—Choose RSA, TRL, or License Key (the demo software package uses RSA signatures on most architectures). The digital signature algorithm and signature strength you select are used for digitally signing licenses, capability response envelopes, and product notification messages.
- **Signature and digest strengths**—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 files you specified, or uses the default names if none were specified. (The utility reports an error if the producer keys are invalid or evaluation keys have expired.) The back-office identity file is used when signing licenses and notification messages, the client–server identity is used with served licenses, and the client-identity file is used in license-enabled and notification-enabled code.

Updating Your 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.

You can generate new identity data 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 identity data using `pubidutil` in UI mode, provide the required information as described in section Entering Your Identity Data. Make sure that you use all original settings (identity files, identity name, publisher name, and signature details), with the exception of the publisher keys which you need to replace with the new keys that you received from Revenera.

Console Mode

To update identity data using `pubidutil` in console mode, run the `pubidutil` script 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.

Further Tasks and Considerations

Once you have generated the identity data, note 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, and Java XT client applications only.)
- The `printbin` utility, described next, converts the binary client identity data into C code that can be copied into this license-enabled or notification-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 specify the back-office identity data when you start up the server.

Print Binary Utility

The print-binary utility `printbin` displays human-readable contents of binary files such as the following:

- FlexNet Embedded binary license file, capability request, capability response, or trial file created with `licensefileutil`, `caprequestutil`, `capresponseutil`, or `trialfileutil`, respectively
- Request and response messages sent between the FlexNet Connect client and the notification server

The print-binary utility also displays the contents of a binary identity file created with `pubidutil`, and optionally converts it to a Java- or C-compatible format for use in your compiled license-enabled and notification-enabled code.

Viewing Contents

To view the binary's contents (mainly keys and their values), use the command:

```
printbin binaryFile.bin
```

You can list multiple files in the `printbin` command to view their contents in a single output.

To display all the contents of the binary file (or files), use the `-full` switch.

Caution ▪ *When using the “-full” switch, be aware that some information displayed might be used internally by Revenera for troubleshooting purposes and is subject to change between releases. Do not rely on this information for your own troubleshooting or coding purposes.*

Viewing Contents and Validating Signatures

To view contents of a binary file, such as a FlexNet Embedded license file or FlexNet Connect message file, and validate any signatures contained in the file, use the command:

```
printbin -id IdentityBackOffice.bin binaryFile.bin
```

where `IdentityBackOffice.bin` is the producer identity file that you previously created with the Publisher Identity Utility and used to sign and convert the binary file.

Displaying Binary in Compiler-Readable Format

To display contents of a binary identity file in a format that can be used in C code (typically as a hard-coded array to be passed to FlexNet Embedded or FlexNet Connect functions that require the binary data), use the `-C` switch. The following shows the command used for the identity-client binary file:

```
printbin -C IdentityClient.bin
```

The resulting array will be directed to the console, or you can add the `-o` *`outputFile`* switch to save the output in a file. The output for a client-identity file should look similar to the following:

```
/*
    Publisher Identity Version = 0
    Identity Name = demo-medium-trl
    Publisher Name = demo
    Publisher Key = ...
    Signature Type = TRL
    Signature Strength = 1
*/
static const unsigned char identity_data[] = {
    0x68, 0x61, 0x70, 0x70, 0x79, 0x20, 0x6c, 0x69, 0x63, 0x65,
    0x6e, 0x73, 0x69, 0x6e, 0x67, 0x21, 0x20, 0x2d, 0x72, 0x6f,
          ...
    0x62, 0x65, 0x72, 0x74, 0x64, 0x00
};
```

You can then compile the client identity data in your FlexNet Embedded or FlexNet Connect code.

Conversion 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 call `FlxBase64Decode` before passing it to the FlexNet Embedded functionality.

***Note ▪** FlexNet Embedded uses the base-64 encoding used by MIME, where encoded data follows these requirements:*

- *Uses the letters A–Z and a–z, numerals 0–9, and "+" and "/" characters*
- *Uses "=" as padding character*
- *Encodes lines with a maximum of 76 characters*
- *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).
- `-java [-package_name]`: Option that converts and displays the binary data to Java-compatible format

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, 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.
  - Do not enable caching for dongle hostids (`flexid9` or `flexid10` hostid type) or for removable Ethernet adapters (`mac` or `mac_ecmc` hostid type)
  - 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 7-1 ▪ 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).
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).
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=0002000501010002f403
```

License Conversion Utility

The 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 host before distribution.

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 text license 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 a feature-definition syntax similar to the following. For details on the syntax for feature definitions, refer to the “Feature Definitions” section in the FlexNet Embedded client SDK user guide specific to the toolkit used to implement your licensing code.

```
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 host. 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 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 pre-loaded on a host to enable a customer to test product functionality for a limited duration. Trial license rights can also be loaded after the initial deployment to act as an emergency license.

Its usage is as follows:

```
trialfileutil -id id-file -product product-id [-expiration date] [-duration seconds]
              [-trial trial-id] [-once | -always] lic-file binary-file
```

The arguments include the following:

- `-id` *`id-file`*: File containing your producer back-office identity, required to digitally sign the trial rights
- `-product` *`product-id`*: The 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`: Trial can be loaded only once on a host; default unless `-always` is specified
- `-always`: Trial can always be loaded on a host (rarely used)
- *`lic-file`*: 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 (called `unsignedInput.lic`, for example) using a feature-definition syntax similar to the following. For details on the syntax for feature definitions, refer to the “Feature Definitions” section in the FlexNet Embedded client SDK user guide specific to the toolkit used to implement your licensing code.

```
INCREMENT survey demo 1.0 permanent uncounted
INCREMENT highres demo 1.0 permanent uncounted
```

The FlexNet Embedded binding functionality locks the trial rights to the host on which the trial is activated, and so the feature need not include a `HOSTID` value. Similarly, the trial’s expiration, whether based on duration or explicit expiration date, overrides the expiration date of all features in the unsigned text license; and the trial’s activation date overrides any start date (`START` keyword value) specified for a feature.

Next, run `trialfileutil` to generate a binary version of the trial:

```
trialfileutil -id IdentityBackOffice.bin -product SampleApp -trial 1 -duration 86400
    unsignedInput.lic signedTrialOutput.bin
```

Copy the binary output file—`signedTrialOutput.bin`, in this example—to the client system. You can now run license-enabled code that processes and acquires the trial license rights, such as the **Trials** example program. The example command uses the default trial duration of one day from the time the trial rights are processed.

Capability Server Utility

The Capability Server utility is used to test and debug the online capability exchange functionality of a FlexNet Embedded client application. The utility functions as a simple back-office server that receives `HTTP POST` capability requests from clients—FlexNet Embedded license-enabled code or local license servers—and then generates and returns capability responses. The utility also accepts synchronization messages from license servers (but generates no client records).

***Caution** ▪ The Capability Server utility is provided for testing purposes only; it is not intended for use in a production environment.*

Refer to the following for more information about using the Capability Server utility:

- Considerations for Using the Utility
- Usage
- Starting and Stopping the Capability Server Utility
- About License Templates
- Endpoint for Sending Capability Requests to the Utility

Considerations for Using the Utility

The Capability Server utility operates as a simple back-office server. Before using this utility, consider the following:

- **No license accounting**—The utility performs no accounting of license rights. That is, it does not limit the number of licenses issued. If a client requests 100 copies of particular license right (that is, license template), the utility activates 100 copies. If another client requests 100 copies of the same license right, the utility activates another 100 copies. It never responds with an "insufficient count" message.
- **No CLS support**—The utility supports the FlexNet Embedded client application and the FlexNet Embedded local license server, but not the Cloud Licensing Service (CLS) license server, as clients.
- **No client records generated**—The utility does not create or manage client records during synchronization from a license server.
- **Console mode only**—The utility runs only in console mode, not as a daemon or service.
- **Other limitations**—It does not support HTTPS, synchronization recovery, or license server failover.
- **Response lifetime**—The lifetime of a capability response generated by the Capability Server utility is 1 minute.

Usage

The following shows the usage for the Capability Server utility:

```
capserverutil -help | -id id-file -template template-dir [-port port] [-v]
```

Arguments include the following:

- `-id` *`id-file`*: The binary file containing the back-office identity, required to digitally sign the capability response.
- `-template` *`template-dir`*: The directory containing license templates that the utility uses to store and manage license rights. A sample `templates` directory is provided in the `bin\tools` directory where the utility resides, but you can specify your own location (making sure that you include the appropriate path). See About License Templates for more information.
- `-port` *`port`*: The port on which the utility listens. If no port is specified, 8080 is used.
- `-v`: The flag to provide more detail in the utility output.

Starting and Stopping the Capability Server Utility

To start the Capability Server utility, enter the **`capserverutil`** command similar to this (which, in this case, assumes the default port 8080 and specifies the detailed format for output):

```
capserverutil -id IdentityBackOffice.bin -template templates -v
```

To stop the utility, press Enter.

About License Templates

The utility uses license templates as means of organizing license rights to simulate a real back-office server system. Each template, identified by either a client device hostid (*`device_hostid`*`.lic`) or a rights ID (*`rightsID`*`.lic`), stores a set of licenses. You can create your own license templates or use the sample license templates found in the `bin\tools\templates` directory. These two sample templates, `1234567890.lic` and `li1.lic`, work easily with the example applications and test tools (included in the SDK) that use a back-office server, but you can use the samples for your own tests.

The following sections provide more information about how the Capability Server utility uses the templates to activate license rights:

- Use of License Templates to Generate Responses
- Examples
- Creating a License Template

Use of License Templates to Generate Responses

When license-enabled client code sends a capability request to the Capability Server utility, the utility follows this general process to generate a capability response.

It first searches for a license template with a name that matches the hostid of the device sending the request. If a match exists, the utility returns a capability response with the licenses found in that license template, but ignores any rights ID sent in the request.

However, if no license template matches the device hostid sending the capability request, the utility then searches for a license template with a name that matches a rights ID sent in the request. If a match exists, the utility returns a capability response with the licenses found in the license template.

Examples

The examples described next use the sample license templates, located in the `bin\tools\templates` directory, to illustrate how the Capability Server utility activates license rights. Consider the contents of these sample license templates:

- The `1234567890.lic` template contains the following licenses:

```
INCREMENT survey 1.0 permanent 1
INCREMENT highres 1.0 permanent 1
```

- The `li1.lic` template contains these licenses:

```
INCREMENT f1 1.0 permanent 5
INCREMENT f2 1.0 permanent 10
```

Keep in mind that the client in the following examples can be either FlexNet Embedded license-enabled code or a FlexNet Embedded license server.

Example 1: Activate all rights mapped to a client device

Suppose the client on device hostid “1234567890” sends a capability request without a rights ID. To simulate the search for all license rights mapped to the device hostid, the Capability Server utility looks for a license template with a name matching the hostid. When it locates license template `1234567890.lic`, it sends a capability response containing a copy of the rights in that template (that is, **1** count of `survey` and **1** count of `highres`).

A rights ID sent in the same capability request would be ignored in this case.

Example 2: Activate a a specific rights ID

Suppose the client on device hostid “1111” sends a capability request for **2** copies of the rights ID `li1`. To simulate the search for a specified rights ID, the utility first looks for a license template with a name matching the device hostid (“1111”). Finding no matching template, it locates the license template (`li1.lic`) matching the rights ID and sends a capability response containing **2** copies of all rights in that template—that is, **10** counts of `f1` (2 times the initial 5 counts) and **20** counts of `f2` (2 times the initial 10 counts).

Creating a License Template

To create a license template, you can use one of the sample license templates as a basis. In a text editor, set up an `INCREMENT` line for each feature, providing the feature’s name, version, expiration, and count, and defining any additional attributes as needed (see Feature Attributes to Consider Adding). The following shows sample contents for a license template. The same content format is used whether you are defining license rights for a FlexNet Embedded client or a license server.

```
INCREMENT f1 1.0 permanent 6
INCREMENT f2 1.0 permanent 10 VENDOR_STRING="global"
INCREMENT f3 1.0 1-jan-2025 5
INCREMENT m4 1.0 permanent 15 METERED UNDO_INTERVAL=120
```

Save the file, using a device hostid or a rights ID for the file name and adding the `.lic` extension. The utility does not discern the hostid type; it simply processes it as a string.

Keep in mind that, when a license template containing both non-metered and metered features is used to satisfy a capability request from a license server, both the metered and the non-metered features are activated on the server. If the same template is used to satisfy a request from a FlexNet Embedded client application, only the non-metered features are activated on the client (since the client can obtain metered features only through a license server).

Feature Attributes to Consider Adding

The following lists some feature attributes you might want to add for a given feature. (This is not an exhaustive list, just a list of suggestions. For more information about attributes, see the "Feature Definitions" section in your FlexNet Embedded client SDK user guide.)

- `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 capability request utility `caprequestutil` enables you to manually generate a capability request for a client device and save it as a file or send it to the license server or back office. Its common usage is:

```
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 | storage file | -response file ]
    [-activate activation_id [copies]...] [-feature name version [count] [partial]...]
    [-requestor requestor-id] [-acquisition acquisition-id] [-enterprise enterprise-id]
    [-correlation correlation-id] [-bindingBreakType binding-break-type]
    [-bindingGracePeriodEnd binding-grace-period-end] -requestAll [-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 available to the capability request utility, run **`caprequestutil`** with the **`-help`** switch. Some switches are used only for infrequently encountered scenarios and are not described in the following list.)

- `[-id` *`pubidfile`* `| -publisher` *`name`* `identity` *`identity-name`*`]`: The binary producer client identity file (normally called `IdentityClient.bin`), created using `pubidutil`; or the producer and identity name used to create the identity.
- `-host` *`host_id`*: The client hostid or transaction ID to use.

  To specify one or more secondary hostids, repeat this argument for each additional hostid. The first hostid used with this option will be considered the main hostid. Each subsequent hostid is considered secondary. When license reservations are used, only the main hostid and the first secondary hostid are used in the reservation search. (For more information, see About Main and Secondary Hostids in the License Reservations section of Chapter 5, More About Basic License Server Functionality.)
- `-idtype` *`type`*: The hostid type, one of `any`, `ethernet`, `flexid9`, `flexid10`, `internet` (for IPv4), `internet6` (for IPv6), `string` (the default), or `vm_uuid`. (Some of these types are supported only by the classic prebuilt FlexNet Embedded server application.) 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 the user guide for the FlexNet Embedded client SDK specific to your programming language.
- `-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.
- `-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.)
- `-feature` *`name version`* `[`*`count`*`] [partial]...`: One or more desired features to add to a capability request sent to a FlexNet Embedded server; the count is 1 if omitted.

  The optional `partial` attribute tells the license server to send whatever 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 Granting All Available Quantity for a Feature in Chapter 6, Advanced License Server Features, for details.

  This option can be used with the `preview` value for `-operation` to preview counts for specified features. It is not compatible with the `-requestAll` option.
- `-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-Selector Filtering in Chapter 6, Advanced License Server Features, for details.
- `-force`: The “force response” flag, which indicates that a server response is required even if license rights on the host 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-Request Processing in Chapter 6, Advanced License Server Features, 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 your FlexNet Embedded client SDK user guide.

- `-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 feature, see Binding-Break Detection with Grace Period in Chapter 6, Advanced License Server Features.
- `-bindingGracePeriodEnd` *`binding-grace-period-end`*: The timestamp indicating when the for a binding break on the license server expires. When this option is included, the SOFT status changes to a HARD status when the expires. For more information about the binding-break detection feature, see Binding-Break Detection with Grace Period in Chapter 6, Advanced License Server Features.
- `-requestAll`: The flag indicating that the client is requesting all features and their versions from the license server. (This option works only with the **`preview`** value for `-operation` and is not compatible with the `-feature` option.)
- `-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 Chapter 6, Advanced License Server Features.
- *`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://`*`LicenseServerHostName`*`:7070/request`**.
    - For a CLS license server, a typical value is **`https://`*`siteID`*`-uat.compliance.flexnetoperations.com/instances/`*`instanceID`*`/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 capability response utility `capresponseutil` enables you to manually generate a capability response without using a back office or license server. Its usage is:

```
capresponseutil -id id-file -host host_id [-idtype type] [-timestamp seconds]
    [-timestamp-milliseconds milliseconds] [-lifetime seconds] [-attr key1 val1...]
    [-machine machine-type] [-status code detail] [-vmname name] [-vmattr key value]
    [-clone] [-server server-id] [-serverIdType type] [-resetBinding]
    [-serverInstance instance-number] [-feature name version [count] [maxCount]...]
    [-preview] [-enterpriseId enterpriseId] text_license binary_response
```

The following describes commonly used arguments and switches. (For a complete list of arguments available to the capability response utility, run **`capresponseutil`** with the **`-help`** switch. Some switches are used only for infrequently encountered scenarios and are not described in the following list.)

- `-id` *id-file*: Your binary producer back-office identity file, created using `pubidutil`. This is required to sign the output response file.
- `-host` *host_id*: The hostid of the client.
- `-idtype` *type*: The hostid type of the client, one of `any`, `ethernet`, `flexid9`, `flexid10`, `internet` (for IPv4), `internet6` (for IPv6), `string` (the default), `vm_uuid`, or `publisher_defined` (for a producer-defined hostid). If using a producer-defined hostid, also include `PUBLISHER_DEFINED=`*hostid_value* in text license file specified in the command.

  ***Note** ▪ Some of these types are supported only by the classic prebuilt FlexNet Embedded server application.*

- `-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 **`02/22/2019`**. (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`*: The 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.
- `-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.
- `-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 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` (expired)

  See Binding-Break Detection with Grace Period in Chapter 6, Advanced License Server Features, for details.
- `-clone`: Designation that the client device is a clone suspect.
- `-server`: 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 feature, see Binding-Break Detection with Grace Period in Chapter 6, Advanced License Server 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 the user guide for the FlexNet Embedded client SDK specific to your programming language.
- `-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.

- -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 Chapter 6, Advanced License Server Features.
- -enterpriseId *enterpriseId*: The enterprise ID to be included in the generated capability response. An example use of this option is to aggregate licenses by the enterprise ID. The enterprise ID must match the enterprise ID generated by FlexNet Operations.
- *text_license*: The unsigned text license 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 host 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 and are available for acquisition.

***Note** ▪ 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.*

Secure Profile Utility

The FlexNet Embedded secure profile utility (secureprofileutil) configures existing identity binary data to enable a greater level of anchor security than what is normally provided for trusted storage on machines running the license server. (See Secure Anchoring in Chapter 6, Advanced License Server Features, for more information.)

Before using this utility, you must obtain the binary file containing your identity data for the license server (for example, identityClientServer.bin) from FlexNet Operations or by running the pubidutil utility (see Publisher Identity Utility).

Secure anchoring must be enabled before you generate license server policy settings, as described in License Server Configuration Utility. The updated identity data for the license server is used to generate these settings in the producer-settings.xml file.

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 local license server supports only one security profile, called server-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 the **secureprofileutil** command against the license server's identity binary data, specifying the desired specific security profile. The following shows the command syntax:

