
# Lightstreamer JMS Extender

Last updated: 5th February 2016

# Table of contents

# 1 Introduction

The Lightstreamer JMS Extender enables you to use the Java Message Service API from within any JavaScript code. This way, any HTML page running inside a web browser, as well as any Node.js application, can easily become a JMS client.

On the client side, the full JMS API is exposed as part of the provided JavaScript client library.

On the server side, a specially configured Lightstreamer Server, together with a specially crafted extension of its kernel, is provided to connect to any JMS broker.

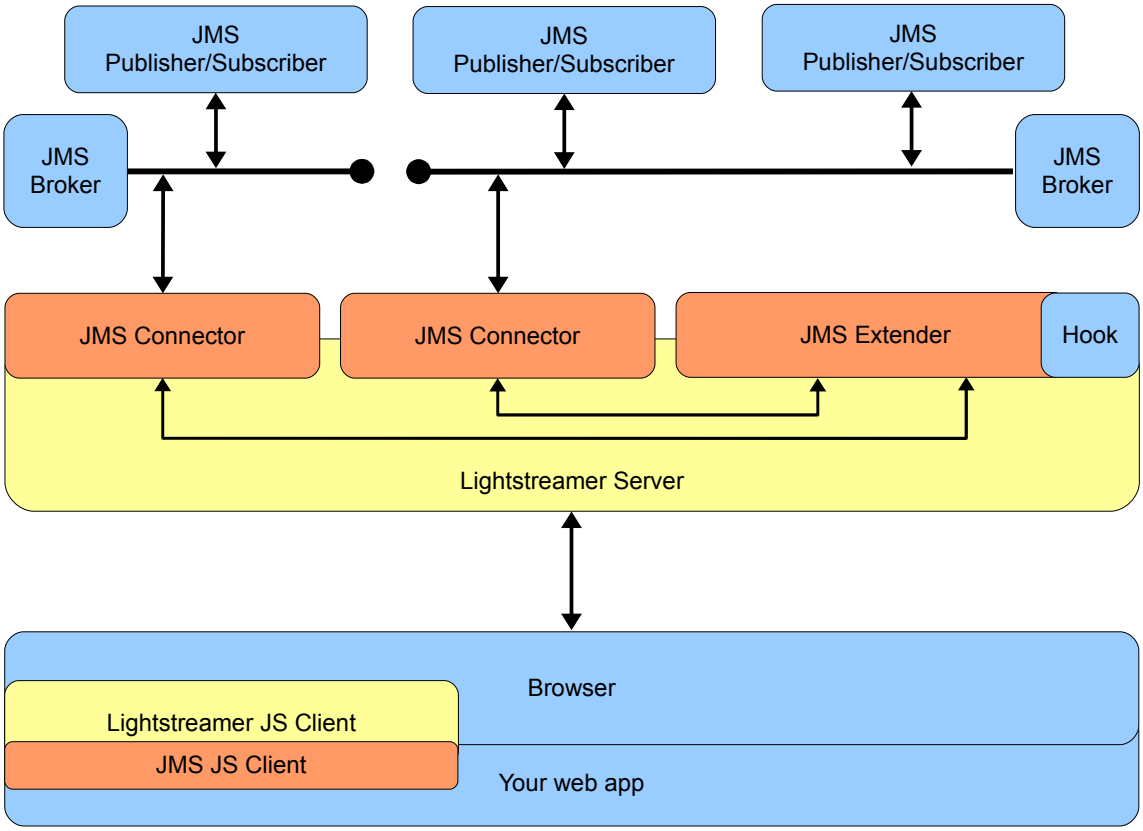
The highly optimized Lightstreamer engine and protocol are used to deliver all types of JMS messages over standard HTTP and WebSockets, with low latency and massive scalability.

This guide describes how the JMS Extender works, together with its configuration and deployment. Common knowledge of JMS principles are required to fully understand what follows.

# 2 Features and Architecture

The JMS Extender is a set of Lightstreamer components that provides access to the regular JMS API for any HTML or JavaScript application. Its main features are:

- Full JMS API available on any Web browser, as well as any Node.js application, as a standard JavaScript library.
- Access to any JMS broker through the Internet, passing through any proxy, firewall, and other network intermediaries.
- JMS connection pooling, offloading and automatic reconnection, for best performance, scalability and availability.
- Full support for specific JMS characteristics like once-and-only-once message delivery.
- Hook API for custom authentication and authorization, directly managed by the JMS Extender.




The diagram above shows the big picture of the JMS Extender architecture, with color-coded boxes depending on which layer they belong to:

- yellow boxes belong to common Lightstreamer server and client layer;
- orange boxes belong to the JMS Extender layer;
- blue boxes belong to third-party/application layer.

The JMS Extender is made up of three main components:

- *JMS Connectors*: their duty is to send and receive messages from topics and queues, and deliver them on specific Lightstreamer items.
- *JMS Extender*: a Lightstreamer Server kernel extension whose duty is to coordinate the activity of JMS connectors and to optionally authenticate users and authorize their requests. Custom authentication and authorization can be implemented through the provided Hook API.
- *JMS Extender JavaScript Client*: its duty is to provide a JMS-like API based on JavaScript, while employing, under the hood, the Lightstreamer JS client library.

