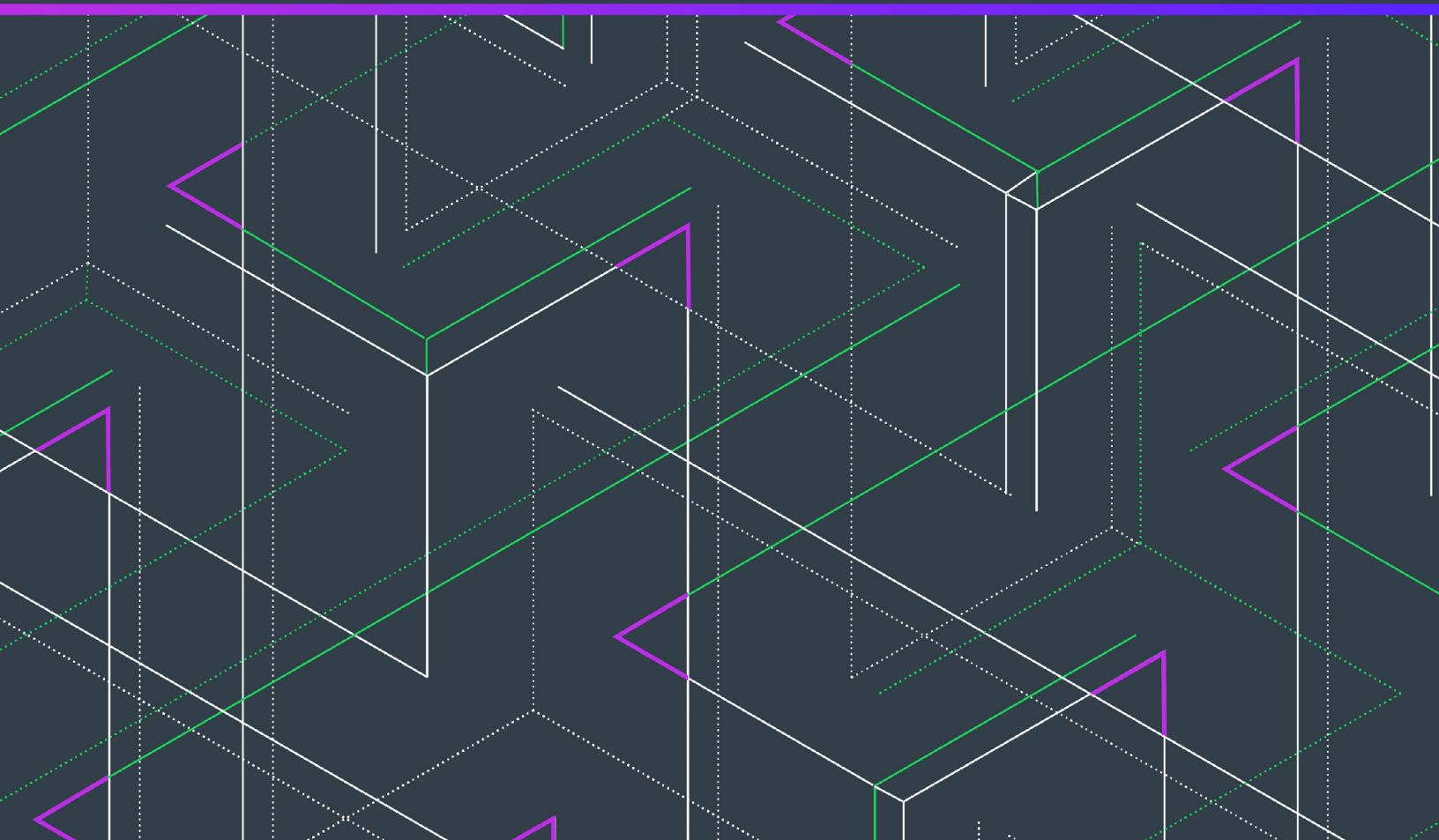
revenera™

Cloud Monetization API 2020 R3

User Guide



Legal Information

Book Name:	Cloud Monetization API 2020 R3 User Guide
Part Number:	CMAPI-2020R3-GD00
Product Release Date:	July 2020
Documentation Last Updated:	April 8, 2019

Copyright Notice

Intellectual Property

Restricted Rights Legend

Contents

1

Introduction

A license server—either a Cloud License Server (CLS) instance hosted by Revenera or a FlexNet Embedded local license server—supports a licensing interface in which capability exchanges are handled using REST APIs, instead of FlexNet Embedded SDK language-based APIs that you incorporate in your client code. This book provides guidance in using this REST interface to check out and return those features to which an end user of your client application has been licensed.