```
secureprofileutil -profile profilename input_clientServerIdentityFile.bin
    output_clientServerIdentityFile.bin
```

The arguments include:

- `-profile` *`profilename`*: Name of the security profile you want to use to implement secure anchoring. (Currently, only the `server-medium` security profile is available.)
- *`input_clientServerIdentityFile`*`.bin`: Name of the file containing the license server's identity binary data obtained from FlexNet Operations or generated using the `pubidutil` utility (see Publisher Identity Utility).
- *`output_clientServerIdentityFile`*`.bin`: Name you want to give the output file containing license server's identity binary data configured for secure anchoring.

To ensure that secure anchoring is enabled, run the `printbin` utility on the update identity file; it will include an `AnchorConfiguration` element if secure anchoring is enabled. See Print Binary Utility for more information.

License Server Configuration Utility

The FlexNet Embedded local license server configuration utility, `flexnetlsconfig`, generates a settings file (in `.xml` format) that defines your license server policies. These policies control the various operations that the license server can perform, such as synchronization to and from the back office, licensing distribution, logging, license server failover, and others. You must include this file with the distribution of the license server to the enterprise customer.

The following shows the command syntax for generating the policy settings file:

```
flexnetlsconfig -help | -version | [-prop prop_file] [-o output_file] -id -id_file [-admin-password
    -password] [-producer-password -password]
```

The arguments include:

- `-version` *`version`*: The current version of the configuration tool.
- `-prop` *`prop_file`*: Name of the file containing any policy settings that you want update or add to the file.
- *`-o output_file`*: Name you want to give the output `.xml` file containing your policy settings for the license server.

  The expected name on Windows is `producer-settings.xml`. If you generate the file with a different name, the enterprise customer must change the value of the `PUBSETTINGS` parameter in the local settings file `flexnetls.settings` distributed with the license server. If you omit this argument, the settings are written to screen output.
- `-id` *`id_file`*: The license server's identity binary data obtained from FlexNet Operations or generated using `pubidutil` (see Publisher Identity Utility). This file is required.
- `-admin-password` *`-password`*: Provides administrator credentials that are required to deploy a license server with administrative security enabled. For more information, see Considerations about Settings.

- `-producer-password` *-password*: Provides credentials for the producer account. Credentials are required to supply or delete checkout filters, and change configuration values not editable by the default administrator account (for example, `lfs.syncTo.enabled`).

You can generate a file containing only the default policy settings or a file containing customized settings.

Policy Settings Reference

For a description of each policy setting available for inclusion in this file, the setting's default value, and whether it is editable by the enterprise license server administrator, see Appendix A, Reference: Policy Settings for the License Server. Note that, by default, some available settings are not included in the default version of the settings file. Therefore, to enable and control certain types of license server functionality, you need to add specific settings to this file. The chapters More About Basic License Server Functionality and Advanced License Server Features, which describe different types of functionality available with the license server, inform you which settings you need to include if are enabling a specific function.

Generating the Policy Settings

To generate an `.xml` file listing the default policy settings, execute a command similar the following, specifying the output file name and license server's identity binary file. This command generates the `producer-settings.xml` file with the settings shown in Default Policy Settings. (In Windows, use **`flexnetlsconfig.bat`**.)

```
flexnetlsconfig -o producer-settings.xml -id IdentityClientServer.bin
```

To generate a file with customized settings, create a separate "properties" text file that defines only the settings you want to update or add. Then run a command similar to the following, where `properties1.txt` is the properties file. See Sample Properties File for Updates or Additions for an example of a properties file.

```
flexnetlsconfig -prop properties1.txt -o producer-settings.xml -id IdentityClientServer.bin
```

This command generates a policy settings file that includes the default settings plus any setting overrides and additions defined in the properties file.

Default Policy Settings

The following shows the contents of a sample file generated with default policy settings:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<configuration version="1">
     <!--
      database.backup-enabled=false
      fne.syncTo.repeats=5m
      lfs.capability.enabled=true
      lfs.capability.repeats=1d
      lfs.capability.retryCount=3
      lfs.capability.retryRepeats=30s
      lfs.syncFrom.enabled=false
      lfs.syncTo.delay=2s
      lfs.syncTo.enabled=false
      lfs.syncTo.includeAll=true
      lfs.syncTo.pagesize=50
      lfs.syncTo.repeats=5m
      lfs.syncTo.retryCount=4
```

```
        lfs.syncTo.retryRepeats=5m
        licensing.allowVirtualClients=true
        licensing.allowVirtualServer=true
        licensing.defaultBorrowGranularity=second
        licensing.hostIdValidationInterval=2m
        licensing.responseLifetime=1d
        licensing.security.json.enabled=true
        logging.directory=${base.dir}/logs
        security.anonymous=false
        security.enabled=false
        security.http.auth.enabled=true
        security.ip.whitelist=
        security.jwt.cookies=false
        security.token.duration=1d
        server.accessLogPattern=access_yyyy_mm_dd.request.log
        server.backupMaintenance.interval=3d
        server.extendedHostId.enabled=true
        server.forceTSResetAllowed=false
        server.publisherDefinedHostId.policy=disabled
        server.syncCompatibility=false
        server.trustedStorageDir=${base.dir}
        -->
     <data>
AAAZxQAAB6...
APxB...</data>
     <publisher>demo</publisher>
     <signature>rRi0HvqsIZtCZMh9lCYulIeFUj0=</signature>
</configuration>
```

Sample Properties File for Updates or Additions

When you want to edit the default settings in `producer-settings.xml`, create a properties text file that lists the policy settings you are editing or adding. List each setting on a separate line in *setting_key*=*value* format. The following contents in the properties file will add the `lfs.url` policy and reset the current value of `lfs.sycTo.enabled` to **true**:

```
lfs.url=https://demo.compliance.flexnetoperations.com/flexnet/deviceservices
lfs.syncTo.enabled=true
```

When the settings file is generated with the properties file, the updates are applied. The snippet of the settings file that follows shows that these updates have occurred (when compared with the default settings shown in Default Policy Settings):

- The setting `lfs.url=https://demo.compliance.flexnetoperations.com/flexnet/deviceservices` has been added.
- The existing setting `lfs.syncTo.enabled` has been updated from `false` to **true**.

```
<configuration version="1">
     <!--
        database.backup-enabled=false
        fne.syncTo.repeats=5m
        lfs.capability.enabled=true
        lfs.capability.repeats=1d
        lfs.capability.retryCount=3
        lfs.capability.retryRepeats=30s
```

```
lfs.syncTo.enabled=true
lfs.syncTo.includeAll=true
lfs.syncTo.pagesize=50
lfs.syncTo.repeats=5m
lfs.syncTo.retryCount=4
lfs.syncTo.retryRepeats=5m
lfs.url=https://demo.compliance.flexnetoperations.com/flexnet/deviceservices
licensing.allowVirtualClients=true
licensing.allowVirtualServer=true...
...
```

Considerations about Settings

Keep the following in mind when defining settings:

- The policy settings file is platform-independent, so choose values appropriate for the platform on which the license server will run (for example, no spaces in path names for Linux installations) or for both platforms if the license server supports both Windows and Linux.
- If the license server is using a back office (FlexNet Operations) to activate its license pool and to synchronize license distribution and usage, you must include the `lfs.url` setting to identify the URL for the back office.
- If you are migrating to the FlexNet Embedded local license server from the classic FlexNet Embedded server application, change the default `false` value for the `server.syncCompatibility` setting to **`true`**. This update is required for a successful migration and for successful synchronization activities after migration.
- If you want to deploy a license server with administrative security enabled, set the `security.enabled` policy to **`true`**. In the **`flexnetlsconfig`** command, you must also include the `-admin-password` option to specify the password for the default administrator account. The password is not listed in the generated `producer-settings.xml` file for security purposes. For more information about administrative security, see Administrative Security in Chapter 6, Advanced License Server Features.

Distributing this File

To install the license server, this file must reside in the same directory as the `flexnetls` batch or shell file (found in the `server` directory in your license server software package). Make sure this file is placed in the correct location when the license server is distributed to the enterprise customer.

Updating the File After Server Installation

If you update the policy settings file and redistribute it, the license server that runs with this new file must be stopped and restarted to put the new policies into effect.

8

Troubleshooting Reference

This chapter offers some troubleshooting measures for issues commonly encountered when using the FlexNet Embedded local license server.

- If you install the license server as a Window service, the server runs under the user context of the network service. When you then try to reset the trusted storage database using the **flexnetls -reset-database** command, you will be running under the current user context and will not be able to reset the trusted storage database.

  As a workaround:

  - For the server.trustedStorageDir setting in the producer-settings.xml file, define an explicit value for the default trusted-storage location that is user-independent. (Currently, the default value $ {base.dir} is different for each user. The license server gets the USERPROFILE value, appends **flexnetls\producer**, and uses this path to name the trusted storage directory.)
  - Ask the enterprise license server administrator to configure the *base.dir* value. To do this, the administrator adds the BASE_DIR property to flexnetls.settings and defines a specific path for this property.

    When you explicitly set the value of *base.dir*, the license server does not append flexnetls*producer* to the value. However, when *base.dir* is not explicitly set, the server uses flexnetls*producer* to segregate data for multiple producers who do not set this directory.

- When you start the license server in HTTP mode, but attempt to access REST endpoints using the HTTPS mode, an error is generated and recorded in the license server log (an excerpt of which is shown here):

  ```
  badMessage: 400 Illegal character for HttpChannelOver...
  ```

  As a workaround, either start the license server in HTTPS mode, or use only HTTP to access the REST endpoint.

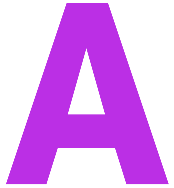
- When you attempt to start the license server installed in a Linux environment, the startup might fail with an error similar to the following, as recorded in the license server log:

  ```
  java.net.UnknownHostException: machinename: machinename: Temporary failure in name resolution"
  [90028-179]
  ```

As a possible workaround, check the `hosts` file in `/etc` and determine whether the host has a name-resolution line added. If not, then add the line, which includes the machine's IP address and hostname, as shown in this example:

```
10.99.999.999 CentOS63x64999
```

- If the license server has not been provisioned with any features, it responds to client requests with a 503 error with the error message "`Server instance <instanceID> is not ready`". This is expected behaviour because the server is not ready in this state.



Reference: Policy Settings for the License Server

The following chart serves as reference to the various settings that you can include in the policy settings file, `producer-settings.xml`, which you create and distribute with the FlexNet Embedded local license server. These settings define policies for the various operations that the license server can perform, such as synchronization to and from the back office, licensing distribution, logging, license server failover, and others. (You use the license server configuration utility to generate this file, as described in License Server Configuration Utility in Chapter 7, Producer Tools.)

Note that this chart also includes policy settings related to license server failover, even though these settings cannot be set using `producer-settings.xml`. Instead, these settings can be edited using the FlexNet License Server Administrator command-line tool or the **/configuration** API.

In the chart, the specific settings that enterprise license-server administrator can override show “Yes” in the “Editable?” column. To view all settings and to override editable ones, the administrator must have access to the `/configuration` REST API endpoint (see APIs to Manage Instance Configuration in *Chapter 4, License Server REST APIs*). The administration tools included in the license server software package—and that you can make available to the enterprise administrator—provide facilities that access this API, enabling the administrator to view and override settings (see the *FlexNet Embedded License Server Administration Guide*).

Table A-1 ▪ License Server Policy Settings

Setting	Description	Edit-able?
database.backup-enabled	The property that determines whether the license server takes a backup of trusted storage at given times and stores it on the server. Should trusted storage become corrupt, the license server administrator can then restore it from the backup without contacting the back office. See Trusted Storage Backup and Restoration for details. (Default is `false`.) ***Note ▪*** *Note that regular trusted-storage backups can negatively affect the license server performance.*	No

Table A-1 ▪ License Server Policy Settings (cont.)

Setting	Description	Editable?
Licensing Policies		
licensing.clientExpiryTimer	The frequency of checks for expired features on clients. (An expired feature is one whose borrow interval has expired or that has reached its expiration date as defined in the back office.) Expired features found during a given check are returned to the license-server feature pool. The frequency value can be specified with an optional unit-suffix letter—`s`, `m`, `h`, `d`, or `w`—indicating seconds, minutes, hours, days, or weeks. If no suffix is used, the server assumes the value is in seconds. The default frequency for these checks is `2s`. Consider increasing this value if the current frequency is interfering with capability-request processing or with overall throughput, particularly when features have large borrow intervals. (Minimum is `1s`.)	Yes
licensing.dropClientEnforcedDelay	The delay that is enforced between client deletion requests. This value can be specified with an optional unit-suffix letter—`s`, `m`, `h`, `d`, or `w`—indicating seconds, minutes, hours, days, or weeks. If no suffix is used, the server assumes the value is in seconds. This setting can also be used to disallow the deletion of clients; in this case, set the value `DROP_CLIENT_DISALLOWED`. (Default is `0s`, meaning deleting client records is allowed and there is no enforced delay between deletions.)	No
licensing.responseLifetime	The lifetime of a served-license response on the client. This value can be specified with an optional unit-suffix letter—`s`, `m`, `h`, `d`, or `w`—indicating seconds, minutes, hours, days, or weeks. If no suffix is used, the server assumes the value is in seconds. If this value is `0` (zero), the response has an unlimited lifetime. (Default is `1d`.)	No
licensing.allowVirtualClients	The property that determines whether virtual client devices are allowed to obtain licenses. (Default is `true`.)	No
licensing.allowVirtualServer	The property that determines whether the license server is allowed to run on a virtual host. (Default is `true`.)	No

Table A-1 ▪ License Server Policy Settings (cont.)

Setting	Description	Editable?
licensing.defaultBorrowGranularity	The time unit to which the borrow interval used by the license server rounds up. Valid values include `day`, `hour`, `minute`, or `second`. (Default is `second`.) For example, if the borrow interval (which is always expressed in seconds) is 60 seconds, and the borrow granularity is **`day`**, then a license issued at 5:05:01 PM expires at 11:59:59 PM—the borrow interval (5:06:01 PM) rounded to the *end of the nearest day*. Likewise, if granularity is **`minute`**, expiration is at 5:06:59 PM. If the granularity is **`second`**, expiration is 5:06:01 PM. This setting is used for those clients that do not specify one in their capability requests. ***Note ▪** For FlexNet Embedded client SDKs released before version 4.0, the granularity is always “day”, regardless of this setting.*	No
licensing.borrowInterval	The borrow interval for served licenses. This *server borrow interval* is only considered if the back office does not specify a borrow interval for a feature within the license model. The current version of FlexNet Operations mandates that a borrow interval (also referred to as the *feature borrow interval*) be specified for new license models. This value can be specified with an optional unit-suffix letter—`s`, `m`, `h`, `d`, or `w`—indicating seconds, minutes, hours, days, or weeks. If no suffix is used, the server assumes the value is in seconds. (Default is `1w`.) For information on how to determine the effective borrow interval, see licensing.borrowIntervalMax.	No

Table A-1 ▪ License Server Policy Settings (cont.)

Setting	Description	Edit-able?

licensing.borrowIntervalMax	Restricts the borrow period of the clients. This value can be specified with an optional unit-suffix letter—`s`, `m`, `h`, `d`, or `w`—indicating seconds, minutes, hours, days, or weeks. If no suffix is used, the server assumes the value is in seconds. Default is `NOT_CONFIGURED` (`0`). This is also referred to as the *admin borrow interval*.
	The following helps you determine the effective borrow interval.
	• If the feature borrow interval has been set in the back office, the 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 `licensing.borrowIntervalMax`)
	• If the feature borrow interval has *not* been set in the back office, the borrow interval is the lowest of the following values:
	  ◦ server borrow interval (defined in `producer-settings.xml` by the property `licensing.borrowInterval`)
	  ◦ client borrow interval (set in a client capability request)
	  ◦ admin borrow interval (set using `licensing.borrowIntervalMax`)
	A feature’s current borrow expiration can never exceed the final expiration time for that feature. In addition, a borrow-interval granularity may be applied to the effective borrow interval.
	This parameter cannot be used for metered features.
		Yes

Table A-1 ▪ License Server Policy Settings (cont.)

Setting	Description	Edit-able?
licensing.renewInterval	The default renew interval is set as a percentage of the effective borrow interval. It is a security setting that may be used to require the served clients to contact the license server. This value specifies how often—if ever—the client may attempt to recontact the local license server. Successful contact extends the expiration based on the effective borrow period (in other words, the timer for the effective borrow interval is restarted). If set to zero, the renew interval is at client discretion. (Default is `15`.) For information on how to determine the effective borrow interval, see licensing.borrowIntervalMax. ***Important ▪** This specification by itself does not lead to enforcement. The client-side APIs must extract this value from the license server capability response and take appropriate action.* **Example** Effective borrow interval = `604800s` (1 week); renew interval = `15` Suppose the device initially obtains its license on January 1. The 7-day effective borrow interval results in a license that is valid until January 8. The 15% renew interval indicates that the device should attempt to recontact the server after 90720 seconds, that is, on January 2 (15% of 604800 seconds = 90720 seconds = approx. 1 day). If successful, the license is now valid until January 9. If unsuccessful, the license remains valid until the end of the original borrow period, which is still January 8.	No
licensing.hostIdValidationInterval	The frequency with which the license server validates that its hostid has not changed. This value can be specified with an optional unit-suffix letter—`s`, `m`, `h`, `d`, or `w`—indicating seconds, minutes, hours, days, or weeks. If this value is `0` (zero), validation is disabled. (Default is `2m`.)	No

Table A-1 ▪ License Server Policy Settings (cont.)