## Relationship with Lightstreamer Server

If you are not familiar with Lightstreamer Server terminology, here is a quick reference:

- the **Lightstreamer Server** itself is a stand-alone process that runs in a Java Virtual machine; it handles the connections with the clients and dispatches real-time data back and forth between the clients and any back-end system;
- Lightstreamer **adapters** are custom server-side components attached to the Lightstreamer Server, whose role is to interface the Kernel with any data source, as well as to implement custom authentication and authorization logic; adapters subdivide the data set they provide in different **items**;
- clients connect to the Lightstreamer Server by opening a **session**;
- clients get asynchronous real-time **updates** from the Lightstreamer Server through a **subscription** to a specific **adapter item**;
- clients can also send real-time **messages** to the Lightstreamer Server through a **sendMessage** call.

The JMS Extender is built on top of state-of-the-art Lightstreamer technology, but it's not a common adapter set. It is an extension of the Lightstreamer kernel, fully integrated with its thread pooling, logging and monitoring subsystems.

The mapping between Lightstreamer Server concepts and typical JMS entities is the following:

- a **JMS connection** is implemented through a **Lightstreamer session**;
  - NOTE 1: clients can reconnect automatically to Lightstreamer when a network drop occurs, in this case the JMS connection is re-established automatically on a new Lightstreamer session;
  - NOTE 2: since one JMS connection can host more than one JMS session, JMS sessions are multiplexed on a single Lightstreamer session by assigning them a unique identifier (a GUID).
- **JMS destinations** are implemented through **adapter items**, where each item represents unique pair of JMS session and destination; when topic session sharing is enabled, an item representing a topic may be shared among many JMS sessions; for this reason the number of consumers and items may be different.
- **JMS messages** traveling **from a broker to clients** are called "downstream" messages, and are delivered to clients through **updates**;
  - NOTE: while one JMS message is always delivered with one update to a single client, the same JMS message may be delivered to multiple clients with multiple updates (one for each client) when topic session sharing is enabled (see Configuration and Deployment and later *Special Considerations*); for this reason the number of sent downstream messages and updates may be different.
- **JMS messages** traveling **from clients to a broker** are called "upstream" messages, and delivered to the JMS Extender through **sendMessage** calls;
  - NOTE: while one JMS message is always delivered with one sendMessage call, multiple sendMessage calls may be used to implement the JMS session's acknowledge mode (see later *Special Considerations*); for this reason the number of receive sendMessage calls and upstream messages may be different.

Since the Monitoring Dashboard provides all the indicators typical of a Lightstreamer Server, in addition to those specific of the JMS Extender, this mapping provides important information to better understand the Extender's condition. Keep it for reference in case of doubt.

# 3 Configuration and Deployment

The deployment of JMS Extender is an easy process. A step-by-step guide is provided to configure connectors for one or more JMS broker.

## Requirements

The JMS Extender requires:

- a Java 7 or newer Java Virtual Machine (the most recent available JVM is always recommended);
- the official JMS 1.1 or 2.0 API jar;
- a JMS broker compliant with JMS specifications 1.1 or 2.0.

## How to Install

Follow the **GETTING_STARTED.TXT** guide for step-by-step instructions on download, configuration and test of a new JMS Extender installation.

Once you have your JMS Extender up and running, you can find detailed information regarding JMS connector parameters and general configuration in the following paragraphs.

## JMS Connector Parameters

The JMS connector supports the following parameters:

- **install_dir** (tag): optional, if specified jars and classes for this JMS connector are searched inside “lib” and “classes” subdirectories of the specified install directory; path is relative to the configuration file;
- **disabled**: optional, if set to **true** this connector is skipped and will not work; may be useful when trying different configuration; if set to **false** or not specified, the connector will work regularly;
- **jndi_factory**: mandatory, specifies the initial context factory class for the JNDI lookup;
- **url_pkg_prefixes**: optional, some JMS brokers require this parameter to function properly, it specifies the URL package prefixes; see your broker documentation to know if it is required and which value it requires;
- **jms_url**: mandatory, specifies the JMS broker URL;
- **conn_factory_name**: mandatory, name of the connection factory to be used; this name is looked up through the JNDI;
- **connect_timeout_millis**: mandatory, the timeout the connector will wait before giving up the JMS connection; also the timeout for client requests when the JMS broker is unavailable (i.e. temporarily disconnected or not responding); must be specified in milliseconds;
- **retry_delay_millis**: mandatory, the delay the connector will wait after a connection failure before retrying a new connection; must be specified in milliseconds;
- **topic_session_sharing**: optional, when set to **false** it will disable the sharing of a common session between all topic simple subscriptions (i.e. not durable); if set to **true** or not specified the connector will share a single session for all topic simple subscriptions (see *Special Considerations*);
- **connection_pool_size**: optional, specifies the number of JMS connections the connection pool will have; if not specified the pool will have a size of 5; it may also be set to 0, and in that case there will be no connection pooling at all, i.e. each client will have its own dedicated

connection to the JMS broker (this may pose scalability problems, see *Special Considerations*);