About the Cloud Monetization API

The Cloud Monetization API (CMAPI) uses a REST interface run against a CLS instance or local license server to manage licensing for your product. Keep in mind the CMAPI is not a replacement for SDK-driven licensing. However, it does provide an alternative licensing method suited for certain client applications, such as the following:

- Applications hosted on SaaS platforms on behalf of your customers. Users of such applications are generally licensed by their user log-in information, not by their devices.
- Code installed on small-footprint devices.
- Applications written in languages that the FlexNet Embedded SDKs do not support.

The CMAPI currently offers basic licensing functionality—mainly checkout and return operations—but does not support all the functionality provided in SDK-driven licensing. Although more functionality might be added to this REST interface in future releases, it will never be able to support every function available with the SDK—mainly because, with the CMAPI, the mechanisms used to process client licenses obtained from the license server are completely controlled by your application logic code. Most notably, no FlexNet Embedded trusted storage is implemented within your application.

What's in this Guide

The *Cloud Monetization API User Guide* includes the following chapters:

Table 1-1 • *Cloud Monetization API User Guide*

Topic	Content
Introduction	An overview of the book contents.
Performing a JSON Capability Exchange	Instructions on how to use the Cloud Monetization API to send JSON-formatted capability requests to the license server to obtain licenses for a given enterprise client.
Reference: REST Interface Details for the Cloud Monetization API	A detail summary of those REST APIs in the Cloud Monetization API that perform capability exchanges.
Sample Implementation for Validating Responses	Sample client script for validating signed capability responses when using the Cloud Monetization API.

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

Performing a JSON Capability Exchange

The following sections provide information and instructions for performing capability exchanges using the Cloud Monetization API (CMAPI):

- Accompanying Documentation: License Server Producer Guide
- About JSON Security Setup
- Obtaining Features Using the "/access_request" APIs
- Previewing Available Features Using the "/preview_request" API
- Description of Request Elements
- Securing the Capability Exchange
- Sample Non-Signed Capability Responses
- Sample Signed Capability Response
- Status Codes
- Considerations When Using License Server Failover with the Cloud Monetization API

The instructions in this chapter assume that you have set up JSON security for the capability exchanges (see About JSON Security Setup).

Accompanying Documentation: License Server Producer Guide

The *FlexNet Embedded License Server Producer Guide* provides information about license server configuration, installation, security, and functionality that supports client licensing. It also includes information about accessing the license server's REST interface in general and about setting up security both for those REST APIs that administer the server and for the CMAPI. Consequently, the *Producer Guide* is frequently referenced in the *Cloud Monetization API User Guide* as resource for additional information and should therefore be used in conjunction with this user guide to provide full-scale information about using the CMAPI.

About JSON Security Setup

It is strongly recommended that you set up JSON security to protect the capability exchanges initiated by the licensed client application using the CMAPI. Instructions for this setup process are provided in the section "API for Enabling JSON Security for the Cloud Monetization API" in the *License Server REST APIs* chapter of the *FlexNet Embedded License Server Producer Guide*.

To provide background for the information in this chapter and the remainder of the guide, the following is a brief overview of the JSON security setup. Keep in mind that full details for this setup are found in the *Producer Guide*, as stated in the previous paragraph.

- **Step 1.** Basically, setup for JSON security begins when the producer generates a private-public key pair to be used for capability request validation and distributes this key pair to the enterprise.
- **Step 2.** The client uploads the public key to the license server and generates a JSON Web Token (JWT) signed with the private key.

  As described later in this chapter (see Securing the Capability ExchangeSecuring the Capability Exchange), the licensed client application will attach this JWT to each capability request, enabling the license server to validate the JWT against the public key to ensure that the request is from an authorized client.
- **Step 3.** When the client uploads the public key to the license server (for use by the server to validate the JWT), the server automatically generates a second key pair to be used to validate capability responses.
- **Step 4.** The server presents the public key in this second key pair to the client. If the client is using signed responses, the producer (or the enterprise) stores the key the in a location accessible to the licensed client application (or the client can simply be provided the key).
- **Step 5.** The license server uses the private key in the key pair to generate a JSON Web Token (JWS) for the capability response it sends to the client application (if the request was posted using `/signed_access_request`).

  As described later in this chapter (see Securing the Capability ExchangeSecuring the Capability Exchange), the client application can validate that the public key (provided to the client by the license server) matches the private key used to generate the JWS, thus ensuring that response is from the expected license server and that response contents have not been altered.