Setting	Description	Edit-able?
licensing.defaultTimeZone	Defines the time zone that will be applied when determining a feature's expiry date, start date, and issue date. Valid values: • **UTC**— If UTC is set, a feature's start date is the start of the specified day in Coordinated Universal Time (UTC). Equally, a feature will expire at the end of the day of the configured expiry date in UTC time. This is the default value. • **SERVER**—If SERVER is set, a feature's start date is the start of the specified day in the server's default time zone. Equally, a feature will expire at the end of the day of the configured expiry date in the server's default time zone. See Setting the Server Time Zone for additional information.	No
licensing.security.json.enabled	The option that enables (`true`) or disables (`false`) security for JSON capability exchanges used in REST-driven licensing. When the value is **`true`**, a token signed by an enterprise private key must be attached to each capability request, allowing the license server to authenticate the request against matching public key data stored on the server. See API for Enabling JSON Security for the Cloud Monetization API in *Chapter 4, License Server REST APIs* for details. (Default is `true`.)	Yes
licensing.backup.uri	(Defined on back-up or main license server in a failover configuration; optional) The URI of the back-up server to be included in the capability response to the client device. Use the following format: `http://`*`server:port`*`/fne/bin/capability` where *server*:*port* is the back-up license server's name and port number, as in: **`http://22.22.2.222:7070/fne/bin/capability`**	Yes
licensing.main.uri	(Defined on back-up or main license server in a failover configuration; optional) The URI of the main server to be included in the capability response to the client device. Use the following format: `http://`*`server:port`*`/fne/bin/capability` where *server*:*port* is the main license server's name and port number, as in: **`http://11.11.1.111:7070/fne/bin/capability`**	Yes
License Server Settings		
server.uuid	Host UUID that uniquely identifies this server instance when communicating with the back office.	No

Table A-1 ▪ License Server Policy Settings (cont.)

Setting	Description	Edit-able?
server.trustedStorageDir	The directory in which trusted storage resides. (Default is `${base.dir}`, which points to the `flexnetls/`*`producer_name`* folder in the service's or user's home directory.)	No
server.publisherDefinedHostId.policy	The property that determines whether to enable support for the use of a producer-defined hostid to identify the license server. To enable support, use the value **`STRICT`**. (Default is `false`, meaning support for this feature is disabled.) If the property is set to **`STRICT`**, the `server.hostType.order` property (if set) is ignored.	No
server.extendedHostId.enabled	The property that enables support for the use of extended hostids to identify the license server. (Default is `true`.)	No
server.hostType.order	The property that enables producers to specify the order in which the local license server picks the hostid type. The order of the hostid types is specified as a comma-separated list, for example: `server.hostType.order=ETHERNET,FLEXID9,FLEXID10,VM_UUID` Valid values are all hostid types, with the exception of producer-defined hostid types. The following settings override the `server.hostType.order` property: • **server.publisherDefinedHostId.policy**—If a producer-defined hostid type is set using the `server.publisherDefinedHostId.policy` property, the `server.hostType.order` property is ignored. • **ACTIVE_HOSTID**—If `ACTIVE_HOSTID` is set either through the REST API or using a configuration file during server startup, the `server.hostType.order` property is ignored.	No
server.forceTSResetAllowed	The property that determines whether trusted storage can be reset when unsynchronized data still exists on the license server. (Default is `false`.)	No
server.backupMaintenance.interval	(Defined on back-up license server in a failover configuration; required) The maximum amount of time that the back-up server can serve licenses in a failover event. This value can be specified with an optional unit-suffix letter—`s`, `m`, `h`, `d`, or `w`—indicating seconds, minutes, hours, days, or weeks. If no suffix is used, the server assumes the value is in seconds. If this value is set to `0`, the back-up license server will serve licenses for an unlimited time while in failover mode. (Default is `3d`.)	No

Table A-1 • License Server Policy Settings (cont.)

Setting	Description	Editable?
server.syncCompatibility	(Used for migration from the classic FlexNet Embedded server application) The property that enables proper conversion of time units used for synchronization to and from the back office during the migration from the FlexNet Embedded server application to the FlexNet Embedded local license server. (Default is `false`.)	No
binding.break.policy	(Used to enable the binding-break detection feature described in Binding-Break Detection with Grace Period in Chapter 6, Advanced License Server Features.) The type of action to take when a binding break is detected in any of the producer-specified binding elements: • **hard**—An immediate hard binding break is imposed—that is, the license server can no longer serve licenses. • **soft**—A soft binding break is in effect, allowing the license server to continue to serve licenses despite the break. • **interval**—A soft binding break with a grace period is in effect, allowing the license server to serve licenses until the grace period expires. Then a hard binding break goes into immediate effect, and licenses can no longer be served. The length of the grace period is the *interval* value, specified with an optional unit-suffix letter—`s`, `m`, `h`, `d`, or `w`—indicating seconds, minutes, hours, days, or weeks (for example, `3d` is 3 days). If no suffix is used, the server assumes the value is in seconds. Once set, this policy is not visible in `producer-settings.xml`. However, the binding status is viewable through the license server's `/binding` REST API and in the license server log. It is also reported to the FlexNet Embedded clients and the back office through capability exchanges with the license server.	No
Back Office URL		
lfs.url	The URL for back office to which the license server sends capability requests and synchronization data. The property is required for the online deployment model of the license server.	No
Policies for Polling Back Office for License Updates		
lfs.capability.enabled	The property that determines whether capability-request polling is enabled. If polling is enabled, a capability request is sent to the back office periodically to update the license server's license rights. This property is used for the online deployment model of the license server. (Default is `true`.)	No

Table A-1 ▪ License Server Policy Settings (cont.)

Setting	Description	Edit-able?
lfs.capability.repeats	The amount of time between polling sessions to the back office. The value can be specified with an optional unit-suffix letter—`s`, `m`, `h`, `d`, or `w`—indicating seconds, minutes, hours, days, or weeks. If no suffix is used, the server assumes the value is in seconds. (Default is `1d`; minimum is `10s`.)	Yes
lfs.capability.retryCount	The number of polling attempts allowed if the initial attempt fails. (Default is `3`.)	Yes
lfs.capability.retryRepeats	The amount of time between polling attempts, if the initial attempt fails. The value can be specified with an optional unit-suffix letter—`s`, `m`, `h`, `d`, or `w`—indicating seconds, minutes, hours, days, or weeks. If no suffix is used, the server assumes the value is in seconds. (Default is `30s`; minimum is `1s`.)	Yes
Policies for Synchronizing to Back Office		
lfs.syncTo.enabled	The property that determines whether synchronization to the back office is enabled. This property should be viewed in combination with `lfs.syncTo.includeAll`: • `lfs.syncTo.enabled=true` and `lfs.syncTo.includeAll=true`: (Online synchronization) This mode collects all historical client actions in the synchronization history and uploads this data to the back office as part of the synchronization. • `lfs.syncTo.enabled=true` and `lfs.syncTo.includeAll=false`: (Online synchronization) This mode collects only the current state for each active client device at the point of synchronization and uploads this data to the back office • `lfs.syncTo.enabled=false` and `lfs.syncTo.includeAll=true`: (Offline synchronization) This mode collects all historical and current client actions. This data is retained on the license server until the offline synchronization tools are run (see Offline Synchronization to the Back Office). • `lfs.syncTo.enabled=false` and `lfs.syncTo.includeAll=false`: No synchronization data is collected (synchronization is disabled). Client data is deleted from the license server as soon as the client expires. (Default is `false`.)	No
lfs.syncTo.pagesize	The maximum number of client records to include in a synchronization message to the back office. A smaller page size limits the memory overhead at the expense of having multiple synchronization transactions. (Default is `50`; minimum is `10`; maximum is `256`.)	Yes

Table A-1 ▪ License Server Policy Settings (cont.)

Setting	Description	Edit-able?
lfs.syncTo.threads	The number of parallel threads allocated to handle the synchronization of metered-usage and license-distribution data to the back office. (Default is `1`.)	Yes
lfs.syncTo.repeats	The amount of time between synchronization sessions to the back office. The value can be specified with an optional unit-suffix letter—`s`, `m`, `h`, `d`, or `w`—indicating seconds, minutes, hours, days, or weeks. If no suffix is used, the server assumes the value is in seconds. (Default is `5m`; minimum is `10s`.)	Yes
lfs.syncTo.retryCount	The number of synchronization attempts to the back office allowed when an initial attempt fails. (Default is `4`.)	Yes
lfs.syncTo.retryRepeats	The amount of time between synchronization attempts when an initial attempt fails. The value can be specified with an optional unit-suffix letter—`s`, `m`, `h`, `d`, or `w`—indicating seconds, minutes, hours, days, or weeks. If no suffix is used, the server assumes the value is in seconds. (Default is `5m`; minimum is `1s`.)	Yes
lfs.syncTo.delay	At license server startup, the amount of time the server should wait before initiating a synchronization session to the back office. (Default is `2s`; minimum is `2s`.)	Yes
lfs.syncTo.includeAll	The property that determines whether historical license-distribution data for concurrent features is collected and sent to the back office as part of the synchronization. This property should be viewed in combination with `lfs.syncTo.enabled`: • `lfs.syncTo.enabled=true` and `lfs.syncTo.includeAll=true`: (Online synchronization) This mode collects all historical client actions in the synchronization history and uploads this data to the back office as part of the synchronization. • `lfs.syncTo.enabled=true` and `lfs.syncTo.includeAll=false`: (Online synchronization) This mode collects only the current state for each active client device at the point of synchronization and uploads this data to the back office • `lfs.syncTo.enabled=false` and `lfs.syncTo.includeAll=true`: (Offline synchronization) This mode collects all historical and current client actions. This data is retained on the license server until the offline synchronization tools are run (see Offline Synchronization to the Back Office). • `lfs.syncTo.enabled=false` and `lfs.syncTo.includeAll=false`: No synchronization data is collected (synchronization is disabled). Client data is deleted from the license server as soon as the client expires. (Default is `true`.)	No

Table A-1 ▪ License Server Policy Settings (cont.)

Setting	Description	Edit-able?
Policies for Synchronizing from Back Office		
lfs.syncFrom.enabled	The property that determines whether license-recovery from the back office is enabled. If recovery is enabled, the metered-usage data and license-distribution state for concurrent features is recovered from the back office when the license server initially starts up with a new or reset trusted storage. (Default is `false`.)	No
Policies for License Server Failover These settings are normally set by the license server administrator on the back-up server in a failover configuration. The administrator can set these using the FlexNet License Server Administrator command-line tool. Note that these settings cannot be set using `producer-settings.xml`. As a producer, you can edit these settings via the **/configuration** API. The setting **fne.syncTo.repeats** can only be edited via the **/configuration** API.		
fne.syncTo.enabled	(Defined on back-up license server only; required) The property that determines whether to enable “license server to license server” synchronization in a failover configuration. (Default is `false`.)	Yes

fne.syncTo.mainUri	(Defined on back-up license server only; required) The URI of the main license server in a failover configuration. Use the following format:
	`http://`*`server`*`:`*`port`*`/fne/bin/capability`
	where *server*:*port* is the main license server’s name and port number, as in:
	`http://11.11.1.111:7070/fne/bin/capability`
		Yes

fne.syncTo.repeats	(Can only be set using the **/configuration** API; defined on back-up license server only) The amount of time between synchronization sessions from the main server to the back-up server. (The back-up server initiates the sessions.) The value can be specified with an optional unit-suffix letter—`s`, `m`, `h`, `d`, or `w`—indicating seconds, minutes, hours, days, or weeks. If no suffix is used, the server assumes the value is in seconds. (Default is `300s`; minimum is `5m`.)	No
fne.syncTo.pagesize	(Defined on back-up license server only) The maximum number of client records to include in a synchronization message to the back-up server. A smaller page size limits the memory overhead at the expense of having multiple synchronization transactions. (Default is `100`.)	Yes
fne.syncTo.retryCount	(Defined on back-up license server only) The number of synchronization attempts from the main server allowed when an initial attempt fails. (Default is `1`.)	Yes

Table A-1 ▪ License Server Policy Settings (cont.)

Setting	Description	Edit-able?
fne.syncTo.retryRepeats	(Defined on back-up license server only) The amount of time between synchronization attempts when an initial attempt fails. The value can be specified with an optional unit-suffix letter—`s`, `m`, `h`, `d`, or `w`—indicating seconds, minutes, hours, days, or weeks. If no suffix is used, the server assumes the value is in seconds. (Default is `60s`.)	Yes
Security Policies		
security.enabled	The option that enables (`true`) or disables (`false`) administrative security on the license server. When administrative security is enabled, REST APIs used to administer the license server are “secured” (that is, credentials are required to access them). See Managing Administrative Security on a Local License Server or CLS Instance. When this option is **`true`**, the remaining policies in this *Security Policies* section are in effect. (Default is `false`.)	Yes
security.token.duration	(Available only when `security.enabled` is **`true`**) The duration of the JSON Web Token (generated when a user successfully authenticates credentials to access secured REST API endpoints on the license server). When the token expires, credentials must be re-entered to re-authorize. The value can be specified with an optional unit-suffix letter—`s`, `m`, `h`, `d`, or `w`—indicating seconds, minutes, hours, days, or weeks. If no suffix is used, the server assumes the value is in seconds. (Default is `1d`.)	Yes
security.http.auth.enabled	(Available only when `security.enabled` is **`true`**) The option that enforces the use of HTTPS to access secured REST APIs. • When **`false`**, the policy enforces the use of HTTPS to access REST API endpoints. (An error is generated during attempts to access REST API with HTTP.) • When **`true`**, the policy allows either HTTP or HTTPS to access REST API endpoints. This policy also enforces the use of HTTPS when calling the `/access_request` REST API used to perform a JSON capability exchange, as described in API for Enabling JSON Security for the Cloud Monetization API. (Default is `true`.)	Yes

Table A-1 ▪ License Server Policy Settings (cont.)

Setting	Description	Edit-able?
security.ip.whitelist	(Available only when `security.enabled` is `true`) The list of IP addresses for those components (devices) that you determine should have access to the license server without having to provide credentials to access secured REST API endpoints. For example, you might want a machine in your IT department to have such access to the endpoints for fixing issues or performing maintenance. List only IP4 or IP6 addresses; and separate each address with a comma, as this example value shows: `111.222.2.2,111.333.3.3`	Yes
security.anonymous	(Available only when `security.enabled` is `true`) The option that determines whether or not users need credentials for "read" access to the secured REST API endpoints. ● When the value is `true`, all user accounts are automatically given "read" rights (ROLE_READ) and do not need to provide credentials for "read" access. ● When the value is `false`, a given user account must be explicitly assigned ROLE_READ in order to perform "read" operations. (The exception occurs when no role is assigned to an account, in which case ROLE_READ is assigned as the only role by default.) Credentials are then required to perform any "read" operation. If an account is not authorized for ROLE_READ, no "read" access is given. This setting provides additional protection against unauthorized queries on the license server. (Default is `false`.)	Yes
Logging Policies		
logging.directory	The directory to which the license server writes the log for the license server. The default is `${base.dir}/logs`, where `${base.dir}` points to the `flexnetls/`*`producer_name`* folder in the service's or user's home directory.	No

Table A-1 • License Server Policy Settings (cont.)

Setting	Description	Edit-able?
logging.threshold	The lowest level of log-message granularity to record—FATAL, ERROR, WARN, INFO, LICENSING, POLICY, or DEBUG. For example, if **FATAL** is set, only messages about fatal events are recorded. However, if **WARN** is set, fatal-event, error, and warning messages are recorded. The POLICY threshold records information about the checkout process when feature selectors are used to filter features (see Feature-Selector Filtering). This threshold can be especially useful for monitoring checkouts when an MVEL script is used to manage feature-selector priorities. (Default is INFO.) **Logging categories** **FATAL**—Errors that prevent the server from starting up **ERROR**—Serious errors **WARN**—Warnings **INFO**—Informational messages **LICENSING**—Server responses such as, for example, capability responses and JSON replies **POLICY**—Additional information for checkout filters **DEBUG**—Additional debug-level information. The license server should not use a logging level of **DEBUG** for a long period, because it can have a negative impact on license server performance. It is not recommended to use **DEBUG** on production license servers.	Yes
graylog.host	The host name of a Graylog server, if any, to which logging messages are sent.	Yes
graylog.threshold	The lowest level of log-message granularity to record—FATAL, ERROR, WARN, INFO, LICENSING, POLICY, or DEBUG. For example, if **FATAL** is set, only messages about fatal events are recorded. However, if **WARN** is set, fatal-event, error, and warning messages are recorded. (Default is WARN.)	Yes
Anchor Policies		

Table A-1 ▪ License Server Policy Settings (cont.)

Setting	Description	Edit-able?
anchoring.request.counter.limit	The threshold for the maximum number of capability requests received from clients at which the anchor on the license server is updated. This threshold is used in conjunction with `anchoring.time.limit threshold`. Whichever threshold is reached first triggers an anchor update. When changing the default for either policy, consider that an anchor update incurs overhead. If you have also enabled trusted-storage backups on the license server (`database.backup-enabled` is **`true`**), a backup is performed each time the anchor is updated. You need to balance the frequency of anchor updates with your security requirements and the enterprise threshold for possible data loss during trusted-storage restoration. (Default and minimum is `1000`.)	No
anchoring.time.limit	The threshold specifying the maximum amount of time to elapse before the anchor on the license server is updated. The value can be specified with an optional unit-suffix letter—`s`, `m`, `h`, `d`, or `w`—indicating seconds, minutes, hours, days, or weeks. If no suffix is used, the server assumes the value is in seconds. This threshold is used in conjunction with `anchoring.time.limit threshold`. Whichever threshold is reached first trigger an anchor update. (Default and minimum is `60s`.)	No

Effects of Special Request Options on Use of Reservations

Two options that you can enable in a capability request affect how the license server processes the request when reservations are used. (For more about standard request processing, see Processing the Capability Request When Reservations Are Used in Chapter 5, More About Basic License Server Functionality.) The following describes the basic behavior of these options, whether license reservations are used or not:

- **Incremental capability request**—When the license server processes a capability request marked as "incremental", it attempts to renew all licenses currently served to the client and include them in the capability response along with the requested desired features. By default, the incremental flag is not included in the request; the server considers any licenses currently served to the client as "returned" and sends a capability response with the requested desired features only. For more details, see Incremental Capability-Request Processing in Chapter 6, Advanced License Server Features.
- **Partial-checkout feature**—If a desired feature is marked as "partial" in the capability request, the license server can send whatever 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 enabled for a desired feature, the license server grants the feature only if the count requested for that feature is available on the server. For more details, see Granting All Available Quantity for a Feature in Chapter 6, Advanced License Server Features.