- **client_id_prefix**: optional, specifies the prefix of client ID for pooled JMS connections, a progressive number will be added as a suffix (starting with 1); use this parameter to assign different client IDs to connection pools of different nodes in a cluster; if the pool size is just 1, no suffix will be added; moreover, if the JMS broker supports multiple connections with same client ID (the JMS Extender is able to autodiscover this feature) again no suffix will be added; finally, if it is not specified pooled connections will have no client ID at all;
- **principal** and **credential**: optional, they specify the credentials to be used when connecting to the JNDI initial context; if not specified, no credentials will be passed to the JNDI during initial context creation;
- **user_name** and **password**: optional, they specify the user name and password to be used when connecting to JMS; if not specified, no user name or password will be passed to the JMS broker during connection;
- **individual_acknowledge_value**: optional, specifies the numeric value for the non-standard "individual acknowledge" mode, supported by some JMS brokers (see below on how to find this value); when specified, it will enable two additional acknowledge modes on the JMS JS Client: *DUPS_OK* and *INDIVIDUAL_ACK* (see *Special Considerations*); when set to **auto** the JMS Extender will try to autodiscover if the mode is supported and with which value;
- **pre_acknowledge_value**: optional, specifies the numeric value for the non-standard "pre acknowledge" mode, supported by some JMS brokers (see below on how to find this value); when specified, it will provide a minor but significant performance advantage when using the *PRE_ACK* acknowledge mode on the JMS JS Client ; when set to **auto** the JMS Extender will try to autodiscover if the mode is supported and with which value;
- **thread_pool_max_size**: optional, specifies the maximum size the thread pool dedicated to JMS operations, including sending, receiving and acknowledging messages, creating and closing consumers and producers, committing, rolling back and recovering sessions, etc.; this thread pool is shared between all client sessions of this connector; if not specified the default size is 100;
- **basic_statistics_log_interval_millis**: the interval to log basic statistics like the number of consumers and producers, the current upstream and downstream message frequency, etc.; statistics are logged on the “JMSExtenderLogger.statistics” logger at INFO level; must be specified in milliseconds;
- **latency_statistics_log_interval_millis**: the interval to log statistics on the delay that occurs between sending a message downstream and receiving its acknowledge from the client (“acknowledge latency”); each report contains a simple chart with latency distribution and details up to the 90$^{th}$ percentile; statistics are logged on the “JMSExtenderLogger.statistics” logger at INFO level; must be specified in milliseconds.

Note: values for **individual_acknowledge_value** and **pre_acknowledge_value** parameters may be found by looking at the numerical value of the corresponding Java constant fields. E.g. HornetQ 2.4 has the following values in the *HornetQJMSConstants* class:

- `INDIVIDUAL_ACKNOWLEDGE = 101`
- `PRE_ACKNOWLEDGE = 100`

## Common Configuration Parameters

The section related to common configuration supports the following parameters:

- **hook**: optional, the fully qualified name of a Java class that will receive callbacks for different JMS Extender operations related to authentication and authorization of users and their requests (see *The JMS Extender Hook*);

## Classpath and JMS Broker Libraries

In most situations, configuring the JMS Extender is an activity limited to the “jms_connectors” subdirectory. Its structure is the following:

- */Lightstreamer_JMS_Extender/*
  - *jms_connectors/*
    - *jms_connectors_conf.xml*
    - *jms_connectors_log_conf.xml*
    - *lib/*
      - <common external libraries here>
    - <broker name>/
      - lib/
        - <broker-specific external libraries here>

If you connect to just one JMS broker, you may safely put its client libraries under “jms_connectors/lib” directory. If, on the contrary, you work with multiple JMS broker (or multiple versions of the same JMS broker), keeping client libraries separated may avoid conflicts of unpredictable outcome.

To do this, just create appropriate subdirectories under the “jms_connectors” directory, each with its own “lib” subdirectory” (as shown in the directory tree above), and specify the installation directory for each JMS connector in the “jms_connectors_conf.xml” file. E.g.:

```
<jms_connector name="HornetQ">

    <!-- Install directory -->
    <install_dir>HornetQ</install_dir>

    <!-- JNDI Factory Name -->
    <param name="jndi_factory">org.jnp.interfaces.NamingContextFactory</param>

    <!-- URL Package Prefixes (if needed) -->
    <param name="url_pkg_prefixes">org.jboss.naming:org.jnp.interfaces</param>

    <!-- JMS Broker Main URL -->
    <param name="jms_url">jnp://localhost:1100</param>

    <!-- Connection Factory Name -->
    <param name="conn_factory_name">ConnectionFactory</param>

    <!-- Connection Timeout -->
    <param name="connect_timeout_millis">5000</param>

    <!-- Reconnection Delay -->
    <param name="retry_delay_millis">2000</param>
</jms_connector>
```

This will make the JMS Extender look for jars and classes under “jms_connectors/HornetQ/lib” and “jms_connectors/HornetQ/classes” respectively.

Moreover, you may need to add under “jms_connectors/lib” (or the “lib” specific for you JMS connector) **any library that defines objects you are going to transfer via JMS object messages**. The JMS Extender will need their class definitions to serialize/deserialize them to/from JSON objects.

## JMS Extender Examples

Several examples of JMS Extender applications, with full source code, are provided on GitHub. The examples need a JMS broker to run with. You can choose whatever JMS broker you prefer. Examples provide extensive informations for the following brokers: HornetQ, TIBCO EMS, ActiveMQ, and JBossMQ.

