

JLicensure User Guide

Note: This is a preliminary version of the JLicensure User Guide. The contents and layout of this document are subject to change in the near future.

Introduction

Before you can effectively use your own licensing system with JLicensure, there are some fundamentals that you should understand. This documentation explains the basic principles of JLicensure and how you can use those principles to design your very own licensing system on top of them.

Understanding licenses

JLicensure's main design principle is to keep the system as generic and simple as possible. In JLicensure terms, a *license* is basically nothing more than a signed *license request*. And a *license request* is basically nothing more than an arbitrary piece of data sent to the server by the client application.

To examine the nature of a *license request*, please take a look at the following simple example:

License Request

Dear license-granting server,

I am **John Foobar** and I am using your software called "**CoolProduct**", version **2.0**.

I was wondering if you could grant me a license to run your software on my personal computer today. My current hard and software configuration is: (...) / My current environment parameters are: (...)

Upon reception of a license request, the server decides whether it wants to grant the license or reject it. To do that, the license request usually contains some sort of identifying information, for instance a software serial number, or in this case, the name, software and version number.

Based on the given information, the server either rejects the request or it simply signs it and sends it back, as shown below:

License

License Request

Dear license-granting server,

I am John Foobar and I am using your software called "CoolProduct", version 2.0.

I was wondering if you could grant me a license to run your software on my personal computer today. My current hard and software configuration is: (...) / My current environment parameters are: (...)

Signature: License Server

The client software checks the validity of the server's signature and proceeds execution if it is valid.

If you look at the license request above, you can see that the information contained in the request can actually be divided into two separate parts.

1. One part contains all the information that is needed by the server to **identify the licensee (LicenseeData)** (in this case, "John Foobar" using "CoolProduct", version "2.0"). The server's decision on whether a license should be granted or not is based on that information (in our example, the server could use the information to see whether "John Foobar" actually bought a copy of "CoolProduct", version "2.0" earlier).
2. The other part contains the rest of the information provided in the license request that additionally constrains the validity of the license. The server does not need to parse that information, it doesn't even need to understand the contents of that information, it simply signs it together with the whole license request. Because of this, the client doesn't have to transmit this part of the information, instead, a simple hash value generated from the information is transmitted. We call this hash value the **"License Token"**.

Reflecting the above considerations, the LicenseRequest interface is defined as shown below:

```
/**
 * A license request consists of the licensee data describing the licensee
 * and a unique data token to be part of the issued license.
 */
public interface LicenseRequest extends Command
{
    /**
     * Getter for property licenseeData.
     * @return Value of property licenseeData.
     */
    public LicenseeData getLicenseeData();

    /**
     * Getter for property token.
     * @return Value of property token.
     */
}
```

```
public byte[] getToken();  
}
```

Licensee Data

To give the licensee data some sort of structure while trying to keep things as simple and convenient as possible, we defined a `LicenseeData` which basically holds a generic set of string properties:

```
public interface LicenseeData extends java.io.Serializable  
{  
    (...)  
  
    /**  
     * Get a specific property value.  
     * @param property the property name  
     * @return the property's value  
     */  
    public String getProperty(String property);  
  
    /**  
     * Set a specific property value.  
     * @param property the property name  
     * @param value the property's value  
     */  
    public void setProperty(String property, String value);  
  
    (...)  
}
```

Using this interface, constructing the licensee data for the above license request would look something like this:

```
LicenseeData licenseeData = newLicenseeData();  
  
licenseeData.setProperty("name", "John Foobar");  
licenseeData.setProperty("product", "CoolProduct");  
licenseeData.setProperty("version", "2.0");
```

License Token

Even though the license token is just a small number of bytes containing an arbitrary hash value, it is in fact a powerful tool to constrain the validity of a license. Since the token is constructed on the client side, there are numerous ways to bind the license to any given system parameter available on the client machine – without having to know about those parameters on the server side. All the information that is used to construct the token stays where it is, on the client side, only the token is transmitted. This allows the client software to potentially use personal system properties to bind the license to, while still ensuring your customer's privacy.