To help you understand the effects that these options have on how the license server grants licenses when reservations are available, refer to the following sections describing scenarios:

- Scenario Assumptions
- Incremental Option Enabled
- Partial-Checkout Option Enabled
- Both Incremental and Partial-Checkout Options Enabled

For the sake of simplicity, the feature version is omitted in these scenarios. Anytime feature `F1` is mentioned, assume that its version is 1.0.

Scenario Assumptions

The scenarios are based on the following assumptions about the license server and the current feature distribution:

- Feature Allocation on the License Server in the Scenarios
- Starting State for Each Scenario

Feature Allocation on the License Server in the Scenarios

The license server has 10 counts of feature F1:

- 3 reserved counts for device D1
- 2 reserved counts for user U1
- 5 shared counts available

Starting State for Each Scenario

The starting state for each scenario is a follows:

- Device D1 has been served 4 counts (all 3 reserved counts, 1 shared count).
- Device D2 has been served 3 counts (3 shared counts).
- User U1 currently has no served counts.
- 1 shared count remains available.

Incremental Option Enabled

When a capability request is marked as "incremental", the server first searches the reserved and shared counts to satisfy whatever count is needed to preserve current licenses on the client device. Then, if desired features are specified in the request, the license server attempts to grant the requested features from the remaining reserved and shared counts.

If no desired features exist in the request, the license server attempts to satisfy only the feature counts needed to preserve current licenses on the client device. Any remaining reserved features are not automatically served when the "incremental" option is enabled. (For a description of normal processing, see Basic Process for Granting Licenses and its related scenarios.)

The following scenarios assume that the capability request is marked as "incremental".

- Scenario 1—Incremental: No Device or User Reservations Available
- Scenario 2—Incremental: Device Reservations But No User Reservations Available
- Scenario 3—Incremental: User Reservations But No Device Reservations Available
- Scenario 4—Incremental: Device and User Reservations Available

Scenario 1—Incremental: No Device or User Reservations Available

When user U7 (who has no reservations) sends a incremental capability request from device D2 (which also has no reservations), the 3 counts of F1 currently served to D2 are returned to the shared counts. (Four shared counts are now available.)

The following describes possible scenarios of what happens next:

- If the incremental request has no desired features, 3 shared counts are served to preserve the previous 3 counts on the device.
- If the incremental request specifies 1 count of F1, 3 shared counts are served to preserve the previous count. Additionally, the 1 remaining available shared count is served as the 1 desired count.
- If the incremental request specifies 2 counts of F1, 3 shared counts are served to preserve the previous count. However, no additional counts are served to satisfy the 2 desired counts since only 1 shared count remains available.

Scenario 2—Incremental: Device Reservations But No User Reservations Available

When user U7 (who has no reservations) sends a capability request from device D1, the 4 counts of F1 currently served to D1 (3 reserved counts for D1 and 1 shared count) are returned. (Two shared counts are now available.)

The following describes possible scenarios of what happens next:

- If the incremental request has no desired features, the 3 counts of F1 reserved for D1 plus 1 shared count are served to preserve the previous 4 counts on the device.
- If the incremental request specifies 1 count of F1, the 3 counts reserved for D1 plus 1 shared count are served to preserve the previous count. Additionally, the 1 remaining available shared count is served as the 1 desired count.
- If the incremental request specifies 2 counts of F1, the 3 counts reserved for D1 plus 1 shared count are served to preserve the previous count. However, no additional counts are served to satisfy the 2 desired counts since only 1 shared count remains available.

Scenario 3—Incremental: User Reservations But No Device Reservations Available

When user U1 sends a capability request from device D2 (which has no reservations), the 3 counts of F1 currently served to D2 are returned to shared counts. (Four shared counts are now available.)

The following describes possible scenarios of what happens next:

- If the incremental request has no desired features, the 2 counts of F1 reserved for U1 plus 1 shared count are served to preserve the previous 3 counts on the device.
- If the incremental request specifies 3 counts of F1, the 2 counts of F1 reserved for U1 plus 1 shared count are served to preserve the previous count. Additionally, the remaining 3 available shared counts are served to satisfy the desired 3 counts.
- If the incremental request specifies 4 counts of F1, the 2 counts of F1 reserved for U1 plus 1 shared count are served to preserve the previous count. However, no additional counts are served to satisfy the 4 desired counts since only 3 shared counts remain available.

Scenario 4—Incremental: Device and User Reservations Available

If user U1 sends a capability request from device D1, the 4 counts of F1 currently served to D1 (3 reserved and 1 shared) are returned. (Two shared counts are now available.)

The following describes possible scenarios of what happens next:

- If the incremental request has no desired features, the 3 counts of F1 reserved for D1 plus 1 of the 2 reserved counts for U1 are served to preserve the previous 4 counts on the device. However, the remaining 1 reserved count (for U1) is not automatically served once the preserved count is satisfied.
- If the incremental request specifies 2 counts of F1, the 3 counts reserved for D1 and 1 of the 2 counts reserved for U1 are served to preserve the previous count. Additionally, the remaining 1 reserved count for U1 and 1 shared count are served to satisfy the desired 2 counts.
- If the incremental request specifies 4 counts of F1, the 3 counts reserved for D1 and 1 of the 2 counts reserved for U1 are served to preserve the previous count. However, no additional counts are served to satisfy the 4 desired counts since only 1 reserved count for U1 and 2 shared counts remain available.

Partial-Checkout Option Enabled

The following scenarios assume that the feature F1 is enabled for partial checkout:

- Scenario 1—Partial Checkout: No Device or User Reservations Available
- Scenario 2—Partial Checkout: Device Reservations But No User Reservations Available
- Scenario 3—Incremental: User Reservations But No Device Reservations Available
- Scenario 4—Partial Checkout: Device and User Reservations Available

Scenario 1—Partial Checkout: No Device or User Reservations Available

When user U7 (who has no reservations) sends a capability request from device D2 (which also has no reservations), the 3 counts of F1 currently served to D2 are returned to the shared counts. (Four shared counts are now available.)

The following describes possible scenarios of what happens next:

- If the request has no desired features, no counts are served since neither D2 nor U7 has reserved counts.
- If the request marking F1 for partial-checkout asks for 4 counts of F1, the remaining 4 available shared counts are served.
- If the request marking F1 for partial-checkout asks for 5 counts of F1, the remaining 4 available shared counts are served, satisfying 4 of the 5 desired counts.

Scenario 2—Partial Checkout: Device Reservations But No User Reservations Available

When user U7 (who has no reservations) sends a capability request from device D1, the 4 counts of F1 currently served to D1 (3 reserved counts for D1 and 1 shared count) are returned. (Two shared counts are now available.)

The following describes possible scenarios of what happens next:

- If the request has no desired features, the 3 counts of F1 reserved for D1 are served.

- If the request marking F1 for partial-checkout asks for 4 counts of F1, the 3 counts reserved for D1 plus 1 shared count are served.
- If the request marking F1 for partial-checkout asks for 6 counts of F1, the 3 counts reserved for D1 and the remaining 2 available shared counts are served, satisfying 5 of the 6 desired counts.

Scenario 3—Partial Checkout: User Reservations But No Device Reservations Available

If user U1 sends a capability request from device D2 (which has no reservations), the 3 counts of F1 currently served to D2 are returned to shared counts. (Four shared counts are now available.)

The following describes possible scenarios of what happens next:

- If the request has no desired features, the 2 counts of F1 reserved for U1 are served.
- If the request marking F1 for partial-checkout asks for 4 counts of F1, the 2 counts reserved for U1 plus 2 shared counts are served.
- If the request marking F1 for partial-checkout asks for 7 counts of F1, the two counts reserved for U1 and the remaining 4 available shared counts are served, satisfying 6 of 7 desired counts.

Scenario 4—Partial Checkout: Device and User Reservations Available

If user U1 sends a capability request from device D1, the 4 counts of F1 currently served to D1 (3 reserved and 1 shared) are returned. (Two shared counts are now available.)

The following describes possible scenarios of what happens next:

- If the request has no desired features, all reserved features—3 counts of F1 for D1 and 2 counts for U1—are served.
- If the requests marking F1 for partial-checkout asks for 7 counts of F1, the 3 counts reserved for D1, the 2 counts reserved for U1, and the remaining 2 available shared counts are served.
- If the request marking F1 for partial-checkout asks for 8 counts of F1, the 3 counts reserved for D1, the 2 counts reserved for U1, and the remaining 2 available shared counts are served to satisfy 7 of the 8 desired counts.

Both Incremental and Partial-Checkout Options Enabled

The following scenarios assume that the capability request is flagged as incremental and feature F1 is enabled for partial checkout:

- Scenario 1—Incremental and Partial Checkout: No Device or User Reservations Available
- Scenario 2—Incremental and Partial Checkout: Device Reservations But No User Reservations Available
- Scenario 3—Incremental and Partial Checkout: User Reservations But No Device Reservations Available
- Scenario 4—Incremental and Partial Checkout: Device and User Reservations Available

Scenario 1—Incremental and Partial Checkout: No Device or User Reservations Available

When user U7 (who has no reservations) sends a incremental capability request from device D2 (which also has no reservations), the 3 counts of F1 currently served to D2 are returned to the shared counts. (Four shared counts are now available.)

The following describes possible scenarios of what happens next:

- If the incremental request has no desired features, 3 shared counts are served to preserve the previous 3 counts on the device.
- If the incremental request marking F1 for partial-checkout asks for 1 count of F1, 3 shared counts are served to preserve the previous count. Additionally, the remaining 1 available shared count is served as the 1 desired count.
- If the incremental request marking F1 for partial-checkout asks for 2 counts of F1, 3 shared counts are served to preserve the previous count. Additionally, the remaining 1 available shared count is served to satisfy 1 of the 2 desired counts.

Scenario 2—Incremental and Partial Checkout: Device Reservations But No User Reservations Available

When user U7 (who has no reservations) sends a capability request from device D1, the 4 counts of F1 currently served to D1 (3 reserved counts for D1 and 1 shared count) are returned. (Two shared counts are now available.)

The following describes possible scenarios of what happens next:

- If the incremental request has no desired features, the 3 counts of F1 reserved for D1 plus 1 shared count are served to preserve the previous 4 counts on the device.
- If the incremental request marking F1 for partial-checkout asks for 1 count of F1, the 3 counts reserved for D1 plus 1 shared count are served to preserve the previous count. Additionally, the remaining 1 available shared count is served as the 1 desired count.
- If the incremental request marking F1 for partial-checkout asks for 2 counts of F1, the 3 counts reserved for D1 plus 1 shared count are served to preserve the previous count. Additionally, the remaining 1 available shared count is served to satisfy 1 of the 2 desired counts.

Scenario 3—Incremental and Partial Checkout: User Reservations But No Device Reservations Available

When user U1 sends a capability request from device D2 (which has no reservations), the 3 counts of F1 currently served to D2 are returned to shared counts. (Four shared counts are now available.)

The following describes possible scenarios of what happens next:

- If the incremental request has no desired features, the 2 counts of F1 reserved for U1 plus 1 shared count are served to preserve the previous 3 counts on the device.
- If the incremental request marking F1 for partial-checkout asks for 3 counts of F1, the 2 counts of F1 reserved for U1 plus 1 shared count are served to preserve the previous count. Additionally, the remaining 3 available shared counts are served to satisfy the desired 3 counts.

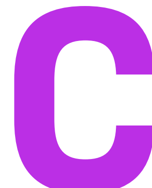
- If the incremental request marking F1 for partial-checkout asks for 4 counts of F1, the 2 counts of F1 reserved for U1 plus 1 shared count are served to preserve the previous count. Additionally, the remaining 3 available shared counts are served to satisfy the desired 3 of the desired 4 counts.

Scenario 4—Incremental and Partial Checkout: Device and User Reservations Available

If user U1 sends a capability request from device D1, the 4 counts of F1 currently served to D1 (3 reserved and 1 shared) are returned. (Two shared counts are now available.)

The following describes possible scenarios of what happens next:

- If the incremental request has no desired features, the 3 counts of F1 reserved for D1 plus 1 of the 2 reserved counts for U1 are served to preserve the previous 4 counts on the device.
- If the incremental request marking F1 for partial-checkout asks for 2 counts of F1, the 3 counts reserved for D1 and 1 of the 2 counts reserved for U1 are served to preserve the previous count. Additionally, the remaining 1 reserved count for U1 and the remaining 1 available shared count are served to satisfy the desired 2 counts.
- If the incremental request marking F1 for partial-checkout asks for 4 counts of F1, the 3 counts reserved for D1 and 1 of the 2 counts reserved for U1 are served to preserve the previous count. Additionally, the 1 remaining reserved count for U1 and the remaining 2 available shared counts are served to satisfy the desired 3 of the desired 4 counts.



SysV Alternative for Installation on Linux

As an alternative to installing the FlexNet Embedded local license server as a Linux service using systemd (described in Chapter 3, Installing and Running the License Server), you can use the SysV init runlevel system to perform the installation. This type of installation is an option only if your operating system supports SysV.

This following sections describe how to use SysV to install and run the license server as a Linux service:

- Files Required for License Server Installation Using SysV
- Prepare for Linux Installation
- Install and Start as a Linux Service
- Install and Start in Console Mode
- Edit Local Settings Post-Installation
- Uninstall the License Server

Files Required for License Server Installation Using SysV

The following files, distributed with the license server, support the installation and running of the license server service under SysV:

- `producer-settings.xml`
- `flexnetls`
- `configure`
- `local-configuration.yaml`

Note ▪ *The example `local-configuration.yaml` file included in the license server package is for installs under SysV. For systemd installs, the install script generates the `local-configuration.yaml` file.*

Residing in the same directory as `flexnetls.jar`, these files work together to generate the components required to install and start the license server service in a SysV configuration.

For a list of all files (aside from the SysV installer files) that can be distributed with the license server, see License Server Components in Chapter 2, Getting Started.

Prepare for Linux Installation

Use the following steps to configure license server in preparation for installation.

Task ***To prepare for installation***

1. If you have not already done so, extract the contents of the `.tar` file to the an installation directory.
2. Copy the `producer-settings.xml` file that you created into the `server` directory in the installation directory for the license server software package. This file defines operational policies for your license server. See License Server Configuration Utility in Chapter 7, Producer Tools, for details about this file.
3. To create and edit settings for your local environment, execute **`sudo ./configure`** from the installation directory to generate the `/etc/default/flexnetls-`*`producer_name`* file. Do either of the following:
   - To generate the file with the default settings, simply issue the **`sudo ./configure`** command.
   - To generate the file with settings specific to your environment, run the command with one or more of the options described in the following table to edit settings as needed. For example, to update the port and user name for the license server service, you might enter the following, where **`7071`** and **`user1`** are the new values to which you are updating the port and user name:

     ```
     sudo ./configure --port 7071 --user user1
     ```

     Note ▪ *Any setting value that uses a space must be enclosed in double quotations (for example, "user 1").*

Table C-1 ▪ Local Settings for the License Server on Linux

Option	Description
--program-dir *dir*	The installation location of the license server service (default is `/opt/flexnetls/`*`producer_name`*).
--data-dir *dir*	The location of trusted storage (default is `/var/opt/flexnetls/`*`producer_name`*). This overrides the `server.trustedStorageDir` value in the `producer-settings.xml`.

Table C-1 ▪ Local Settings for the License Server on Linux (cont.)

Option	Description
--port *port*	The listening port used by the license server service (default is `7070`). If the device on which the license server is running uses multiple network interfaces, you can use the `port` option to specify the interface that you want the license server to use. Simply include the IP address for the interface in square brackets, as shown in this example: `--port [127.0.0.1].1443`
--user *user_name*	The user name under which the service runs (default is `flexnetls`).
--group *group_name*	The group name under which the service runs (default is `flexnetls`).
--java_home *path* **or** **--jre_home *path***	The path for JDK or JRE installation that the license server should use. By default, the license server uses the value of your `JAVA_HOME` or `JRE_HOME` system environment variable, whichever is defined on your device, to determine the Java installation location. However, if you want the license server to use different Java installation on your system, provide the installation's explicit path for either the `java_home` or `jre_home` setting. (This override pertains to the license server only; your system continues to use the Java installation defined by the system environment variable.)

Install and Start as a Linux Service

Use these steps to install and start the license server as a service on your Linux device.

Task ***To install and start the license server as a service on Linux***

1. Execute `sudo ./flexnetls install` to install the license server as a service. If the server service is installed properly, you receive the message `Service flexnetls-producer_name installed`, where `flexnetls-`*`producer_name`* is the name of the service.
2. Go to the `/etc/init.d` directory, and execute `sudo service flexnetls-`*`producer_name`*` start` to start the service. You should receive a message stating that the service has started.

   Note ▪ *Before starting a license server that uses the VM UUID as its hostid, you might need to run a special service to enable the retrieval of the VM UUID. See Manage the License Server Service on Linux in Chapter 3, Installing and Running the License Server, for details.*

3. To confirm that the service is running, execute `sudo service flexnetls-`*`producer_name`*` status`. You should receive the message `flexnetls is running`. (To stop the service, execute `sudo service flexnetls-`*`producer_name`*` stop`.)

4. To view the license server log, navigate to the server's logging directory (by default, /var/opt/flexnetls/*producer_name*/logs), and review the contents of the appropriate .log file.

Note ▪ *Once you install the license server as service, you cannot use any of the "flexnetls" non-service command-line options, such as "console" or "install".*

Install and Start in Console Mode

Use these steps to install and start the license server in console mode on your Linux device.

Note ▪ *Run the license server in console mode for testing and development purposes only. It is recommended that the enterprise customer run the license server only as a service.*

Task ***To install and start the license server in console mode on Linux***

1. If you attempt to start the license server in console mode as a non-root user, you receive the message You must be a member of the flexnetls group to do this. Therefore, as a root user, run the following command to add your user name as a non-root user to the flexnetls group:

```
usermod -g flexnetls user
```

Note ▪ *Before starting a license server that uses the VM UUID as its hostid, you might need to run a special service to enable the retrieval of the VM UUID. See Manage the License Server Service on Linux in Chapter 3, Installing and Running the License Server, for details.*

2. Log in as the non-root user, navigate to the server directory, and run the command **./flexnetls**.

Edit Local Settings Post-Installation

As part of the installation process, the license server package generates a local settings file that configures the license server for your local environment, as described in the previous section, Prepare for Linux Installation. The installation process provides a facility that lets you change default settings before the settings file is generated.