The list of examples, together with links to corresponding GitHub projects, is available at the URL below:

⇒ **demos.lightstreamer.com/?p=jmsextender&t=client&a=javascriptclient**

# 4 The JMS JavaScript Client API

The JMS JavaScript Client provides, on the client side, a set of APIs consistent with the standard JMS APIs. Common objects like TopicConnection, QueueSession, ObjectMessage and so on are replicated and used consistently. Experienced JMS users will find themselves at home.

## JavaScript Libraries

The JMS JavaScript client library comes in four different flavors. The only difference between them is the way they should be used.

**lightstreamer-jms_globals.js**
Once the file is included on a page, a global reference to each included class is created on the window object. This is the easiest way to use the library, but as a drawback, it is also the easiest way to get a name collision in your application. If you go down this way, ensure that no other code/library on your page declares a global having the same name as the ones used in the Lightstreamer JMS js library.
NOTE: This option can only be used inside a browser page.

**lightstreamer-jms_globals_namespace.js**
The difference between this and the lightstreamer-jms_globals.js version is that one single global named Lightstreamer is created and a reference to each class is then attached to it (like Lightstreamer.Class). This version is much safer than the simple globals one as only one name is reserved by the Lightstreamer library, so that name collision will unlikely happen.
NOTE: This option can only be used inside a browser page

**lightstreamer-jms.js**
In this case, no globals are created and Lightstreamer classes must be accessed using proper calls to the AMD require method. Unfortunately, this approach will pollute the page with a number of define calls using three or four letter module names (all of the names using a common "ls" prefix). This is completely safe only if Lightstreamer is the only library using the AMD mechanism on the page. If you are including other libraries that use AMD, or your own code uses AMD to define its own modules, you may need to choose the next option.

**lightstreamer-jms_namespace.js**
This is the preferred way to use the Lightstreamer JavaScript library. Each class will be defined as Lightstreamer/Class so that the risk of having two defines mapped to the same name is close to zero.

Note that the JMS client library depends on the Lightstreamer JavaScript client: be sure to use the same flavor for both libraries or incompatibilities may arise.

## JMS Features

Since a discussion of the JMS API is out of scope, here follows a brief list of API differences:

- While the structure of the JMS API set is consistent, **only a subset of specific class and instance methods are present**. You should check for the availability of a specific API on the API reference docs before counting on it.
- A number of **type-specific APIs have been collapsed to a single API**, since JavaScript is a non-typed programming language; i.e. on the Message object there's just a setObjectProperty method, no setIntProperty, setLongProperty, setStringProperty, etc.
- **StreamMessage is not implemented**.

- **Connection error monitoring is not implemented**. The JMS JavaScript client library reconnects automatically to the JMS Extender in case of a client-side connection drop. Moreover, the JMS Extender itself reconnects automatically to the JMS broker in case of a server-side connection drop.
- Support for **synchronous receiving of messages** is supported only through *receiveNoWait*, due to the asynchronous nature of Javascript.
- Some operations that on JMS are synchronous here are **implemented asynchronously**; i.e. temporary queues and topics, connections, and most of JMS exceptions are received asynchronously.

Here is a list of specific JMS features supported by the JMS JS Client:

- **Producers** and **consumers** on any **topic or queue**.
- **Durable subscribers**.
- **Asynchronous message consuming** via message listeners.
- **Synchronous message consuming** via *receiveNoWait*.
- **Message selectors**.
- 5 different **acknowledge modes**.
- 4 different **message types** (text, object, map and bytes).
- **Session recovery**.
- **Transactions**.
- **Temporary queues and topics**.
- **Message correlation**.
- Connection **client IDs**.

⇒ Please refer to the API reference included in the SDK for JavaScript Clients for detailed documentation.

## Topic Subscription Example

Here is how a simple topic subscription is accomplished using JMS JavaScript Client:

```
ConnectionFactory.createConnection(
        "http://my.push.server:8080/", "HornetQ", null, null, {
        onConnectionCreated: function(conn) {
                conn.setExceptionListener({
                        onException: function(exception) {
                                // Handle exceptions here
                        }
                });

                var session= conn.createSession(false, "PRE_ACK");
                var topic= session.createTopic("stocksTopic");
                var consumer= session.createConsumer(topic, null);

                consumer.setMessageListener({
                        onMessage: function(message) {
                                // Handle messages here
                        }
                });

                conn.start();
        }
});
```

The only difference with the JMS equivalent is in the ConnectionFactory: here it requires the JMS Extender address and the connector identity, and the topic connection is returned asynchronously instead of synchronously.

## Queue Sending Example

Here is how sending a message through a queue is accomplished using JMS JS Client:

```
ConnectionFactory.createConnection(
        "http://my.push.server:8080/", "HornetQ", null, null, {
        onConnectionCreated: function(conn) {
                conn.setExceptionListener({
                        onException: function(exception) {
                                // Handle exceptions here
                        }
                });

                var session= conn.createSession(false, "AUTO_ACK");
                var queue= session.createQueue("stocksQueue");
                var producer= session.createProducer(queue, null);

                var msg= session.createTextMessage("some text");
                producer.send(msg);

                conn.start();
        }
});
```

As in the topic subscription example, the only difference with standard JMS code is in the creation of the Connection: here it requires the JMS Extender address and the connector identity, and the queue connection is returned asynchronously instead of synchronously.