For example, a common practice to construct a token is to use information that uniquely identifies the machine the software is run on – that way the license is bound to that particular machine and is rendered invalid when the software is transferred to different computer.

Several approaches as to how the token can effectively be used in different types of

licensing scenarios are to be discussed later.

License Table Design

The default-implementation of the JLicensure Server uses an SQL database table to store licenses. The design of that license table is up to you, you can create a new table from scratch or, if feasible, use an existing database table. The table should provide the following columns:

- A **token** and a **signature** column of type varchar and a **count** column of type integer. Please see the JLicensure Installation Guide for further information.
- A varchar column for every property to be used by the LicenseeData to identify incoming license requests. In our example, that is three columns named "name", "product" and "version". A more appropriate way to identify licenses might be a unique software serial number instead of a person's name.
- A varchar column for every additional property to be contained in the LicenseeData when adding new licensee data to the table using signed commands (see JInstaller Installation Guide, section JLicensure WebModule). For example, you might want to store a credit card transaction number together with the newly inserted row if you're adding licenses from an automatic web shop system.
- Additional arbitrary columns to be used by you or other software parts accessing the license table.

Licensing Scenarios

Depending on the data that is used to construct the license token, licenses can be bound to any kind of data available on the client-side, allowing a great variety of different licensing models. A license stays valid as long as the information used to construct the license token doesn't change.

To check the existence and the validity of a license, the client software constructs a license request and invokes the checkLicense method of the LicenseCheck interface. The default-implementation backs up previously downloaded licenses in the client computer's local preferences cache. If no license is available in the local cache or if it has become invalid, a license is automatically requested from the server.

To construct the token, the client software adds data to the LicenseRequestBuilder interface by calling one of its add-methods.

Machine-based licenses

Probably one of the most common practice is to bind a license to the unique installation of the local machine. To do that, one may add a number of descriptive hardware identifiers to the LicenseRequestBuilder interface. For your convenience, the LicenseRequestBuilder interface has a method addHostUnique() that does the job for you:

```
LicenseRequestBuilder requestBuilder = JLicensureFactory.createLicenseRequestBuilder();
```

```
requestBuilder.addHostUnique();  
LicenseRequest req = requestBuilder.toLicenseRequest(licenseeData);
```

A license obtained from a license request as shown above will stay valid as long as the software is not transferred to a different machine. Because of that, it is usually enough to perform the license check once after the program has been started.

Here's a different example. Let's assume we wanted to create licenses that are valid for only one specific user name, but for all computers in the same network subnet. In this case, the license request construction would look like this:

```
LicenseRequestBuilder requestBuilder = JLicenseFactory.createLicenseRequestBuilder();  
requestBuilder.addArbitrary(getSubnetString().getBytes());  
requestBuilder.addArbitrary(getUserName().getBytes());  
LicenseRequest req = requestBuilder.toLicenseRequest(licenseeData);
```

Expiring licenses

Counter-based:

A license may expire after a program or a program function has been executed for a certain number of times. In this case, it is necessary to obtain a new license from the server every time the function or the program is executed (because a counter can only be safely stored on the server side).

Since the default-implementation of the server's LicenseGrant interface already uses a count-based granting mechanism, all you have to do is initialize the "count" column in the server's license table with the desired number and make sure that the granted licenses are valid only once when requested. You can do that by adding random data to the LicenseRequestBuilder:

```
LicenseRequestBuilder requestBuilder = JLicenseFactory.createLicenseRequestBuilder();  
requestBuilder.addArbitrary(getRandomData());  
LicenseRequest req = requestBuilder.toLicenseRequest(licenseeData);
```

Time-based:

Even though you could add time-based values (such as the current year) when constructing the license token, a simple token-based solution might be problematic due to the following reasons:

- A license that is valid only before a certain expiration date cannot be constructed on the client-side because the expiration date is not known to the client.
- The date and time information obtainable on the client computer may not be accurate.

A safe way to install time-based licenses is to add a column "expires" to the server's license table and install a custom LicenseGrant implementation that rejects license request after the expiration date.