However, if you need to modify the license server's local settings *after* you have installed the license server on a Linux platform, use these procedures:

- When Server Runs as a Service
- When Server Runs Console Mode

When Server Runs as a Service

Use this procedure to update local settings when the license server is running as a SysV service on Linux.

Task ***To modify local settings after installing the license server as a service on Linux***

1. Open the `/etc/default/flexnetls-`*`producer_name`* file in a text editor, and edit the settings as needed. (For example, you can uncomment settings or change existing values.) See the previous section Prepare for Linux Installation for a description of most settings.

   Note that the following settings are automatically added (as commented text) to the file at the time it is created with the `configure` tool:

Setting	Description
ACTIVE_HOSTID=*value/type*	The hostid to use for the license server. The hostid can only be set using a configuration file if no hostid has yet been specified for the license server. Once a hostid has been set, it can only be changed using the REST API `/hostids/selected` or using FlexNet License Server Administrator. The syntax is *`value/type`* (for example, `7200014f5df0/ETHERNET`). If no value is specified for this setting and no active hostid has yet been set for the license server, the license server uses, by default, the first available Ethernet address on the machine. If using a dongle ID or a hostid other than the first available Ethernet address, the license server administrator can specify it here. ***Important ▪*** *It is not recommended to change the hostid of a license server that has licenses mapped in the back office (FlexNet Operations). If the hostid is changed to a value that is different to that specified for the license server in the back office, any existing licenses mapped to the license server that is locked to the old hostid in the back office will be orphaned.* *To prevent this from happening, it is best practice to return the license server in the back office. During the return operation, you as the producer can transfer the licenses to a different device; this can be the same machine with the desired hostid. After the transfer, wait for the license server to synchronize with the back office (server synchronization occurs based on synchronization policies or on-demand).* *For more information on returning the license server, refer to the FlexNet Operations User Guide, section “Returning a Device”.* ***Important ▪*** *If you specify a server hostid using `ACTIVE_HOSTID`, the `server.hostType.order` property (if set) is ignored.*

Setting	Description
HTTPS_SERVER_FILE=*path*	The path to the HTTPS "server" configuration file used to support *incoming* HTTPS from client devices. For details on how the enterprise creates this configuration file and configures the server, see Incoming HTTPS in Chapter 6, Advanced License Server Features.
HTTPS_CLIENT_FILE=*path*	The path to the HTTPS "client" configuration file used to support *outgoing* HTTPS communication to FlexNet Operations. ***Note*** ▪ *The "client" configuration file is being deprecated and will be removed in a future release. For more information about the configuration file and HTTPS setup on the license server, see Outgoing HTTPS in Chapter 6, Advanced License Server Features.*
SERVER_ALIAS=*alias*	A user-defined name (sometimes called *host name*) for the license server. This name is added to server's capability requests to the back office, where it is then saved and used to identify the license server in the FlexNet Operations Producer and End User portals. One important use for this setting is that the alias can be included in the initial capability request sent at server registration, providing a helpful name by which users can identify the new server in the portals. If no alias is sent at registration, the server is identified by its hostid.
EXTENDED_SUFFIX=*suffix*	The suffix used for the extended hostid feature. See Extended Hostids in Chapter 6, Advanced License Server Features for details.
EXTRA_SYSPROPERTIES= *-property1=value1* [*-property2=value2*...]	One or more system properties (each in -D*key=value* format) that are passed to the Java Runtime system. The license server depends on the Java Runtime Environment to support certain network functionality, such as specifying the HTTP proxy. For example, if you plan to have the license server communicate with the back office through an HTTP proxy, use this setting to identify the proxy parameters needed to configure the server. (For details, see Proxy Support for Communication with the Back Office in Chapter 6, Advanced License Server Features.) The following shows example proxy parameters listed as -D system properties for this setting: `EXTRA_SYSPROPERTIES="-Dhttp.proxyHost=10.90.3.133 -Dhttp.proxyPort=3128 -Dhttp.proxyUser=user1a -Dhttp.proxyPassword=user1apwd35"`

Setting	Description
BACKUP_SERVER_HOSTID	The hostid of the back-up license server in a failover configuration with the current license server (as the main server), if the back-up server is "unknown"—that is, not registered—in FlexNet Operations. This setting adds the back-up server's hostid to the capability request sent by the current license server to the back office. The back office then automatically registers the back-up server in a failover configuration with the main server. This process saves the extra step of having to manually register the failover pair in FlexNet Operations. For more information, see Failover Using Synchronization to FlexNet Embedded in Chapter 5, More About Basic License Server Functionality. Make sure the back-up server's hostid is the same type (for example, "ETHERNET") as the hostid type of the current (main) license server. You might need to include this setting manually if it is not automatically generated in the settings file.

2. Execute `sudo service flexnetls-producer_name restart` to restart the license server service.

When Server Runs Console Mode

Use this procedure to update local settings when the license server is running in console mode on Linux.

Task ***To modify local settings after installing the license server in console mode on Linux***

1. Close the command window running the license server in console mode to stop the server.
2. Use command-line options with `flexnetls` to update settings. For example, to update the port setting, enter a command similar to this:

   ```
   ./flexnetls -port 7171
   ```

   For a description of the options, see the previous section Prepare for Linux Installation.
3. From the server directory, run `./flexnetls` to restart the server in console mode.

Uninstall the License Server

Use the appropriate procedure to uninstall the license server:

- Uninstall the License Server Service
- Uninstall the License Server Running in Console Mode

Uninstall the License Server Service

Use this procedure to uninstall the license server service.

Task ***To uninstall the license server service***

1. Execute the following command: `sudo service flexnetls-producer_name stop.`
2. Delete the `/opt/flexnetls/`*producer_name* folder.
3. Delete the `/etc/init.d/flexnetls-`*producer_name* and `/etc/default/flexnetls-`*producer_name* files.
4. Optionally, delete these files (listed here with their default locations):
    - The trusted storage files in `/var/opt/flexnetls/`*producer_name*
    - The log files in `/var/opt/flexnetls/`*producer_name*`/logs`

    Trusted storage and log locations are defined by the license server policies `server.trustedStorageDir` and `logging.directory`, respectively, the defaults for which are based on `${bases.dir}`. Depending on the values set for these policies on your server, your trusted storage and log files might be in locations different from those mentioned in this step. See Appendix A, Reference: Policy Settings for the License Server.

Uninstall the License Server Running in Console Mode

Use this procedure to uninstall the license server running in console mode.

Task ***To uninstall the license server running in console mode***

1. Close the command window running the license server.
2. Delete both the `/opt/flexnetls/`*producer_name* folder and these files:
    - `/etc/init.d/flexnetls-`*producer_name*
    - `/etc/default/flexnetls-`*producer_name*
3. Optionally, delete the following (listed here with their default locations):
    - The trusted storage files in `/var/opt/flexnetls/`*producer_name*
    - The log files in `/var/opt/flexnetls/`*producer_name*`/logs`

    Trusted storage and log locations are defined by the license server policies `server.trustedStorageDir` and `logging.directory`, respectively, the defaults for which are based on `${bases.dir}`. Depending on the values set for these policies on your server, your trusted storage and log files might be in locations different from those mentioned in this step. See Appendix A, Reference: Policy Settings for the License Server.

Manage the License Server Service: Command Summary

The following command options are available to manage a license server service installed using a SysV init run-level system. To obtain the *complete syntax usage* for a given command, refer to the following sections:

- Install and Start as a Linux Service
- Edit Local Settings Post-Installation

- Uninstall the License Server

Table C-2 ▪ Options Available for Managing the License Server Service

Option	Description
install	Installs the license server as a service.
start	Starts the license server service.
stop	Stops the license server service.
restart	Stops and restarts the license server service (for example, after you have made changes to local server settings in `/etc/default/flexnetls-`*`producer_name`*).
status	Shows the status of the license server service.
reset-database	Resets trusted storage on the license server. This option is rarely used except for testing purposes or deploying extended hostids. The enterprise license server administrator should not use this option unless you provide a directive to do so in specific situations.
restore-database	Restores trusted storage from a backup copy stored on the license server. See Trusted Storage Backup and Restoration. ***Important ▪*** *When the local license server is running as a service, the* *`-restore-database`* *command only restores the database if a custom installation location is defined. The command cannot be used to restore trusted storage from the backup at the service mode installation location.*

Model Definition Grammar for Named License Pools

The model definition defines named license pools and conditions for rules that allow or deny access to features.

The following sections provide information and examples for creating model definitions:

- Model Definition Grammar and Syntax—EBNF
- Use Case Examples and Their Model Definitions for Named License Pools
- Model Definition Examples for Reservations Converted to Named License Pools

Model Definition Grammar and Syntax—EBNF

The following section shows all possible grammar structures of the language used to write a model definition. The syntax representation is based on the Extended Backus–Naur Form (EBNF), which is a formal notation that can be used to describe other languages. It is grouped into the Parser and the Lexer. Both can be used with ANTLR (for more information, see antlr.org).

***Important ▪** Each definition can have only one model. Reserved model names are `reservations` and `default`.*

Parser

```
parser grammar model;
options { tokenVocab=modelLexer; }

// Parser
model: MODEL model_name LBRACE partitions? modelRule* RBRACE EOF;

model_name: QUOTEDSTRING ;
partitions: PARTITIONS LBRACE partition* RBRACE ;

partition: PARTITION partition_name LBRACE feature_spec* RBRACE ;
partition_name: QUOTEDSTRING ;
```

```
feature_spec: feature_name feature_version amount vendor_match?  ;
feature_name: QUOTEDSTRING ;
feature_version: major_version ( DOT minor_version )? ;
major_version: DIGIT+ ;
minor_version: DIGIT+ ;
amount: integer_amount | percentage_amount | REMAINDER ;
integer_amount: DIGIT+ ;
percentage_amount: DIGIT+ PERCENT ;

vendor_match: VENDOR STRING MATCHES match_string;
match_string: QUOTEDSTRING ;

modelRule: ON compound_matcher LBRACE rule_body RBRACE ;

compound_matcher: negatable_matcher compound_rhs* ;

compound_rhs: compound_and_rhs | compound_or_rhs ;

compound_and_rhs: (AND | C_AND) negatable_matcher ;
compound_or_rhs:  (OR  | C_OR) negatable_matcher ;

negatable_matcher: negated_matcher | simple_matcher ;

negated_matcher: (NOT | C_NOT) simple_matcher ;

simple_matcher: hostid_matcher | hostname_matcher | hosttype_matcher | vendor_dictionary_matcher |
any_matcher ;

hostid_matcher: HOSTID LPAREN parameter_list RPAREN ;
hostname_matcher: HOSTNAME  LPAREN parameter_list RPAREN ;
hosttype_matcher: HOSTTYPE LPAREN parameter_list RPAREN ;
vendor_dictionary_matcher: DICTIONARY LPAREN keyword_parameter_list RPAREN ;
any_matcher: ANY LPAREN RPAREN ;

parameter_list: parameter (COMMA parameter)* ;
parameter: QUOTEDSTRING ;
keyword_parameter_list: keyword_parameter (COMMA keyword_parameter)* ;

keyword_parameter: keyword_key COLON keyword_value ;
keyword_key: QUOTEDSTRING ;
keyword_value: WILDCARD | QUOTEDSTRING ;

rule_body: use_statement? server_specified? action? ;
use_statement: USE partition_name_list ;

server_specified: WITHOUT REQUESTED FEATURES LBRACE partition_server_specified |
feature_server_specified RBRACE ;

partition_server_specified: ALL FROM partition_name_list ;
feature_server_specified: feature_spec* ;

partition_name_list: partition_name (COMMA partition_name)* ;

action: ACCEPT | DENY | CONTINUE ;
```

Lexer

```
lexer grammar modelLexer;

MODEL : 'model';
ON : 'on';
PARTITIONS : 'partitions';
PARTITION : 'partition';
REMAINDER : 'remainder';
ANY : 'any';
HOSTID : 'hostid';
USE: 'use';
ACCEPT : 'accept';
DENY : 'deny';
CONTINUE : 'continue';
WITHOUT : 'without';
REQUESTED : 'requested';
FEATURES : 'features';
ALL : 'all';
FROM : 'from';
HOSTNAME: 'hostname';
HOSTTYPE: 'hosttype';
DICTIONARY: 'dictionary';
VENDOR: 'vendor';
STRING: 'string';
MATCHES: 'matches';

LPAREN : '(';
RPAREN : ')';
LBRACE : '{';
RBRACE : '}';
COMMA  : ',';
COLON  : ':';
WILDCARD : '*';
PERCENT : '%';
DOT     : '.';
AND :   'and';
OR : 'or';
NOT : 'not';
C_AND : '&&';
C_OR  : '||';
C_NOT :  '!';

DIGIT: '0'..'9' ;
QUOTEDSTRING: '"' (~('"' | '\\' | '\r' | '\n') | '\\' ('"' | '\\'))* '"';

// Whitespace, comments
WS           : [ \t\r\n\u000C]+ -> skip;
COMMENT      : '/*' .*? '*/' -> skip ;
LINE_COMMENT : '//' ~[\r\n]* -> skip ;

UNKNOWN_CHAR : . ;
```

Use Case Examples and Their Model Definitions for Named License Pools

The following examples illustrate possible use cases for sharing feature counts between license pools:

- Use Case: Simple Allow List
- Use Case: Simple Block List
- Use Case: Sharing Counts Between Business Units
- Use Case: Assigning Extra Counts To Business Unit
- Use Case: Exclusive Use of Feature Counts for Business Unit
- Use Case: Exclusive Use of Feature Counts for Business Unit With Exception of Specific Clients
- Use Case: Assigning Features Based on Combined hosttype and hostname Properties
- Use Case: Assigning Features Based on Vendor String Property
- Use Case: Device-specific Handling—Sharing Feature Counts Based On Hosttype
- Use Case: Named License Pool Receiving Entire Remaining Feature Count
- Use Case: Using Regular Expression to Allocate License Counts
- Use Case: Making Feature Counts Available to Multiple Business Units
- Use Case: Letting Server Specify Counts
- Use Case: Accumulating Counts from Multiple Named License Pools (“Continue” Action)

Background Information for Use Cases

The following is information to keep in mind when reading the information in this section.

Vendor Dictionary Data

Many of the following use cases use vendor dictionary data to define conditions for license allocation.

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 license server and vice-versa. Basically, the vendor dictionary provides a means to send information back and forth between the client and server for any producer-defined purposes, as needed; FlexNet Embedded does not interpret this data.

Vendor dictionary data is stored as key–value pairs. The key name is always a string, while a value can be a string or a 32-bit integer value, or an array of those types (arrays are only supported when using the Cloud Monetization REST API). Keys are unique in a dictionary and hence allow direct access to the value associated with them.

A common theme across the use cases in this appendix is that feature counts are shared between business units called Sales and Engineering. These business units would be identified by `"business-unit" : ["engineering", "sales" ]` entries in the vendor dictionary.

Communicate the available vendor dictionary data to your enterprise customer's license server administrator if you want to enable them to use this data for license allocation rules.

***Tip** ▪ For an example that demonstrates a vendor dictionary condition that uses an array, see Use Case: Making Feature Counts Available to Multiple Business Units.*

Using the Host Name and Host Type

The host type and host name are values that are optionally set by the software producer on a client device. They are also sometimes referred to as device type and device name, respectively. Both are a type of a human-readable "alias"—in contrast to the hostid—which can optionally be included in a capability request. Communicate the available host types and host names to your enterprise customer's license server administrator if you want to enable them to use the hosttype and hostname rules.

Default Behavior If No Conditions Are Met

When a feature request does not meet any of the conditions in the model definition, an `on any()` condition is expected that either denies or allows access to a license pool. If no such on `any()` condition is provided, the default is that the feature will be served from the default license pool (if features are available). This behavior is equivalent to the following code:

```
on any() {
    use "default"
    accept
}
```

The examples in this chapter generally do not explicitly show this final `on any()` condition.

Use Case: Simple Allow List

Requirement: Allow access to specific hostids

In this scenario, only declared hostids are allowed access. Note that no named license pools are needed in this scenario.

Instead of specifying all hostids in one condition within a rule of access, you can also include multiple rules with the "on hostid" condition in the model definition.

Model Definition Example

```
model "exampleModel" {
    on hostid("F01898AD8DD3/ETHERNET", "5E00A4F17201/ETHERNET") {
        use "default"
        accept
    }

    on any() {
        deny
    }
```

```
}
```

Note ▪ *The hostid is specified as a `value/type` pair (for example, `7200014f5df0/ETHERNET`). If a hostid condition does not specify the hostid type, it is assumed that the hostid is of type `string`.*

Use Case: Simple Block List

Requirement: Deny access to specific hostids

In this scenario, all users except a declared few are allowed access.

Model Definition Example

```
model "exampleModel" {
    on hostid("user1@example.com/USER", "user2@example.com/USER") {
        deny
    }

    on any() {
        use "default"
        accept
    }
}
```

Note ▪ *The hostid is specified as a `value/type` pair (for example, `7200014f5df0/ETHERNET`). If a hostid condition does not specify the hostid type, it is assumed that the hostid is of type `string`.*

Use Case: Sharing Counts Between Business Units

Requirement: Share feature counts between two business units.

This scenario features two business units called Sales and Engineering, which have been defined by entries in the vendor dictionary.

The model definition defines two named license pools, one for the Engineering business unit and another for the Sales business unit. The definition shares the total feature count of 10 for feature f1 equally between both business units, meaning each business unit gets 5.

Model Definition Example

```
model "exampleModel" {
    partitions {
        partition "engineering" {
            "f1" 1.0 5
        }
        partition "sales" {
            "f1" 1.0 5
        }
```

```
    }

    on dictionary("business-unit" : "engineering") {
        use "engineering"
        accept
    }

    on dictionary("business-unit" : "sales") {
        use "sales"
        accept
    }
}
```

Use Case: Assigning Extra Counts To Business Unit

Requirement: Make specified number of features accessible only to one particular business unit.

This scenario features the same business units as before—Sales and Engineering.

For this demonstration, let's assume that the total feature count for f1 is 10. Of the total count, 3 licenses are allocated exclusively to the named license pool called `engineering`. In addition, `engineering` is allowed access to the remaining 7 licenses (on a first-come-first-served basis) from the default license pool.

The Sales business unit (which has no pre-allocated counts in this scenario) only has access to the default license pool.