## Dynamic Module Loading and require.js

The JMS JavaScript Client makes heavy use of dynamic module loading through require.js, as most modern JavaScript libraries do.

Most of the times, your code should only need the connection factory as a starting point, because in the JMS usage pattern all objects are obtained through creation from a parent object (with the exception of the factory, in fact). Thus, most of the times this simple *require* will do:

```
require(["ConnectionFactory"], function(ConnectionFactory) {
        ConnectionFactory.createConnection(url, connectorName, user, password, {
                onConnectionCreated: function(conn) {
                        // Your code here
                }
        });
});
```

## Full Examples

Several examples of JMS Extender applications, with full source code, are provided on GitHub. The list of examples, together with links to corresponding GitHub projects, is available at the URL below:

⇒ **demos.lightstreamer.com/?p=jmsextender&t=client&a=javascriptclient**

# 5 The JMS Extender Hook

The JMS Extender hook is a custom component that runs on the server side, inside the Lightstreamer Server process (it is plugged into the JMS Extender). The hook is a user defined Java class that can be used to intercept specific JMS Extender operations to apply authentication, authorization, and name mangling/decoration.

The hook must subclass the abstract class ***com.lightstreamer.jms_extender.hooks.JmsExtenderHook***, which is defined as follows:

```
public abstract class JmsExtenderHook {

        public void init(File configDir) throws Exception {}

        public boolean onConnectionRequest(
                        String connectionId,
                        String user,
                        String password,
                        Map clientContext,
                        String clientPrincipal) throws HookException {
                return true;
        }

        public void onConnectionClose(
                        String connectionId) {}

        public boolean onDedicatedBrokerConnectionRequest(
                        String connectionId,
                        String jmsConnectorName,
                        String clientId) throws HookException {
                return true;
        }

        public void onSessionOpen(
                        String connectionId,
                        String jmsConnectorName,
                        String sessionGuid) {}

        public void onSessionClose(
                        String connectionId,
                        String jmsConnectorName,
                        String sessionGuid) {}

        public boolean onMessageConsumerRequest(
                        String connectionId,
                        String jmsConnectorName,
                        String sessionGuid,
                        String destinationName,
                        boolean destinationIsTopic) throws HookException {
                return true;
        }

        public void onMessageConsumerClose(
                        String connectionId,
                        String jmsConnectorName,
                        String sessionGuid,
                        String destinationName,
                        boolean destinationIsTopic) {}
```

```
        public boolean onDurableSubscriptionRequest(
                        String connectionId,
                        String jmsConnectorName,
                        String clientId,
                        String sessionGuid,
                        String subscriptionName,
                        String topicName) throws HookException {
                return true;
        }

        public void onDurableSubscriptionClose(
                    String connectionId,
                    String jmsConnectorName,
                    String clientId,
                    String sessionGuid,
                    String subscriptionName,
                    String topicName) {}

        public boolean onMessageProducerRequest(
                        String connectionId,
                        String jmsConnectorName,
                        String sessionGuid,
                        String destinationName,
                        boolean destinationIsTopic) throws HookException {
                return true;
        }

        public void onMessageProducerClose(
                    String connectionId,
                    String jmsConnectorName,
                    String sessionGuid,
                    String destinationName,
                    boolean destinationIsTopic) {}

        public String onObjectMessagePayloadClassRequest(
                        String connectionId,
                        String jmsConnectorName,
                        String sessionGuid,
                        String destinationName,
                        boolean destinationIsTopic,
                        String classFullyQualifiedName) throws HookException {
                return classFullyQualifiedName;
        }

        public Object onObjectMessagePayloadRequest(
                        String connectionId,
                        String jmsConnectorName,
                        String sessionGuid,
                        String destinationName,
                        boolean destinationIsTopic,
                        Object payload) throws HookException {
                return payload;
        }

        public String getDedicatedBrokerConnectionName(
                        String connectionId,
                        String jmsConnectorName,
                        String clientId) {
                return clientId;
        }

        public String getDurableSubscriptionName(
                        String connectionId,
                        String jmsConnectorName,
                        String clientId,
                        String sessionGuid,
                        String subscriptionName,
                        String topicName) {
                return subscriptionName;
        }
}
```

This abstract class and its dependencies are defined in **ls-jms-hook-interface.jar**. Include this jar file in your classpath to develop a custom hook. Once developed, the hook fully qualified class name must be specified in the **hook** parameter of the JMS Extender common configuration section (see *Common Configuration Parameters*), to be instantiated and used.

Hook methods may be distinguished in two different kinds:

- ***Request methods***, which are expected to return a value or exception, and
- ***Notification methods***, which are not expected to return values or exceptions.

For each operation that can be authorized, the hook provides a corresponding request method for operation authorization and a notification for operation termination. Such operations are:

- user connection,
- dedicated connection to the JMS broker (see also *Use of Durable Subscriptions and Scalability Constraints*),
- creation of a message consumer,
- creation of a durable subscriber,
- creation of a message producer.