Obtaining Features Using the "/access_request" APIs

When your client application performs a capability exchange using the CMAPI, it uses the `POST` method on either of these REST APIs to send a capability request in JSON format to the license server to obtain licenses for the requesting enterprise client:

- `/access_request`
- `/signed_access_request`

The Capability Exchange Process

For both APIs, the JSON request body and the inclusion of the JWT with the request are set up the same, as described in Request Body for "/access_request" and "/signed_access_request" and Means to Attach the JWT to the Capability Request, respectively. Once the request is posted to the license server, the server uses its uploaded public key to validate the JWT, signed with the client-side private key, to determine whether the request has been sent by an authorized enterprise client. If the request has been sent by an authorized client, the license server returns a capability response containing the requested features currently available for the client application.

However, the type of response returned to client application depends on the API used in your application logic. The response returned by the `/access_request` API is *unsigned*, while the response returned by the `/signed_access_request` API is signed with a JWS created with the private key on the license server. When the response is signed, the client application validates the signature against the corresponding public key it has been provided to determine whether the license server is an authorized server in the enterprise and whether the response has been altered in transit.

For details about each of these REST APIs, see the Reference: REST Interface Details for the Cloud Monetization API chapter.

For example capability responses, see Sample Response for "/access_request" and Sample Signed Capability Response.

Request Body for "/access_request" and "/signed_access_request"

The following shows an example capability-request body in JSON format for the `/access_request` and `/signed_access request` REST APIs.

For a description of the elements used in the request, see the later section, Description of Request Elements.

```
{
  "hostId": {
    "type": "string",
    "value": "User-1"
  },
  "incremental": false,
  "borrow-interval": "1d",
  "partial": true,
  "features": [
    {
      "count": 5,
      "name": "f3",
      "version": "1.0"
    },
    {
      "count": 3,
      "name": "f4",
      "version": "1.0"
    }
  ],
  "vendorDictionary": {"RS Support":"no"},
  "selectorsDictionary": {"ROLE": "ANY", "REGION": "EMEA"}
}
```

Previewing Available Features Using the "/preview_request" API

An enterprise client can preview features currently available to it on the license server by sending a capability request designated as "preview". In return, the license server sends a capability response specifying the available features, but the features are for viewing 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; no licenses are actually served to the client.

Based on the preview results, the client application can then send a regular capability request (using the `/access_request` or `/signed_access_request` REST API) to check out the available features.

The following sections describe how to preview features:

- The "preview" Capability Exchange Process
- Request Body for "/preview_request"

The "preview" Capability Exchange Process

To preview available features, the client application uses the `POST` method on the `/preview_request` REST API. The JSON body for the "preview" capability request is similar to that used for the `/access_request` API, but with fewer supported properties, as described in Request Body for "/preview_request". The inclusion of the JWT with the request is set up the same as for `/access_request`, as described in Means to Attach the JWT to the Capability Request. Once the request is posted to the license server, the server uses its uploaded public key to validate the JWT, signed with the client-side private key, to determine whether the request has been sent by an authorized enterprise client. If the request has been sent by authorized client, the license server returns a capability response containing a preview of the features currently available to the licensed client application.

Currently, "preview" capability exchanges do not support signed responses.

For an example of a "preview" capability response, see Sample Response for "/preview_request".

For complete details and policies about previewing available features, see the "Capability Preview" section in the *Advanced License Server Features* chapter of the *FlexNet Embedded License Server Producer Guide*.

Request Body for "/preview_request"

The following sections provide example capability-request bodies in JSON format for the `/preview_request` REST API:

- Request to Preview Availability of Specified Features
- Request to Preview All Available Features

Note the following about "preview" capability requests:

- They do *not* support the `borrow-interval`, `incremental`, `partial`, or `vendorDictionary` options.
- They *do* support the use of feature selectors (using the `selectorsDictionary` option), enabling the license server to filter all features or just specified features according to the selector criteria to determine available features.

For a description of the elements used in the request, see the later section, Description of Request Elements.

Request to Preview Availability of Specified Features

The following shows the body of a capability request to preview the availability of specified features for the requesting client hostid:

```
{
  "hostId": {
    "type": "string",
    "value": "client1"
  },
  "features": [
    {
      "count": 10,
      "name": "f1",
      "version": "1.0"
    },
   {
      "count": 5,
      "name": "f3",
      "version": "1.0"
    }
 ],
  "selectorsDictionary": {"ROLE": "ANY", "REGION": "EMEA"}
}
```

Request to Preview All Available Features

The following shows a request to preview all available features—in this case, without feature selectors. (If feature selectors were included, the response would show a preview of all available features that meet the feature selector criteria.)

```
{
  "hostId": {
    "type": "string",
    "value": "client1"
  },
  "selectorsDictionary": {}
}
```

Description of Request Elements

The following describes the elements in a capability request used by the CMAPI:

Table 2-1 • Elements in Capability Requests Used by the CMAPI

Keyword	Description
hostId	(Required) The client hostid type and value: • `type`—The type of hostid: `string`, `ethernet`, `vm_uuid`, `flexid9`, `flexnid10`, or `user`. • `value`—The hostid. Note that this information is returned in the capability response, enabling producers to match the response to the requesting hostid should they use a proxy to intermediate requests and responses to and from the license server.
borrow-interval	(Required except for `/preview_request`) The current borrow interval that applies to all features in the capability request. Specify this value with a unit-suffix letter—`s`, `m`, `h`, `d`, or `w`—indicating seconds, minutes, hours, days, or weeks. Use **`0`** (zero) to specify no override. When using the `/preview_request` API, this option is not supported and therefore should be omitted.
incremental	Option to renew all currently checked-out concurrent features without having to explicitly re-specify them in the capability request. (For details, see the "Incremental Capability-Request Processing" section in the *Advanced License Server Features* chapter in the *FlexNet Embedded License Server Producer Guide*.) Specify either value: • **`true`**—Attempts to renew all currently checked-out features without having to explicitly specify them in the request. • **`false`**—Does not renew any currently checked-out features unless they are explicitly specified in the request. To disable this option, set its value to **`false`**, or omit the option altogether. When using the `/preview_request` API, this option is not supported and therefore should be omitted or set to **`false`**.  ***Note*** • *You can set a negative count for an incremented feature to renew it with a count that is less than its currently checked-out count. See the* ***features*** *keyword.*

Table 2-1 • Elements in Capability Requests Used by the CMAPI

Keyword	Description
partial	Option to check out all remaining counts for features whose available counts on the server fall short of the their requested counts. (For details, see the "Granting All Available Quantity for a Feature" section in the *Advanced License Server Features* chapter in the *FlexNet Embedded License Server Producer Guide*.) Specify either value: • `true`—Checks out all remaining counts for any feature for which the server is unable to satisfy the requested count. • `false`—Does not check out those features for which the server is unable to satisfy requested counts. This option is specified at the request level when using the CMAPI (whereas in SDK-driven licensing, this option is specified at the feature level). To disable this option, set its value to `false`, or omit the option altogether. When using the `/preview_request` API, this option is not supported and therefore should be omitted or set to `false`.
features	The count, name, and version of each requested concurrent or metered feature. Note the following: • To return a concurrent feature, set its count to `0`. • (When `incremental` is `true`) To renew a concurrent feature with a count less than its current checked-out count, explicitly include that feature in the capability request and set its count to a negative value. When the feature is renewed, its count will be its previously checked-out count decremented by the negative value. (To *not* renew a current feature, use a negative value that negates the currently checked-out count.) • For metered features, currently no support exists for the "undo" operation or for correlation, requestor, or acquisition IDs.

Table 2-1 • Elements in Capability Requests Used by the CMAPI

Keyword	Description
vendorDictionary	One or more dictionary entries used to send producer-defined data to the license server. For details, see the "Vendor Dictionary Data" section in the *Advanced License Server Features* chapter of the *FlexNet Embedded License Server Producer Guide*. A vendor dictionary item is entered in either format: ● *"key_string":"string_value"* ● *"key_string":integer* Separate multiple items with commas: `"vendorDictionary": {"compID": 5160 , "Dept" : "Acct" , "team" : "A1"}` To send no dictionary entries, provide no entries between the brackets, or omit the option altogether. When using the `/preview_request` API, this option is not supported and therefore should be omitted. The maximum dictionary size is 7168 bytes after the dictionary's conversion to base64.

Table 2-1 • Elements in Capability Requests Used by the CMAPI

Keyword	Description

selectorsDictionary	One or more feature selectors (that is, feature attributes) that are in addition to the feature names, versions, and counts specified in the capability request. The license server processes these selectors as means of filtering features to include in the capability response. The response will send only those features available to the requesting hostid that meet the all specifications—feature name, version, count *and* all selector criteria—defined in the request.
	
	Purpose of Feature Selectors
	
	The feature selectors provide the additional criteria needed by the producer to license clients from individual pools of entitlements, such as for a given region or role. For example, two features with the same name (`f1`) might each be defined with a different feature selector (one selector specifying `auditing`, the other `payroll`). A client in the Auditing department can then request `f1`, specifying `auditing` as the feature selector, to obtain the version of f1 appropriate for the department.
	
	Feature Selector Setup
	
	Feature selectors are defined as vendor strings for a given feature in the back office and sent with the feature when the license server obtains it. The license server will attempt to filter features by selectors only if the request specifies the `selectorsDictionary` option with a set of selectors. In turn, these selectors must match those already defined for the features that the client wants to obtain. (To view the selectors, if any, defined for a given feature, access the `/features` REST API on the license server, as described in the *License Server REST APIs* chapter of the *FlexNet Embedded License Server Producer Guide*.)
	