Model Definition Example

```
model "exampleModel" {
    partitions {
        partition "engineering" {
            "f1" 1.0 3
        }
    }

    on dictionary("business-unit" : "engineering") {
        use "engineering", "default"
        accept
    }
}
```

Use Case: Exclusive Use of Feature Counts for Business Unit

Requirement: Grant exclusive use of features to a specified business unit

This scenario uses the same business units and named license pools as the previous example: the default license pool, and a named license pool called `engineering` for the Engineering business unit. Other business units also exist (for example, the Sales business unit).

Let's assume that the total feature count for a feature called "cad" is 10. The entire count (expressed here as `100%`) is allocated to the named license pool `engineering`, for exclusive use by the Engineering business unit. This effectively denies access to feature "cad" for any capability request that comes from a different business unit (that is, any request that does not include the vendor dictionary entry `"business-unit" : "engineering"`).

Model Definition Example

```
model "exampleModel" {
    partitions {
        partition "engineering" {
            "cad" 1.0 100%                                // entire feature count
        }
    }

    on dictionary("business-unit" : "engineering") {
        use "engineering"
        accept
    }
}
```

Use Case: Exclusive Use of Feature Counts for Business Unit With Exception of Specific Clients

Requirement: Grant exclusive use of features to a specified business unit, with the exception of specified clients.

This scenario uses the same framework as the previous use case, Use Case: Exclusive Use of Feature Counts for Business Unit, but adds a filter that blocks specified clients.

Capability requests are granted for all clients from the business unit Engineering, except for requests coming from clients with the hostid `"5e00a4f17204/ETHERNET"` or `"5e00a4f17205/ETHERNET"`.

Model Definition Example

```
model "exampleModel" {
    partitions {
        partition "engineering" {
            "cad" 1.0 100%                                // entire feature count
        }
    }

    on dictionary("business-unit" : "engineering") and not hostid("5e00a4f17204/ETHERNET",
    "5e00a4f17205/ETHERNET") {
        use "engineering"
        accept
    }
}
```

Note ▪ *The hostid is specified as a `value/type` pair (for example, `7200014f5df0/ETHERNET`). If a hostid condition does not specify the hostid type, it is assumed that the hostid is of type `string`.*

For information about the operators AND and NOT used in this example, see AND, OR, and NOT Operators.

Use Case: Exclusive Use of Feature Counts for Business Unit and Specified Clients from other Business Units

Requirement: Grant exclusive use of features to a specified business unit and to a number of specified clients.

This scenario uses the same framework as the use case described in Use Case: Exclusive Use of Feature Counts for Business Unit With Exception of Specific Clients. However, instead of filtering out specified clients, it also grants access to a number of select clients that are not part of the Engineering business unit.

Capability requests are granted for all clients from the business unit Engineering, and for clients with the hostid "5e00a4f17204/ETHERNET" or "5e00a4f17205/ETHERNET" (which may well be part of other business units).

Model Definition Example

```
model "exampleModel" {
    partitions {
        partition "engineering" {
            "cad" 1.0 100%                              // entire feature count
        }
    }

    on dictionary("business-unit" : "engineering") or hostid("5e00a4f17204/ETHERNET", "5e00a4f17205/
    ETHERNET") {
        use "engineering"
        accept
    }
}
```

Note *▪ The hostid is specified as a `value/type` pair (for example, `7200014f5df0/ETHERNET`). If a hostid condition does not specify the hostid type, it is assumed that the hostid is of type `string`.*

For information about the operators AND and NOT used in this example, see AND, OR, and NOT Operators.

Use Case: Assigning Features Based on Combined hosttype and hostname Properties

Requirement: Allow only clients that match specified `hosttype` and `hostname` properties access to a named license pool.

This scenario uses the same Engineering business unit and named license pool that you already know from previous examples.

In this example, a producer wants to allow client devices that match both a particular host type and host name access to the named license pool `engineering`. This can be achieved by combining the `hosttype` and `hostname` conditions using the AND operator. Requests from devices that match only the host type or host name, or none of these, are served from the default license pool.

Model Definition Example

```
model "exampleModel" {
    partitions {
        partition "engineering" {
```

```
                "f1" 1.0 100
            }
        }

        on hosttype("tv") and hostname("device-1"){
            use "engineering"
            accept
        }
    }
```

Use Case: Assigning Features Based on Vendor String Property

Requirement: Allocate feature counts to named license pools based on the product name (specified through the vendor string property), and return counts to their original named license pool after use.

***Note** ▪ This use case illustrates how to allocate feature counts to named license pools based on the product name. However, the vendor string can be used to allocate counts on the basis of any of the substitution variables that FlexNet Operations supports. For more information on the vendor string, see the Tip, below.*

In this scenario, feature counts come from a product for which two different versions exist: an Evaluation version and a Permanent version. The Evaluation version includes three features (f1, f2, f3), and the Permanent version includes four features (f1, f2, f4, f5).

As a prerequisite, the product names `Eval` and `Permanent` must be defined in the vendor string `{EntitlementLineItem.productName}` for the respective product version.

***Tip** ▪ The producer defines the vendor string in the license model when they create a line item in the back office. The vendor string can hold arbitrary producer-defined license data. Data can range from feature selectors to pre-defined substitution variables. For more information, see Set Up the Selectors in FlexNet Operations, or Predefined Substitution Variables for License Strings in the FlexNet Operations User Guide.*

The model definition defines four named license pools. The **`vendor string matches`** directive is used to allocate counts—expressed in percent—from the Evaluation and Permanent product versions to different named license pools. The directive is evaluated when the license server loads the model definition.

Requests for feature counts are accepted or denied based on vendor dictionary data. In this scenario, the following vendor dictionary key-value pairs must be set up:

```
business-unit:engineering
business-unit:sales
PartNumber:Eval
PartNumber:Permanent
```

The model definition's rules of access only allow access to a particular named license pool if a capability request satisfies both conditions in the rule: it must match the dictionary entry for `PartNumber` and for `business-unit`. Capability requests that only satisfy one of the conditions or none are rejected, and access to the named license pool is blocked.

Model Definition Example

```
model "exampleModel" {
    partitions {
        partition "eval-engineering" {
            "f1" 1.0 75% vendor string matches "ProductName:Eval"
            "f2" 1.0 75% vendor string matches "ProductName:Eval"
            "f3" 1.0 75% vendor string matches "ProductName:Eval"
        }

        partition "permanent-engineering" {
            "f1" 1.0 75% vendor string matches "ProductName:Permanent"
            "f2" 1.0 75% vendor string matches "ProductName:Permanent"
            "f4" 1.0 75% vendor string matches "ProductName:Permanent"
            "f5" 1.0 75% vendor string matches "ProductName:Permanent"
        }

        partition "eval-sales" {
            "f1" 1.0 25% vendor string matches "ProductName:Eval"
            "f2" 1.0 25% vendor string matches "ProductName:Eval"
            "f3" 1.0 25% vendor string matches "ProductName:Eval"
        }

        partition "permanent-sales" {
            "f1" 1.0 25% vendor string matches "ProductName:Permanent"
            "f2" 1.0 25% vendor string matches "ProductName:Permanent"
            "f4" 1.0 25% vendor string matches "ProductName:Permanent"
            "f5" 1.0 25% vendor string matches "ProductName:Permanent"
        }
    }

    on dictionary("PartNumber":"Eval") and dictionary ("business-unit" : "engineering") {
        use "eval-engineering"
        accept
    }
    on dictionary("PartNumber":"Permanent") and dictionary ("business-unit" : "engineering") {
        use "permanent-engineering"
        accept
    }
    on dictionary("PartNumber":"Eval") and dictionary ("business-unit" : "sales") {
        use "eval-sales"
        accept
    }
    on dictionary("PartNumber":"Permanent") and dictionary ("business-unit" : "sales") {
        use "permanent-sales"
        accept
    }
    on any() {
        deny
    }
}
```

The following diagram illustrates the allocation of features to the four named license pools:

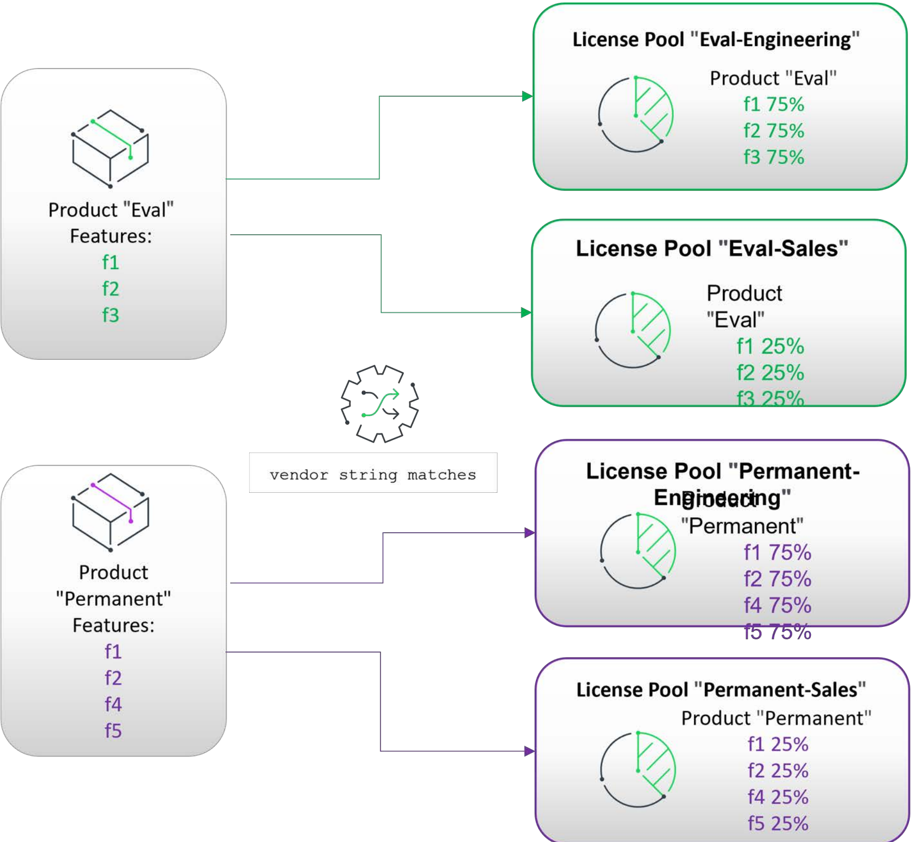


Figure D-1: “vendor string matches” Directive

Use Case: Device-specific Handling—Sharing Feature Counts Based On Hosttype

Requirement: Use the `hosttype` property to route the capability request to the appropriate named license pool.

In this scenario there are three different models of routers, which are distinguished through the `hosttype` property in the capability request. Counts of the “router” feature are split between three different named license pools—called `small`, `medium`, and `large`—and the “on hosttype” condition is used to route the request to the appropriate named license pool.

Model Definition Example

```
model "exampleModel" {
    partitions {
        partition "small" {
            "router" 1.0 5
        }
        partition "medium" {
            "router" 1.0 10
        }
        partition "large" {
            "router" 1.0 20
        }
```

```
    }

    on hosttype("small") {
        use "small"
        accept
    }
    on hosttype("medium") {
        use "medium"
        accept
    }
    on hosttype("large") {
        use "large"
        accept
    }
}
```

Use Case: Named License Pool Receiving Entire Remaining Feature Count

Requirement: Distribute all counts of a feature between three named license pools as equally as possible, with none in the default named license pool.

If you declare three named license pools, two with a count specifier of 33% and one with 34%, integer rounding might cause some counts to slip into the default license pool. You can avoid this by replacing the last count specifier with the `remainder` keyword, which will match all available remaining counts.

Note that any license pool lower than the one specifying the `remainder` keyword in the model definition receives zero counts for that feature.

Model Definition Example

```
model "exampleModel" {
    partitions {
        partition "p1" {                  // Total feature count of f1: 10
            "f1" 1.0 33%                  // Feature count in p1: 3
        }
        partition "p2" {
            "f2" 1.0 33%                  // Feature count in p2: 3
        }
        partition "p3" {
            "f3" 1.0 remainder            // Feature count in p3: 4
        }
    }
```

Use Case: Using Regular Expression to Allocate License Counts

Requirement: Allow clients that match a specific hosttype pattern access to a specified named license pool. Allow clients from business units whose name matches a specific pattern access to two specified named license pools.

As before, this scenario features two business units called Sales and Engineering, which have been defined by entries in the vendor dictionary. Using a regular expression in each condition means that the rule can be simplified—instead of creating a separate condition for every valid hostname or vendor dictionary string, a regular expression defines the pattern that a hostname and string must match to gain access to feature counts.

Model Definition Example

```
model "exampleModel" {
    partitions {
        partition "engineering" {
            "f1" 1.0 50%
        }
        partition "sales" {
            "f1" 1.0 50%
        }

    on hostname("~/[A-Z]{2}[0-9]{3}/") {
        use "engineering"
        accept
    }

    on dictionary("business-unit":"~/Eng[0-9]+/") {
        use "engineering", "sales"
        accept
    }

    on any() {
        deny
    }
```

Interpreting the Regular Expression

- Requests from clients with a hostname matching the following pattern get access to the `engineering` license pool:

    `[A-Z]` any uppercase letter
    `{2}` matched exactly twice
    `[0-9]` any number
    `{3}` matched exactly three times

    Examples for valid hostnames: `AB123`, `CD789`

- Requests from the `engineering` business unit (as defined in the vendor dictionary) matching the following pattern get access to the `engineering` and `sales` license pools:

    `Eng` must start with “Eng” (matching capitalization)
    `[0-9]` followed by any number

    Examples for valid vendor dictionary entries: `Eng123`, `Eng456789`

Use Case: Making Feature Counts Available to Multiple Business Units

Requirement: Make feature count from multiple named license pools available to users that belong to more than one business unit.

Important ▪ *This use case is only applicable to scenarios where the vendor dictionary value can be an array. Arrays are only supported if the capability exchange is handled via the REST interface of the Cloud Monetization API (CMAPI).*

Previous use cases showed how the model definition could allocate feature counts based on a business unit named in the vendor dictionary. However, these only catered for scenarios where a user belongs to a single business unit.

This use case demonstrates a simple way of allocating feature counts to multiple business units—called DevOps and Engineering in this example—that are specified in an array in the vendor dictionary.

The vendor dictionary entry would look similar to this:

```
"vendorDictionary": {
    "business-unit": [
        "Engineering",
        "DevOps",
        "Security"
        ],
    "Region": "EMEA"
}
```

Consider the following capability request that contains an array of strings associated with the `business-unit` key in the vendor dictionary:

```
{
    "features": [
        {
            "count": 20,
            "name": "cad",
            "version": "1.0"
        }
      ],
      "hostId": {
            "type": "string",
            "value": "h1"
    },
    "vendorDictionary": {"Region":"EMEA", "business-unit" : ["Engineering","DevOps"]},
    "borrow-interval": "300"
}
```

The vendor dictionary will match any of these clauses:

- `on dictionary("business-unit", "Engineering")`
- `on dictionary("business-unit", "DevOps")`
- `on dictionary("Region", "EMEA")`

If the vendor dictionary element contains a simple value (string, 32-bit integer) an equality match is done.

If the vendor dictionary element contains an array, a "contains" match is done, meaning that it is sufficient if only one matching element exists in the array.

The conditions in the following model would grant such a capability request—coming from a client device that belongs to multiple business units—access to features in both named license pools, `EngDevOps` and `EngSecurity`.

Model Definition Example

```
model "exampleModel" {
    partitions {
        partition "EngDevOps" {
            "cad" 1.0 30%
        }
        partition "EngSecurity" {
            "cad" 1.0 30%
        }
    }

    on dictionary("business-unit":"Engineering") and dictionary("Region":"EMEA"){
        use "EngDevOps"
        accept
    }

    on dictionary("business-unit":"Security") and dictionary("Region":"EMEA"){
        use "EngSecurity"
        accept
    }

    on any() {
        use "default"
        accept
    }
}
```

Use Case: Letting Server Specify Counts

Requirement: Let the server specify the desired counts, instead of the client

The following model definition demonstrates how to let the server specify the features that should be assigned to a client device. The model uses the `without requested features` directive in combination with a feature specification. Note that the server cannot override a capability request containing desired features coming from the client.

The following rule of access assigns a count of feature f1 to each capability request with the hostname “myhost”.

Model Definition Example

```
model "exampleModel" {
    partitions {
        partition "p1" {
            "f1" 1.0 10
        }
    }

    on hostname("myhost") {
```

```
        use "p1", "default"
        without requested features {
            "f1" 1.0 1
        }
        accept
    }
}
```

You can also combine the `without requested features` directive with `all from "license_pool_name"`. This variation will result in the first capability request receiving all counts from the specified named license pool. For a scenario where `all from "license_pool_name"` might be used, see Scenario: Server-specified Counts.

***Note** ▪ Server-specified desired feature counts should be marked with the "partial" attribute, indicating that partial checkout is allowed if the feature's count falls short of the desired count.*

Use Case: Accumulating Counts from Multiple Named License Pools ("Continue" Action)

Requirement: Allow a capability request to collect feature counts from more than one license pool

This use case defines two named license pools for feature counts. If a capability request comes from Europe (containing the key-value pair `"region"/"Europe"`, defined by the producer using vendor dictionary data), it gets access to named license pool p1. If the request comes from a device labeled "10A" (containing the key-value pair `"model" : "10A"`, defined by the producer using vendor dictionary data), it gets access to named license pool p2. If both conditions are true, the request gets access to both p1 and p2.

Model Definition Example

```
model "exampleModel" {
    partitions {
        partition "p1" {
            "f1" 1.0 5
        }
        partition "p2" {
            "f1" 1.0 5
        }
    }

    on dictionary("region" : "Europe") {
        use "p1"
        continue
    }

    on dictionary("model" : "10A") {
        use "p2"
        continue
    }
}
```

Model Definition Examples for Reservations Converted to Named License Pools

The following examples show model definitions that might result from manually converting reservations to named license pools. Conversion involves creating and uploading a model definition that reproduces the allocation achieved previously through reservations.