An exception applies to sessions, which are notified for both creation and termination (i.e. the session opening method is not a request, this operation can't be denied).

The hook provides also request methods related to payload management of object messages. This kind of messages contains a JSON object that must be deserialized and mapped to a corresponding Java object, whose fully-qualified class name has been specified. These request methods provide an opportunity to verify the fully-qualified class name (before deserialization and mapping) and the resulting payload (after deserialization and mapping).

Finally, the hook provides an opportunity to mangle and/or decorate client IDs of dedicated connections and names of durable subscription (again, *see Use of Durable Subscriptions and Scalability Constraints*).

A brief description of each API method follows.

- **init**: The hook object is dynamically instantiated and requires a default constructor; after that, the init method is called and is passed a pointer to the configuration directory in the *configDir* parameter, where the hook can find any specific initialization file;
    - if the *init* method completes successfully, the hook will be used;
    - if the *init* method terminates with an exception, the JMS Extender will abort and cause the JMS Extender to fail the initialization too.

- **onConnectionRequest**: Called to request authentication and authorization when a user connects to the JMS Extender; it is passed a connection identifier, the user name, the specified password, a context map containing also the HTTP headers and, if available, the client principal;
    - return *true* if the user is authenticated and the connection can proceed, or
    - return *false* or an exception if the connection must be blocked.

- **onConnectionClose**: Called to notify of a user disconnection from the JMS Extender; it is passed the connection identifier.

- **onDedicatedBrokerConnectionRequest**: Called to request authorization when a user tries to open a dedicated connection to the JMS broker; this is a consequence of the user setting the client ID on the connection, and usually it is a prerequisite for a successive durable

subscription (again, *see* *Use of Durable Subscriptions and Scalability Constraints*); it is passed the connection identifier, the connector name and the client ID set by the user:
    - return *true* if the dedicated connection can proceed, or
    - return *false* or an exception if the dedicated connection must be blocked.

- **onSessionOpen**: Called to notify that the user created a new session; it is passed the connection identifier, the connector name and the session GUID (unique identifier).

- **onSessionClose**: Called to notify that the user closed a previously created session; it is passed the connection identifier, the connector name and the session GUID (unique identifier).

- **onMessageConsumerRequest**: Called to request authorization when a user creates a message consumer; it is passed the connection identifier, the connector name, the session GUID, the destination name and a flag that tells if the destination is a topic:
    - return *true* if the consumer can be created, or
    - return *false* or an exception if the consumer's creation must be blocked.

- **onMessageConsumerClose**: Called to notify when a user closes a message consumer; it is passed the connection identifier, the connector name, the session GUID, the destination name and a flag that tells if the destination is a topic.

- **onDurableSubscriptionRequest**: Called to request authorization when a user creates a durable subscription; it is passed the connection identifier, the connector name, the session GUID, the subscription name and the topic name:
    - return *true* if the subscription can be created, or
    - return *false* or an exception if the subscription's creation must be blocked.

- **onDurableSubscriptionClose**: Called to notify when a user closes a durable subscription; it is passed the connection identifier, the connector name, the session GUID, the subscription name and the topic name.

- **onMessageProducerRequest**: Called to request authorization when a user creates a message producer; it is passed the connection identifier, the connector name, the session GUID, the destination name and a flag that tells if the destination is a topic:
    - return *true* if the producer can be created, or
    - return *false* or an exception if the producer's creation must be blocked.

- **onMessageProducerClose**: Called to notify when a user closes a message producer; it is passed the connection identifier, the connector name, the session GUID, the destination name and a flag that tells if the destination is a topic.

- **onObjectMessagePayloadClassRequest**: Called to request authorization of the mapping of a client-side object message payload on a specific Java class; it is passed the connection identifier, the connector name, the session GUID, the destination name, a flag that tells if the destination is a topic and the requested fully-qualified class name:
    - return the accepted fully-qualified class name (may differ than the passed one).

- **onObjectMessagePayloadRequest**: Called to request authorization to send the client-side object message payload once it has been mapped on a Java object; it is passed the connection identifier, the connector name, the session GUID, the destination name, a flag that tells if the destination is a topic and the mapped payload:
    - return the accepted payload (may differ than the passed one).

- **getDedicatedBrokerConnectionName**: Called to give an opportunity to mangle or decorate the client ID of a dedicated connection to guarantee per-user uniqueness (again, see *Use of durable subscriptions and scalability constraints*); it is passed the connection identifier, the connector name and the client ID; note that this method may be called more than once for the same dedicated connection, returned values must be consistent:
  - return the mangled/decorated client ID, or
  - simply return the passed client ID if no mangling or decoration is necessary.

- **getDurableSubscriptionName**: Called to give an opportunity to mangle or decorate the subscription name of a durable subscription to guarantee per-user uniqueness (again, see *Use of durable subscriptions and scalability constraints*); it is passed the connection identifier, the connector name, the client ID, the session GUID, the subscription name and the topic name; note that this method may be called more than once for the same durable subscription, returned values must be consistent:
  - return the mangled/decorated subscription name, or
  - simply return the passed subscription name if no mangling or decoration is necessary.