	Note that the `selectorsDictionary` option is defined at the request level and thus applies to all features specified in the request.
	
	For complete details and policies for defining and using feature selectors, see the "Feature-Selector Filtering" section in the *Advanced License Server Features* chapter of the *FlexNet Embedded License Server Producer Guide*.
	
	Specification in the Request
	
	The following format is used to specify feature selectors:
	
	`"selectorsDictionary": {"selectorKey": "value", "selectorKey": "value"}`
	
	An example is as follows:
	
	`"selectorsDictionary": {"ROLE": "ANY", "REGION": "EMEA"}`
	
	To include no feature selectors by which to filter, provide no entries between the `selectorsDictionary` brackets, or omit the option altogether.

Securing the Capability Exchange

When JSON security is enabled on the license server, your licensed client application code provides the following:

- Means to Attach the JWT to the Capability Request
- Means to Validate Response Signature (if the request was posted using `/signed_access_request`)

Means to Attach the JWT to the Capability Request

The client application is required to include the client's JWT with each capability request sent either through the `/access_request`, `/signed_access_request`, or `/preview_request` REST API. (The JWT is signed with the client's private key that matches the public key uploaded to the license server.) The client code must have a means to obtain this token string—for example, by embedding the token in the client code itself for easy retrieval, providing access to a suitable token in a location external to the application, or generating the token using the private key (not the recommended method).

Once client retrieves the token, it includes the token in an "Authorization" HTTP header, specifying the "Bearer" type, as shown in this example:

```
Authorization: Bearer JKV1QiLCJhbGciOiJIUzeyJ0eXAiOiUxMiJ9... [example token value truncated for purposes of space]
```

Then, when a client sends the JSON capability request to the license server, the server determines that the request is from a valid client 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. If the server cannot read the token, the client making the request will be denied the features. Additionally, the token can include expiry information that the server can read to determine the validity of the request.

Means to Validate Response Signature

If the licensed client application calls the `/signed_access_request` REST API, it should also incorporate a means to validate that the private key used to sign the JWS on the license server matches the public key that has been provided to the client application. For an example implementation, see the chapter Sample Implementation for Validating Responses.

Sample Non-Signed Capability Responses

The following sections provide information about the capability response returned for the various capability request APIs:

- Sample Response for "/access_request"
- Sample Response for "/preview_request"
- Details to Note in the Response

Sample Response for "/access_request"

The following shows a sample capability response body returned when using the POST method on the `/access_request` REST API.

Note that the preview capability request corresponding to this response requested a feature that is not available on the license server. The response uses the `StatusList` section to specify that feature.

```
{
  "features": [
    {
      "expires": "2018-01-14T21:09:19.000Z",
      "version": "1.0",
      "entitlementExpiry": "2020-12-31",
      "count": 5,
      "finalExpiry": "2021-01-31",
      "name": "f3"
      "vendorString": "%%REGION:Global%%"
    },
    {
      "expires": "2018-01-14T21:09:19.000Z",
      "version": "1.0",
      "entitlementExpiry": "2020-07-04",
      "count": 3,
      "finalExpiry": "2021-01-31",
      "name": "f4"
      "vendorString": "%%REGION:EMEA%%"
    }
  ],
  "statusList": [
    {
      "message": "Feature is not available: f8 - 1.0",
      "code": "FEATURE_NOT_AVAILABLE",
      "name": "f8",
      "version": "1.0"
    }
  ],
  "requestHostId": {
    "type": "STRING",
    "value": "User-1"
  }
}
```


Sample Response for “/preview_request”

The following shows a sample capability response body returned when using the POST method on the `/preview_request` REST API.

Note that the preview capability request corresponding to this response included feature selectors. The response lists the available features that meet the selector criteria of the requesting client and uses the `StatusList` section to specify those that did not meet the criteria.


```
{
  "requestHostId": {
    "type": "STRING",
    "value": "client1"
  },
  "features": [
    {
      "version": "1.0",
      "finalExpiry": "permanent",
      "name": "f2",
      "count": 100,
```

```
      "expires": "2018-03-16T17:09:37.000Z",
      "maxCount": 100
      "vendorString": "%%REGION:Global%%"
    },
    {
      "version": "2.0",
      "finalExpiry": "permanent",
      "name": "f2",
      "count": 1,
      "expires": "2018-03-16T17:09:37.000Z",
      "maxCount": 1
      "vendorString": "%%REGION:EMEA%%"
    }
  ],
  "statusList": [
    {
      "message": "Feature is not available due to a server checkout filter rejection: f4 - 1.0",
      "code": "FEATURE_REJECTED_BY_CHECKOUT_FILTER",
      "name": "f4",
      "version": "1.0"
    },
    {
      "message": "Feature is not available due to a server checkout filter rejection: f3 - 1.0",
      "code": "FEATURE_REJECTED_BY_CHECKOUT_FILTER",
      "name": "f3",
      "version": "1.0"
    }
  ]
}
```