Important ▪ *You must delete any existing reservation groups before uploading the model definition.*

Scenario: Counts Reserved By Hostid

The original reservation model consists of per-hostid reserved counts, together with the ability to get counts from the default shared pool. Reservations might look like this when expressed in a model definition:

```
model "model-1" {
    partitions {
        partition "reservation-1" {
            "f" 1.0 1
        }
    }

    on hostid("F01898AD8DD3/ETHERNET") {
        use "reservation-1", "default"
        accept
    }

    on any() {
        use "default"
        accept
    }
}
```

Note that creating rules of access based on hostids is possible, but generally not recommended. Maintaining large lists of rules of access with the hostid condition is tedious and prone to error. Instead, consider basing the rules of access on hostname or hosttype properties or vendor dictionary key/value pairs.

Note ▪ *The hostid is specified as a `value/type` pair (for example, `7200014f5df0/ETHERNET`). If a hostid condition does not specify the hostid type, it is assumed that the hostid is of type `string`.*

Scenario: Server-specified Counts

Sometimes, a capability request does not contain any desired features. In this case, the server can be configured to specify the features that should be assigned to a client device. If a reservation exists for the client that sent the capability request, the reserved counts are acquired. The following model reservation example demonstrates how this behavior is replicated with named license pools. The `without requested features` directive replicates the server-specified count, and `all from "license_pool_name"` allocates all feature counts from the specified named license pool to the first client sending a capability request that fulfills the conditions defined in the rule of access.

The `all from "license_pool_name"` instruction essentially acts as a shortcut to the feature counts specified for the relevant named license pool in the model definition. If the specified named license pool does not have sufficient counts to satisfy the request, counts are allocated from subsequent named license pools listed in the rules of access (provided that the client has access to those named license pools).

Let's assume that the original reservation model consists of per-hostid reserved counts, together with the ability to get counts from the default shared pool. An equivalent model definition and rule of access might look like this:

```
model "reservations" {
    partitions {
        partition "reservation-1" {
            "f1" 1.0 1
        }
    }

    on hostid("F01898AD8DD3/ETHERNET") {
        use "reservation-1", "default"
        without requested features {
            all from "reservation-1"
        }
        accept
    }
}
```

The `without requested features` directive can also be combined with a feature specification to produce more fine-grained rules of access. For a use case example, see Use Case: Letting Server Specify Counts.

As noted in the previous scenario, creating rules of access based on hostids is generally not recommended. Instead, consider basing the rules of access on hostname or hosttype properties or vendor dictionary key/value pairs.

Note *▪ Server-specified desired feature counts should be marked with the "partial" attribute, indicating that partial checkout is allowed if the feature's count falls short of the desired count.*

Note *▪ The hostid is specified as a `value/type` pair (for example, `7200014f5df0/ETHERNET`). If a hostid condition does not specify the hostid type, it is assumed that the hostid is of type `string`.*

Logging Functionality on the Local License Server

This appendix describes the logging functionality available for the local license server, specifically:

- Logging Style
- Custom Log Configurations
- Integration of License Server Logging With External Systems

Logging Style

A logging style configuration parameter can be used to configure rollover, JSON formatting and timestamp behavior. The different logging styles can be specified in the `local-configuration.yaml` file, using the `loggingStyle` property.

Set the property using the following format (the space after the colon is mandatory):

```
loggingStyle: value
```

The property can take one of the following values:

- **DAILY_ROLLOVER**—The log file is closed at midnight local time, compressed with gzip, and a new log file is started. Timestamp values use the local time zone. This is the default if no logging style is specified.
- **DAILY_ROLLOVER_UTC**—Same as `DAILY_ROLLOVER`, but with UTC timestamp.
- **CONTINUOUS**—Legacy value. Rollover is handled as `DAILY_ROLLOVER`.
- **CONTINUOUS_UTC**—Legacy value. Rollover is handled as `DAILY_ROLLOVER_UTC`.
- **JSON_ROLLOVER**—Logs are formatted using JSON. The log file is closed at midnight local time, compressed with gzip (suitable for Filebeat), and a new log file is started. Always uses ISO 8601 UTC timestamps.
- **JSON**—Logs are emitted as JSON to stdout only (suitable for Docker). Always uses ISO 8601 UTC timestamps.

Note ▪ The Docker-friendly JSON logging style is not supported when the local license server is run as a Windows or Linux service (because it only writes to stdout). If set, the style will be changed to DAILY_ROLLOVER.

Custom Log Configurations

If the preconfigured logging styles do not meet your requirements, you can customize the logging configuration. To do so, prepare an external configuration file and specify the path to that file in an environment variable ($LOGGING_CONFIGURATION). You can also specify the path in the `producer-settings.xml` file, using the `server.logConfiguration` property.

Documentation for logback appenders can be found here: https://logback.qos.ch/manual/appenders.html

To construct the external configuration file, extract a logback configuration from the `flexnetls.jar` file and modify it to suit your requirements. The following tables shows the existing configurations.

Table E-1 ▪

Logging Style	Extraction Command
CONTINUOUS	`unzip -p flexnetls.jar BOOT-INF/classes/logback-continuous.xml`
CONTINUOUS_UTC	`unzip -p flexnetls.jar BOOT-INF/classes/logback-continuous.xml`
DAILY_ROLLOVER	`unzip -p flexnetls.jar BOOT-INF/classes/logback-rollover.xml`
DAILY_ROLLOVER_UTC	`unzip -p flexnetls.jar BOOT-INF/classes/logback-rollover.xml`
JSON	`unzip -p flexnetls.jar BOOT-INF/classes/logback.xml`
JSON_ROLLOVER	`unzip -p flexnetls.jar BOOT-INF/classes/logback-rollover-json.xml`

Note ▪ The "_UTC" variants share a configuration file with their local time counterpart, as the difference is expressed in a "logging.timestamp" system property.

Integration of License Server Logging With External Systems

The local license server can pass log entries to external log monitoring software. Some common configurations are documented here, but others are possible by adapting these instructions.

Common configurations described in this section are:

- Graylog

- Elastic Stack
- logz.io

Examples for implementing some of these configurations are provided in the Example Configurations section.

Graylog

The local license server supports Graylog's GELF protocol (see graylog.org/features/gelf). GELF mode is enabled by supplying the following configuration values in `producer-settings.xml`:

- **graylog.host**—The host name of the Graylog server to which logging messages are sent.

  The format for `graylog.host` is: `[protocol:]host[:port]`

  The `protocol` and `port` components are optional, and default to "udp" and 12201, respectively. The supported protocol values are "udp" and "tcp".

- **graylog.threshold**—The lowest level of log-message granularity to record—`FATAL`, `ERROR`, `WARN`, `INFO`, `LICENSING`, `POLICY`, or `DEBUG`. For example, if **`FATAL`** is set, only messages about fatal events are recorded. However, if **`WARN`** is set, fatal-event, error, and warning messages are recorded. (Default is `WARN`.)

Instead of using `producer-settings.xml`, these values can also be supplied using the FlexNet License Server Administrator—see the *FlexNet Embedded License Server Administration Guide*, chapter “Using the FlexNet License Server Administrator Command-line Tool”.

The section Graylog Logging shows an example of how you can implement this configuration.

Elastic Stack

To use Elastic Stack logging, you must configure the server for JSON logging. To do so, in the `local-configuration.yaml` file, specify the following property and value:

```
loggingStyle: JSON_ROLLOVER
```

The Elastic Stack consists of 3 components:

- **Logstash**. This component receives log entries, filters and transforms them and then passes the entries onwards.
- **Elasticsearch**. This component stores the log entries.
- **Kibana**. This component is the GUI where log entries can be searched and visualized.

The following methods are available to pass the log entries to Logstash:

- **Using Logstash’s native support of the GELF protocol**—Set up the `graylog.host` and `graylog.threshold` configuration properties to refer to Logstash.
- **Using Elastic Stack's Filebeat to monitor the server’s logging directory**—The use of Filebeat to send log entries to the Elastic stack is generally preferred to the use of GELF, because the JSON output is richer and more flexible than the GELF log entry encapsulation, and it is more robust when network interruptions happen. For more information about Filebeat, go to elastic.co/beats/filebeat.

The sections Elastic Stack and Filebeat and Elastic Stack and GELF show examples of how you can implement this configuration.

logz.io

logz.io is a managed and customized Elastic Stack logging offering. Refer to the methods in the previous section, Elastic Stack, for a short explanation of how to configure the server to work with it. For more information about logz.io, go to logz.io.

Example Configurations

This section contains examples for external logging configurations.

- Graylog Logging
- Elastic Stack and Filebeat
- Elastic Stack and GELF

Graylog Logging

The following example demonstrates Graylog logging.

Task ***To enable Graylog logging***

1. Start Graylog. You could run Graylog using Docker or run it as a a virtual machine appliance under VirtualBox, depending on your preference.
2. Set `graylog.host` to `localhost` to specify the Graylog server and set your desired log threshold (for example, `INFO` or `LICENSING`). If using the FlexNet License Server Administrator, you would use a command such as the following:

   ```
   flexnetlsadmin -authorize admin -passwordConsoleInput -config -set graylog.host=localhost
       graylog.threshold=LICENSING
   ```

   Instead of using the FlexNet License Server Administrator you could also create a new `producer-settings.xml` file.
3. Restart the license server. Log information will now be sent to Graylog on port 12201 using UDP (the default port and protocol).
4. Configure Graylog to receive the log information. For information, consult the Graylog documentation.

Elastic Stack and Filebeat

This section explains how to log to a Docker installation of the Elastic Stack (Elasticsearch, Logstash and Kibana), using Filebeat to send log contents to the stack. Log data is persisted in a Docker volume called "monitoring-data".

This configuration enables you to experiment with local license server logging to the Elastic Stack. However, it is not recommended for production use, due to insufficient high availability, backup and indexing functionality.

This section is split into the following steps:

1. Preparing the Directory Structure
2. Preparing the Local License Server
3. Creating the “docker-compose” File
4. Creating the Elasticsearch Files
5. Adding a Logstash Configuration
6. Building the Elastic Stack
7. Preparing to Use Filebeat
8. Sending Log Entries to Logstash

Preparing the Directory Structure

This demo uses the following directory structure:

Directory structure

```
demo/
  |
  ----- docker-compose.yml
  |
  ----- elasticsearch/
  |      |
  |      ---- Dockerfile
  |      |
  |      ---- elasticsearch.yml
  |
  ----- filebeat/
  |
  ----- server/
  |      |
  |      --- flexnetls.jar
  |      |
  |      --- producer-settings.xml
  |      |
  |      --- local-configuration.yaml
  |
  ----- logstash.conf
```

Preparing the Local License Server

Follow the steps below to prepare the server to log to a Docker installation. This step assumes that Docker is installed.

Task ***To prepare the local license server for logging to a Docker installation:***

1. Copy the local license server files—`flexnetls.jar`, `producer-settings.xml`, and `local-configuration.yaml`—into the `server` directory.
2. Configure the server for logging in JSON format. Add the following entry to the top section of `local-configuration.yaml`:

    ```
    loggingStyle: JSON_ROLLOVER
    ```

    This causes logs to be written in JSON format to the default location of `$HOME/flexnetls/$PUBLISHER/logs`.
3. Start the license server using the following command from the `server` directory:

    ```
    $ java -jar flexnetls.jar
    ```
4. Confirm that log files with a .json extension are being created.

Creating the “docker-compose” File

Use the sample below to create the `docker-compose.yml` file. Note that this demo uses the producer name “acme”.

docker-compose.yml

```
version: '2.2'
services:
  elasticsearch:
    build:
      context: elasticsearch/
    container_name: elasticsearch
    volumes:
      - monitoring-data:/usr/share/elasticsearch/data
    ports:
      - "9200:9200"
      - "9300:9300"
    environment:
      ES_JAVA_OPTS: "-Xmx512m -Xms512m"
  logstash:
    image: docker.elastic.co/logstash/logstash:7.9.1
    container_name: logstash
    ports:
      - "5000:5000"
      - "5044:5044"
      - "12201:12201/udp"
    expose:
      - "5044/tcp"
      - "12201/udp"
    logging:
      driver: "json-file"
    environment:
      LS_JAVA_OPTS: "-Xmx256m -Xms256m"
    volumes:
      - ./logstash.conf:/usr/share/logstash/pipeline/logstash.conf
    depends_on:
      - elasticsearch
  kibana:
    image: docker.elastic.co/kibana/kibana:7.9.1
    ports:
      - "5601:5601"
    depends_on:
      - elasticsearch
volumes:
  monitoring-data:
    driver: local
```

Creating the Elasticsearch Files

Create the following files in the elasticsearch directory.

elasticsearch/Dockerfile

```
FROM elasticsearch:7.9.1

COPY ./elasticsearch.yml /usr/share/elasticsearch/config/elasticsearch.yml

# FileRealm user account, useful for startup polling.
RUN bin/elasticsearch-users useradd -r superuser -p esuser admin

RUN yum -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm \
    && yum update -y \
    && yum install -y jq \
    && yum upgrade
```

elasticsearch/elasticsearch.yml

```
cluster.name: "docker-cluster"
network.host: 0.0.0.0

# minimum_master_nodes need to be explicitly set when bound on a public IP
# set to 1 to allow single node clusters
# Details: https://github.com/elastic/elasticsearch/pull/17288
discovery.zen.minimum_master_nodes: 1

## Use single node discovery in order to disable production mode and avoid bootstrap checks
## see https://www.elastic.co/guide/en/elasticsearch/reference/current/bootstrap-checks.html
discovery.type: single-node

node.data : true
discovery.seed_hosts : []
```

Adding a Logstash Configuration

Add a Logstash configuration such as the following to the `demo` directory:

logstash.conf

```
input {
  beats {
    port => 5044
  }
}

output {
    stdout { codec => rubydebug }

    if ( "lls-logs" in [tags] ) {
        elasticsearch {
            hosts => ["elasticsearch:9200"]
            id => "lls-logs"
            index => "lls-logs-%{+YYYY.MM.dd}"
            codec => "json"
        }
    }
}
```

Building the Elastic Stack

You can now build the Elastic Stack by executing this command in the `demo` directory:

```
$ docker-compose build
```

Preparing to Use Filebeat

Download and expand Filebeat into the `filebeat` directory. This tool will send JSON log entries to the Elastic Stack. You can obtain a copy from www.elastic.co/downloads/beats/filebeat. On Linux you can also use your package manager (DEB, RPM) to install Filebeat. Using the package manager will install it as a system service with systemd bindings. In this case the configuration can be found in `/etc/filebeat/`.

The Filebeat distribution contains an example `filebeat.yml` file. Replace it with this:

filebeat/filebeat.yml

```
filebeat.registry.path: ${HOME}/.filebeat-registry

filebeat.config:
  modules:
    path: ${path.config}/modules.d/*.yml
    reload.enabled: false

filebeat.inputs:
- type: log
  json.keys_under_root: true
  json.overwrite_keys: true
  json.add_error_key: true
  encoding: utf-8
  tags: ["lls-logs"]
  index : "%{[agent.name]}-lls-%{+yyyy.MM.dd}"
  paths:
    - ${HOME}/flexnetls/acme/logs/*.json

output.logstash:
  hosts: ["localhost:5044"]
```

Note *▪ This sample uses the producer name “acme”. In your file, replace “acme” with your producer name, as found in* `producer-settings.xml`*.*

Sending Log Entries to Logstash

Perform the steps below to log entries to Logstash. You can then view the entries in Kibana.

Task ***To send log entries to Logstash***

1. Bring up the Elastic Stack by executing this command from the demo directory:

    ```
    $ docker-compose up -d
    ```

2. Start the license server (if it is not already running).
3. Start Filebeat to start sending log entries to Logstash:

    ```
    $ ./filebeat -e
    ```

4. Open Kibana with a browser by going to http://localhost:5601. In Kibana's home page, click **Connect to your Elasticsearch index**.
5. Create an index pattern of 'lls-logs-*'. When asked, set '@timestamp' as the primary time field. For information on index patterns, see www.elastic.co/guide/en/kibana/current/tutorial-define-index.html.
6. Click **Discover** (in the grid menu); this should display license server log entries. Consult the Kibana documentation for information about searching the log entries.

Elastic Stack and GELF

The docker-compose configuration is the same as in section Creating the "docker-compose" File, but with a change to the Logstash part (note that the log `entrypoint` key would appear on a single line):

Elastic Stack docker-compose example 2

```
version: '2.2'
services:
  elasticsearch:
    build:
      context: elasticsearch/
    container_name: elasticsearch
    volumes:
      - monitoring-data:/usr/share/elasticsearch/data
    ports:
      - "9200:9200"
      - "9300:9300"
    environment:
      ES_JAVA_OPTS: "-Xmx512m -Xms512m"
  logstash:
    image: docker.elastic.co/logstash/logstash:7.9.1
    container_name: logstash
    ports:
      - "5000:5000"
      - "5044:5044"
      - "12201:12201/udp"
    expose:
      - "5044/tcp"
      - "12201/udp"
    logging:
      driver: "json-file"
    environment:
      LS_JAVA_OPTS: "-Xmx256m -Xms256m"
    entrypoint: logstash -e 'input { gelf { } } output { elasticsearch { id => "lls" hosts => ["http:/
/elasticsearch:9200"] } }'
    depends_on:
      - elasticsearch
  kibana:
    image: docker.elastic.co/kibana/kibana:7.9.1
    ports:
      - "5601:5601"
    depends_on:
      - elasticsearch
volumes:
  monitoring-data:
    driver: local
```

The Filebeat agent is not required in this scenario.

Consuming Streaming Content

The /clients.stream and /features.stream APIs should be used if you need to query large tables. They asynchronously return client and feature data as individual newline-delimited JSON (NDJSON) objects. The content type is "application/x-ndjson", not "application/json". For information about the NDJSON standard, see https://github.com/ndjson/ndjson-spec.

Note that the /clients.stream and /features.stream APIs are available only for the local license server.

The streaming content returned by the /clients.stream and /features.stream APIs should be consumed one line at a time as it arrives. The client that reads the stream should read without pausing, to avoid a disconnect if the connection is determined to be idle.

The following Java example code demonstrates how to use a Jackson ObjectMapper to deserialize each line to a Map<String, Object> representation:

Java Example

```
final static String CLIENT_URL = "http://localhost:7070/api/1.0/instances/~/clients.stream";
// localhost LLS

final URL url = new URL(CLIENT_URL);

HttpURLConnection con = (HttpURLConnection) url.openConnection( );
con.setRequestMethod("GET");
con.addRequestProperty("Accept", "application/x-ndjson");
con.setDoInput(true);
con.setConnectTimeout(30*1000);
con.setReadTimeout(0);

int responseCode = con.getResponseCode();
if ( responseCode == HTTP_OK) {
    InputStream is = con.getInputStream();

    ObjectMapper objectMapper = new ObjectMapper()
```

```
    int     numRecords = 0;
    try (BufferedReader reader = new BufferedReader(new InputStreamReader(is))) {
        while (true) {
            String line = reader.readLine();
            if ( line == null ) {
                // EOF
                break;
            }

            // Convert line from JSON.
            Map<String,Object> value = objectMapper.readValue(line, Map.class)
            System.out.println(value);
            numRecords++;
        }
    }

    System.out.println(numRecords + " client records transferred");
} else {
    System.out.println("Server error: " + responseCode);
}
```

Some HTTP client libraries have direct support for streaming data (for instance, Retrofit's @Streaming annotation).

Workflow Example for Producer-Defined Binding

This appendix provides step-by-step instructions to help you set up binding on a local license server. Using the example implementation that is included in the license server package, the appendix also simulates a binding break and explains how to repair the break.

The following diagram shows an overview of the workflow when binding is broken and subsequently repaired.

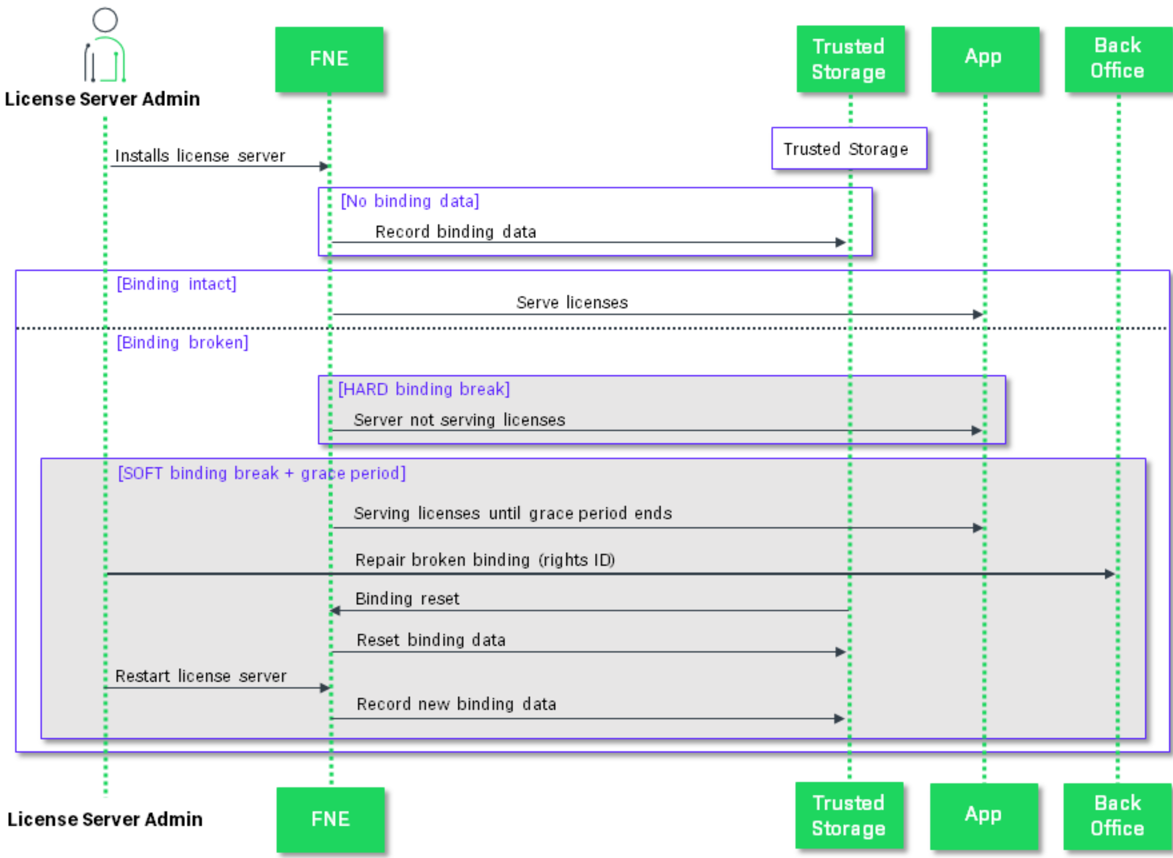


Figure G-1: Workflow when producer-defined binding is in place

This includes the following sections:

1. Adding Binding
2. Simulating a Binding Break
3. Repairing a Binding Break

For general information about binding, see Binding-Break Detection with Grace Period.

Adding Binding

To enable producer-defined binding on the local license server, you need to add the policy setting `binding.break.policy` to the `producer-settings.xml` file. This policy defines the action taken when the binding-break detection feature perceives a break. Specify one of the following values:

- **hard**—An immediate hard binding break is imposed—that is, the license server can no longer serve licenses.
- **soft**—A soft binding break is in effect, allowing the license server to continue to serve licenses despite the break.
- **interval**—A soft binding break with a grace period is in effect, allowing the license server to continue to serve licenses until the grace period expires. When the expiration occurs, a hard binding break goes into immediate effect, and licenses can no longer be served.

  The length of the grace period is the *interval* value, specified with an optional unit-suffix letter—`s`, `m`, `h`, `d`, or `w`—indicating seconds, minutes, hours, days, or weeks (for example, **3d** is 3 days). If no suffix is used, the server assumes the value is in seconds.

Workflow for Adding Binding

The following diagram shows the workflow for activating binding on a local license server.

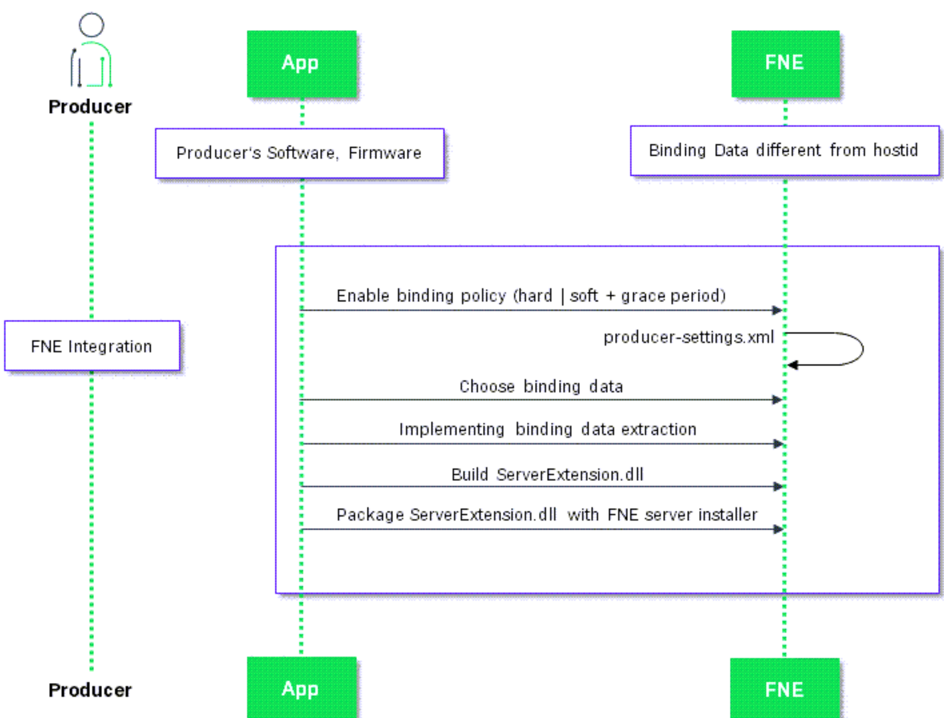


Figure G-2: Adding binding on a local license server

Specifying the Policy Setting in the Configuration File

Let's assume you want to implement a hard binding break. Create a separate "properties" text file with the setting `binding.break.policy=`**hard**. For more information about creating a properties file, see Generating the Policy Settings.

Then run a command similar to the following, where `properties.txt` is the properties file.

```
flexnetlsconfig -prop properties.txt -o producer-settings.xml -id IdentityClientServer.bin
```

This generates a `producer-settings.xml` file that can be used to start up the local license server. However, note that an error occurs during startup of the license server, as shown in the following example log:

```
20:15:06,582 INFO  - Starting FlexNet License Server 2022.05.0 (build 266500)
20:15:06,585 INFO  - Copyright (C) 2013-2022 Flexera Software LLC.
20:15:06,585 INFO  - All Rights Reserved.
20:15:06,585 INFO  - Running as a service
20:15:07,681 INFO  - Host Configuration [os=Linux, mem=2.0GB, mem_free=1.6GB, processors=1,
nonheap_committed=34.6MB, nonheap_init=2.4MB, nonheap_used=33.6MB]
20:15:49,654 INFO  - Server hostids acquired: [00155D900105/Ethernet]
20:15:49,689 ERROR - Unable to load the library:/opt/flexnetls/fnedemo/fnedemo/
libServerExtension64.so:/opt/flexnetls/fnedemo/fnedemo/libServerExtension64.so: cannot open shared
object file: No such file or directory
20:15:49,690 ERROR - Binding info retrieval failed.
20:15:49,740 ERROR - Unable to retrieve binding data from producer supplied dll.
20:15:57,676 INFO  - Active profiles: [lls, singletenant, reservations, performance, service]
20:16:02,655 INFO  - New license server instance GCKSB3SZJMV1 has been constructed
20:16:03,745 WARN  - No active hostID specified, choosing "00155D900105/Ethernet"
20:16:33,744 WARN  - Active hostid selected: "00155D900105/Ethernet"
```

The error occurs because the component `libServerExtension64.so` is missing. This is the shared library that contains the logic for retrieving and checking the binding that is currently used by the server.

***Note** ▪ The missing component depends on the operating system on which the local license server is running.*

Building the Shared Library

To resolve the “missing library” error, you need to build the shared library. This demonstration uses the sample file that can be found in `/examples/publisher_defined_binding_identity`.

This example takes the hard-coded ascii string "life is short" as the binding element:

```
const unsigned char binding_identity_data[] = { 0x6c, 0x69, 0x66, 0x65, 0x20, 0x69, 0x73, 0x20,
0x73, 0x68, 0x6f, 0x72, 0x74 };
```

Compiling the Code

To compile this code on Linux, use a command similar to this:

```
gcc -shared -o libServerExtension64.so ExamplePublisherDefinedBinding.c -I../include -fPIC
```

For Windows, use a commend like the following:

```
cl /LD ExamplePublisherDefinedBinding.c /I PublisherDefinedExtensionExports.h /o
ServerExtension64.dll
```

Copying the Shared Library

Copy the shared library to the location of the `flexnetls.jar` file in a directory named after the publisher name. In this example, the `flexnetls.jar` resides in the folder `fnedemo`, which contains a folder also called `fnedemo` (the producer name).

On Linux, use a command like the following to copy the shared library:

```
Sudo cp libServerExtension64.so /opt/flexnetls/fnedemo/fnedemo
```

Running the Server

Now when you start the server you should see output like this:

```
20:29:45,489 INFO  - Starting FlexNet License Server 2022.05.0 (build 266500)
20:29:45,506 INFO  - Copyright (C) 2013-2022 Flexera Software LLC.
20:29:45,506 INFO  - All Rights Reserved.
20:29:45,506 INFO  - Running as a service
20:29:45,894 INFO  - Host Configuration [os=Linux, mem=2.0GB, mem_free=1.5GB, processors=1,
nonheap_committed=33.8MB, nonheap_init=2.4MB, nonheap_used=32.7MB]
20:30:20,514 INFO  - Server hostids acquired: [00155D900105/Ethernet]
20:30:28,340 INFO  - Active profiles: [lls, singletenant, reservations, performance, service]
20:30:32,503 INFO  - Updated license server instance GCKSB3SZJMV1 is ready
20:30:33,622 WARN  - No active hostID specified, choosing "00155D900105/Ethernet"
20:31:03,622 WARN  - Active hostid selected: "00155D900105/Ethernet"
```

Simulating a Binding Break

To simulate a break, change the example code so that the binding is different from what is now stored in the server-side trusted storage. For this example, change the ascii string “life is short” to “life is not short” as shown below:

```
const unsigned char binding_identity_data[] = { 0x6c, 0x69, 0x66, 0x65, 0x20, 0x69, 0x73, 0x20,
0x6e, 0x6f, 0x74, 0x20, 0x73, 0x68, 0x6f, 0x72, 0x74 };
```

Recompile the code, using the same steps as in Compiling the Code. Copy the shared library to the same location as described in Copying the Shared Library.

Running the Server

You should now see output similar to the following when starting the server:

```
20:54:43,200 INFO  - Starting FlexNet License Server 2022.05.0 (build 266500)
20:54:43,204 INFO  - Copyright (C) 2013-2022 Flexera Software LLC.
20:54:43,204 INFO  - All Rights Reserved.
20:54:43,204 INFO  - Running as a service
20:54:43,737 INFO  - Host Configuration [os=Linux, mem=2.0GB, mem_free=1.5GB, processors=1,
nonheap_committed=33.6MB, nonheap_init=2.4MB, nonheap_used=32.5MB]
20:55:19,882 INFO  - Server hostids acquired: [00155D900105/Ethernet]
20:55:20,528 ERROR - Hard binding break detected at startup.
20:55:27,444 INFO  - Active profiles: [lls, singletenant, reservations, performance, service]
20:55:31,417 INFO  - Updated license server instance GCKSB3SZJMV1 is ready
20:55:32,561 WARN  - No active hostID specified, choosing "00155D900105/Ethernet"
20:56:02,560 WARN  - Active hostid selected: "00155D900105/Ethernet"
```

The server starts but will not serve any licenses. If you attempt to perform a capability request from the server, a message similar to the following is displayed:

```
19:16:49,635 ERROR - [1234567890 request] Features will not be available due to hard binding break.
```

Repairing a Binding Break

Repairing a binding break involves sending a capability request to the back office, as shown in the following diagram.

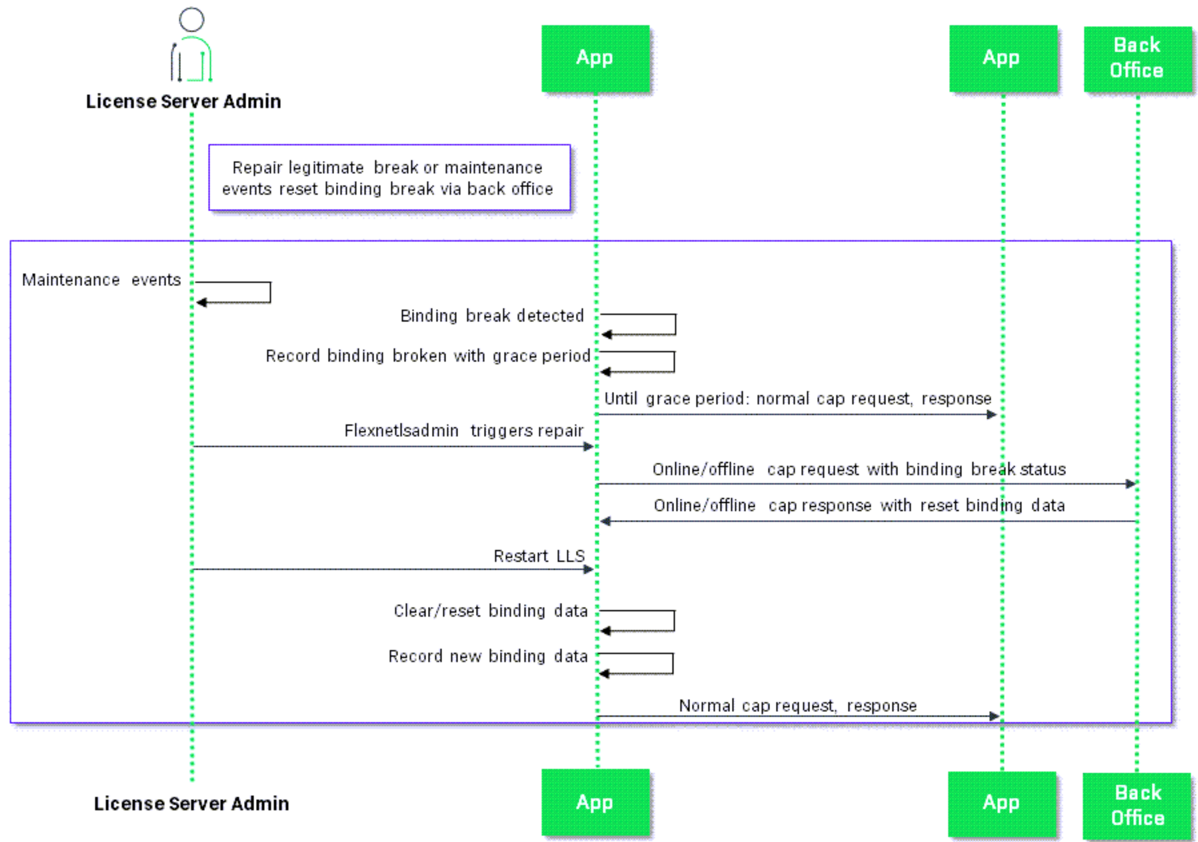


Figure G-3: Workflow for repairing broken binding

It is possible to repair the binding break by sending a capability request to the back office. The server adds the binding break to the request and the back office subsequently creates a repair that will fix the trusted storage and allow licensing activities to continue. In situations where the license server communicates online with the back office, this would occur via capability request polling. In situations where the license server cannot communicate directly with the back office, you need to generate a capability request manually and process it in the back office.

Task ***To repair a binding break***

1. Generate an offline request using flexnetlsadmin:

```
./flexnetlsadmin -server HTTP://localhost:7070/api/1.0/instances/~/ -activate
-id <activation-id> -o request.bin
```

2. Process `request.bin` in the back office (FlexNet Operations) and receive the response

3. Process the response on the local license server:

```
./flexnetlsadmin.sh -server HTTP://localhost:7070/api/1.0/instances/~/ -activate -load
capabilityResponse.bin
```

This procedure also preserves the client history on the server and does not reset trusted storage.

***Note** ▪ flexnetlsadmin does not have an option to send a blank capability request (that is, without an activation id). If you want to send a blank capability request, you can use a tool such as curl to send the request to the **/capability_request** endpoint and capture the output of the response:*

```
curl HTTP://localhost:7070/api/1.0/instances/~/capability_request/offline > request.bin
```