Details to Note in the Response

The following explains information sent in the capability response that is in addition to the information specified in the capability request (and also returned in the response):

- Expiration Information in a Response
- Status and Client Hostid Information in a Response
- Counts in a Preview Response

Expiration Information in a Response

Note that the capability response can provide the following expiration information for a given feature:

- The `expires` element shows the timestamp for the expiration of the current borrow period.
- The `entitlementExpiry` shows the original expiration date for the feature as specified in the entitlement. This date does not include a grace period, even if a grace period is defined in the back office.
- The `finalExpiry` shows the final expiration date for the feature—that is, the `entitlementExpiry` date plus the grace period defined in the back office. If no grace period is defined in the back office, this date is the same as the `entitlementExpiry`.

For more information about the entitlement and final expirations for a feature, see the "Viewing Features in an Expiration Grace Period" section in the *License Server REST APIs* chapter of the *FlexNet Embedded License Server Producer Guide*.

Status and Client Hostid Information in a Response

The status list (empty in the sample body) is used to provide information about requested features that could not be checked out, as in this example status:

```
"statusList": [
  {
    "message": "Insufficient feature count available: f3 - 1.0",
    "code": "FEATURE_COUNT_INSUFFICIENT",
    "name": "f3",
    "version": "1.0"
  }
],
```

Additionally, information about the requesting client hostid sent in the capability request is returned as `requestHostId` details in the response. This information in the response might be useful, for example, in allowing producers to build a configuration that uses a licensing proxy to funnel requests to the license server. The client hostid enables producers to tie the response back to the correct client.

```
"requestHostId": {
  "type": "STRING",
  "value": "User-1"
}
```

Counts in a Preview Response

The preview capability response contains 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 explicitly requested to preview the availability of certain features (each with a specific version and count), the returned value for each available feature is the count that would be served had the client used the `/access_request` or `/signed_access_request` REST API to request the same features.
  - If it requested to preview all features currently available to the client (by not specifying any features explicitly in the request), the returned value for each feature shows the immediately available count—that is, the count reserved for the client plus all shared counts that are not currently served to other clients in the enterprise.
- **maxCount**—The potentially available count that includes the count 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.)

Various factors can affect the calculation of these two counts, as described in the "Capability Preview" section in the *Advanced License Server Features* chapter of the *FlexNet Embedded License Server Producer Guide*.

Based on the preview results, the client application can decide to check out the available features (using the `/access_request` or `/signed_access_request` REST API).

Producer-defined License Data in a Response

The `vendorString` field can take any arbitrary string that has been defined by the producer in the back office during the creation of a feature or a license model. The producer can use this field to include any additional information (such as, for example, the activation ID, feature selectors, etc.) in a licensing model, to execute custom logic on the client.

Sample Signed Capability Response

The following describes the capability response returned by `/signed_access_request`:

- Response Body
- Description of Response Headers

Response Body

The response returned for the `/signed_access_request` API is RFC7515 compliant (a standard for JSON clients) and uses a flattened JWS JSON serialization syntax, as shown in this example response:

```
"header": "ew0KICAia2lkIiA6IDE1MTM2Nzk0ODE4NTkNCn0",
"payload": "ew0KICAiZmVhcHVyZD0NiIDogWyB7DQogICAgIm5hbWUiIDogImYxIiwNCiAgICAiDRvYC21...[truncated]"
"signature": "bRiMOM-QPQ5_PzWDBfVIxjMSZCFVEkgPecwQT8WqUGAtRBsPe334POTRSgSrY1vx2dd6cy...[truncated]"
"protected": "-ew0KICA1YWboNIiA6ICJSUzI1NiINCn0"
```

Description of Response Headers

The following describes the headers used in the JWS JSON syntax:

- **"header"**—The public key ID (kid)—an unprotected header—defining the Unix Epoch timestamp when the public key was created on the license server. The licensed client application can convert this value to a readable timestamp to determine whether the public key has changed.
- **"payload"**—The BASE 64-encoded message identifying the features and status information sent by the license server. The licensed client application can provide a means to decode the response payload to a readable JSON format for use by the client application (for example, to enable the enterprise client to verify the checked out features). The resulting decoded message is in the same JSON format as the response returned for the `/access_request` REST API. See Sample Non-Signed Capability Responses for details.
- **"signature"**—The signature field is a combination of the protected and payload fields, which is then encrypted using the private key stored within the license server. The public key provided by the license server (and made available to the client application independently) is then used to decrypt the signature, enabling the client to confirm whether the sender and the contents of the payload are valid.

  For details about how response validation is enabled during JSON security setup, refer to the section "API for Enabling JSON Security for the Cloud Monetization API" in the *License Server REST APIs* chapter of the *FlexNet Embedded License Server Producer Guide*.
- **"protected"**—The algorithm (RS256) used to sign the response.

Status Codes

The following provides a reference to the most common status codes used to signal success or failure of a REST API invocation:

Table 2-2 • 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.
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.

For a list of exception codes (including those pertaining to the CMAPI) that the license server can generate, see the "Exception Handling" section in *License Server REST APIs* chapter of the *FlexNet Embedded License Server Producer Guide*.

Considerations When Using License Server Failover with the Cloud Monetization API

License server failover has limited compatibility with JSON security set up for the CMAPI, 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. However, the failover process can support JSON security (if this security is enabled) when the client uses the `/access_request` REST API to obtain features. In this scenario, a JSON Web Token—signed with the client's private key generated as part of a private-public key pair by the producer or the enterprise—is attached to the request, enabling the license server to then use the public key to validate the client. This validation depends on the license server administrator first uploading the public key *to both the main server and the backup server* in separate `rest_licensing_keys` calls. (For information about this REST API used to transmit public keys, refer to the section "API for Enabling JSON Security for the Cloud Monetization API" in the *License Server REST APIs* chapter of the *FlexNet Embedded License Server Producer Guide*).

The failover solution does not work with the `/signed_access_request` REST API, as no method exists to copy the private key, internally generated as part of a key pair on the main server to sign capability responses, to the backup server.

For details about license server failover, see the "Failover Using Synchronization to FlexNet Embedded" section in the *More About Basic License Server Functionality* chapter in the *FlexNet Embedded License Server Producer Guide*.

3

Reference: REST Interface Details for the Cloud Monetization API

This chapter is a reference to information about the REST APIs used by the Cloud Monetization API (CMAPI):

- Base URLs
- REST API Reference
- Online REST API Reference Documentation Available

Base URLs

Your licensed client application checks out and returns licenses at the license-server instance level. Consequently, when using the CMAPI, you must prefix the endpoint for each REST API with a base URL containing the instance ID for the CLS instance or local license server. The base URL generally has the following format:

`https://`*`licenseServerHost`*`/api/1.0/instances/`*`instanceID`*

Refer to the following for specific detail:

- For a CLS instance, the *licenseServerHost* is the address of the Cloud-hosting service (which can optionally include a port number); the *instanceID* is the actual ID generated for the instance when it is created. The following shows an example prefix for a Revenera-hosted CLS instance:

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

- 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. A prefix for a local license server might look similar to this:

  `https://11.11.1.111:443/api/1.0/instances/~`

Using the previous example for the Revenera-hosted CLS instance, the endpoint for the `/access_request` API would be the following:

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

It is your responsibility to provide the base URL to the client application.

For more information about the base URL, see the "Accessing the REST APIs" section in the in the *License Server REST APIs* chapter of the *FlexNet Embedded License Server Producer Guide*.

REST API Reference

The following is a summary of the REST interface available to perform capability exchanges when using the CMAPI:

- "/access_request" API Details
- "/signed_access_request" API Details
- "/preview_request" API Details

"/access_request" API Details

The client accesses the `/access_request` API to send the JSON-formatted capability request to the license server and to receive the available features in an unsigned capability response, also in JSON format. If JSON security is enabled, the JWT generated client-side must be attached to each request for validation on the license server. The following lists the API details:

URI	`/access_request`
Method	`POST`
Query parameters	N/A
Request body	`application/json` For an example body, see Sample Non-Signed Capability Responses in the *Performing a JSON Capability Exchange* chapter.

Error codes	`glsErr.serverNotFound`
	`glsErr.jsonLicensingSecurityNoToken`
	`glsErr.userAuthFailed`
	`glsErr.JsonValidationError`
	`glsErr.restParsing`

"/signed_access_request" API Details

The client accesses the `/signed_access_request` API to send a JSON-formatted capability request to the license server and to receive the available features in a *signed* capability response formatted in a flattened JWS JSON serialization syntax. This API requires that the appropriate public keys have been posted to the client and license server (as described in the section "API for Enabling JSON Security for the Cloud Monetization API" in the *License Server REST APIs* chapter of the *FlexNet Embedded License Server Producer Guide*).

Additionally, if the `licensing.security.json.enabled` license server policy is enabled, the JWT that is generated client-side must be attached to each capability request for request validation on the license server.

The following lists the API details:

URI	`/signed_access_request`
Method	`POST`
Query parameters	N/A
Request body	`application/json` (see an example body in Sample Signed Capability Response in the *Performing a JSON Capability Exchange* chapter)

Error codes	`glsErr.serverNotFound`
	`glsErr.jsonLicensingSecurityNoToken`
	`glsErr.userAuthFailed`
	`glsErr.JsonValidationError`
	`glsErr.restParsing`

"/preview_request" API Details

The client accesses the `/preview_request` API to post a JSON-formatted capability request to the license server and to receive a preview of the available features in an unsigned capability response, also in JSON format. If JSON security is enabled, the JWT generated client-side must be attached to each request for validation on the license server.

This capability exchange enables the client to view the features available to it but causes no change to the licensing state of the license server or the client. That is, no feature, reservation, or client records are updated on the license server; no licenses are actually served to the client.

The following lists the API details:

URI	`/preview_request`
Method	`POST`
Query parameters	N/A
Request body	`application/json` For an example body, see Sample Non-Signed Capability Responses in the *Performing a JSON Capability Exchange* chapter.

Error codes	`glsErr.serverNotFound`
	`glsErr.jsonLicensingSecurityNoToken`
	`glsErr.userAuthFailed`
	`glsErr.JsonValidationError`
	`glsErr.restParsing`

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`

4

Sample Implementation for Validating Responses

If the licensed client application calls the `/signed_access_request` REST API in order to obtain features in a JWS-signed response, the application should validate the response signature. It does so by using the public key that it has been provided to verify that the signature in the JWS, signed by the server's private key, matches the payload. This chapter provides a sample implementation for performing this validation, as detailed in these sections:

- Requirements for Running the Sample Python Script
- Sample Script for Validating the Response Signature
- Example Responses

For example commands and code implementations used to set up JSON security before performing capability exchanges, see the section “API for Enabling JSON Security for the Cloud Monetization API” in the *License Server REST APIs* chapter of the *FlexNet Embedded License Server Producer Guide*.

Requirements for Running the Sample Python Script

The sample script provided in this chapter was created using Python. However, you can use any appropriate programming language and libraries to perform the verification within your application.

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 Script for Validating the Response Signature

The following is a sample Python script used to validate the JWS with which the capability response is signed.

```
#! /usr/bin/python
import jwt
from jwt.api_jws import PyJWS
from jwt.exceptions import DecodeError
import json
import datetime
import sys

jws = PyJWS();

# Basic check on CLI arguements
if len(sys.argv) != 3: exit("Usage: " + sys.argv[0] + " <Public key PEM> <JWS>")

# Read the puiblic key from PEM file
public_key = open(sys.argv[1], 'r')
public_key_string = public_key.read()
public_key.close()

# Read JSON input from file
jws_recieved = open(sys.argv[2], 'r')
jws_recieved_json = json.load(jws_recieved)
jws_recieved.close()

# Convert to compact representation for PyJWS library
jws_encoded = (jws_recieved_json["protected"]
    + '.' + jws_recieved_json["payload"]
    + '.' + jws_recieved_json["signature"])

# Use library to check signature
try:
    decoded_payload = jws.decode(jws_encoded, public_key_string)
except DecodeError:
    # On failure exit with error code
    print("Signature verification failed")
    sys.exit(1)
else:
    # Print the valid JSON payload
    print(decoded_payload)

sys.exit(0)
```

Example Responses

The following are sample response messages and exit codes returned by the sample script for the situations described.

Response decoded with the correct public key

The following is a sample response indicating that the payload has been decoded with the correct public key—that is, the public key that matches the private key used to encrypt the signature in the JWS:

```
$ ./VerifyJWS.py ExampleServerPublic.pem SignedResponse.json
{
  "features" : [ {
    "name" : "f1",
    "version" : "1.0",
    "count" : 3,
    "expires" : "2017-12-19T11:39:34.000Z",
    "finalExpiry" : "permanent"
    "vendorString" : "%%REGION:Global%%"   } ],
  "statusList" : [ ]
}
$ echo $?
0
```

Response that has been tampered with to specify more counts for a feature than the license server provided

The following is a sample response indicating that the payload has been tampered with to specify more counts than the license server actually provided:

```
$ ./VerifyJWS.py ExampleServerPublic.pem SignedResponseTampered.json
Signature verification failed
$ echo $?
1
```

Response that has been signed by a license server different from the one expected by the client (thus the private and public keys used do not match)

The following is a sample response indicating that it has been signed by an unauthorized license server:

```
$ ./VerifyJWS.py OtherPublic.pem SignedResponse.json
Signature verification failed
$ echo $?
1
```