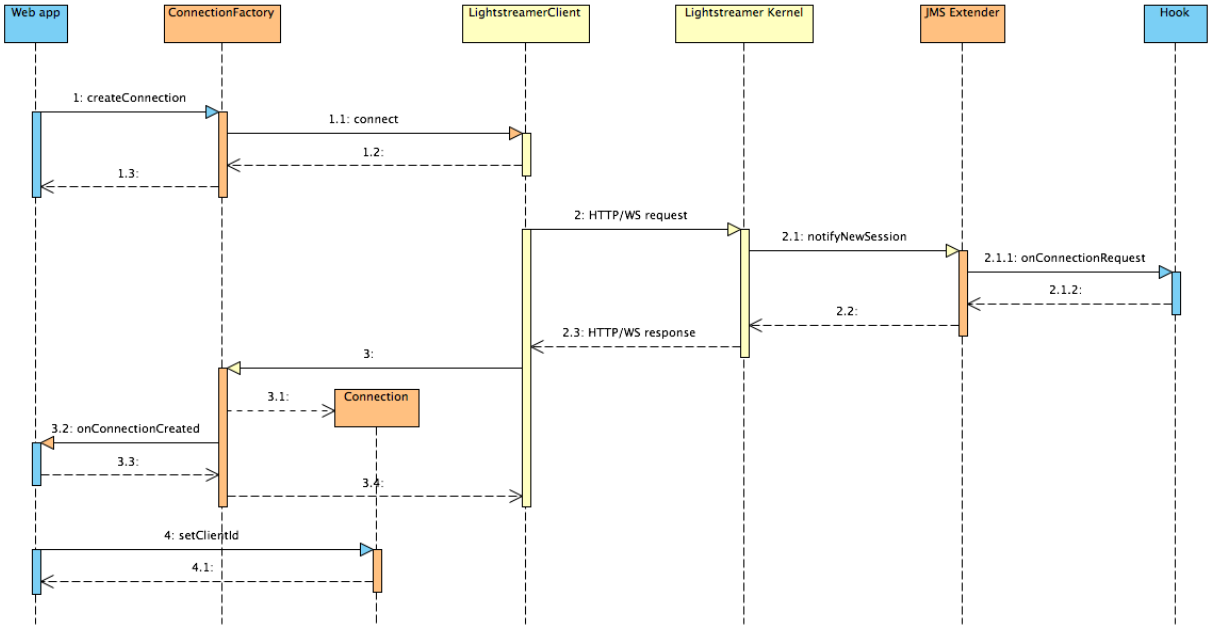
⇒ Please refer to the API reference included in the SDK for JMS Extender Hook for detailed documentation.

Examples of JMS Extender Hook implementations, with full source code, are provided on GitHub. The list of examples, together with links to corresponding GitHub projects, is available at the URL below:

⇒ **demos.lightstreamer.com/?p=jmsextender&t=hook&a=javahook**

## Hook Call Sequence on User Connection

The following diagram shows the sequence of events on both the client side and on the extender side during the creation of a connection:



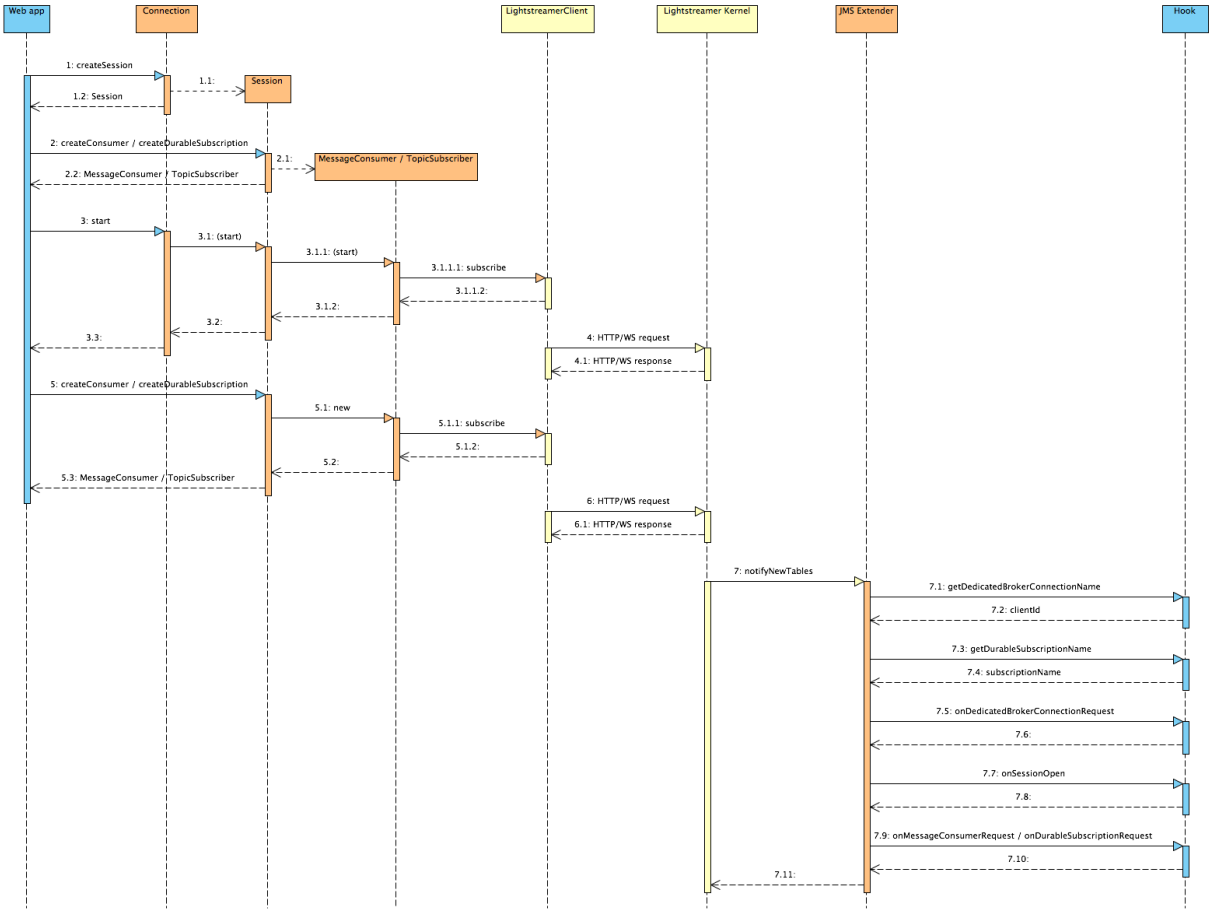

Please note the following details:

- during the connection sequence only the hook's *onConnectionRequest* is called;
- setting the client ID on the received connection has no effect (yet) on the hook.

Due to the asynchronous operation of JavaScript and in particular of the Lightstreamer JavaScript client, most of the hook methods are called when JMS objects are actually used. The two following sequence diagrams will clarify the concept.

# Hook Call Sequence for Message Consuming

The following diagram shows the sequence of events on both the client side and the extender side when a message consumer is created and its connection started:

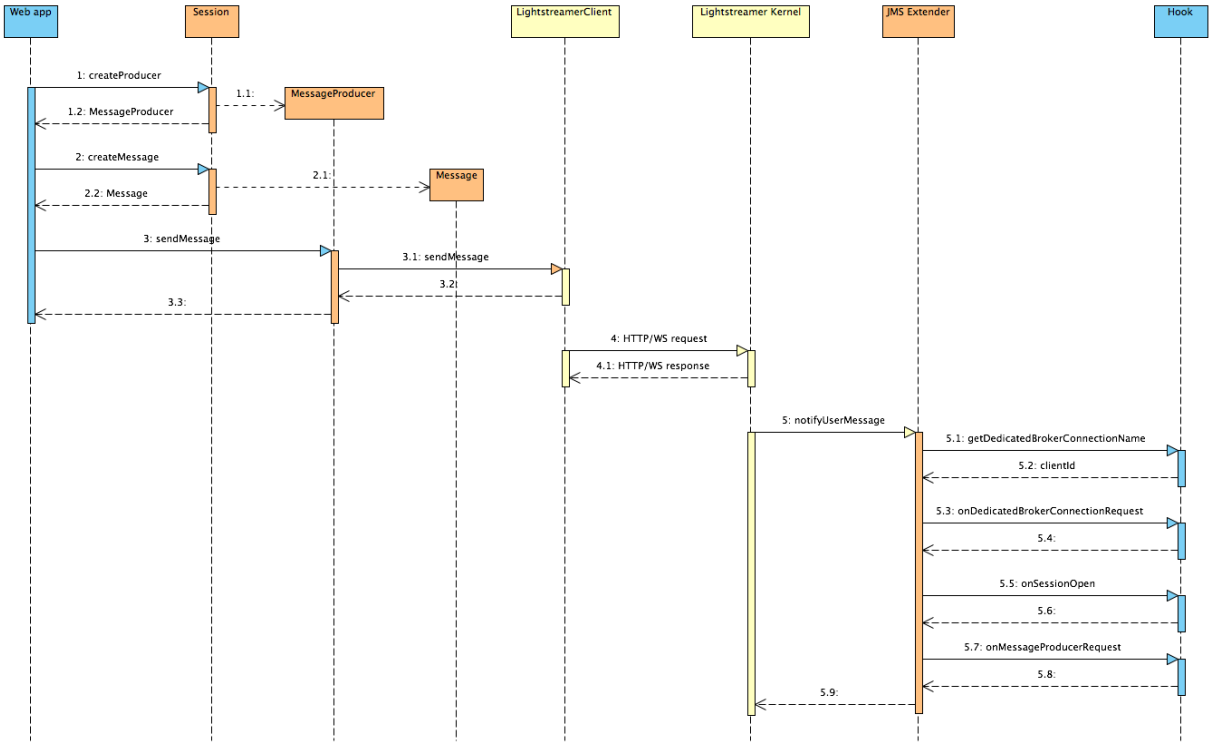



As per JMS specification, creating a message consumer is not enough to start receiving messages: its connection must be started too. When this happens on the client side, the message consumer is actually activated and on the extender side the hook's methods are called accordingly. Message consumers created subsequently will be activated immediately.

Of course, the activation still acts asynchronously on the extender, which will call the hook only at a later stage. If the hook should not authorize any of the request objects, an asynchronous exception would be delivered to the client.

## Hook Call Sequence for Message Producing

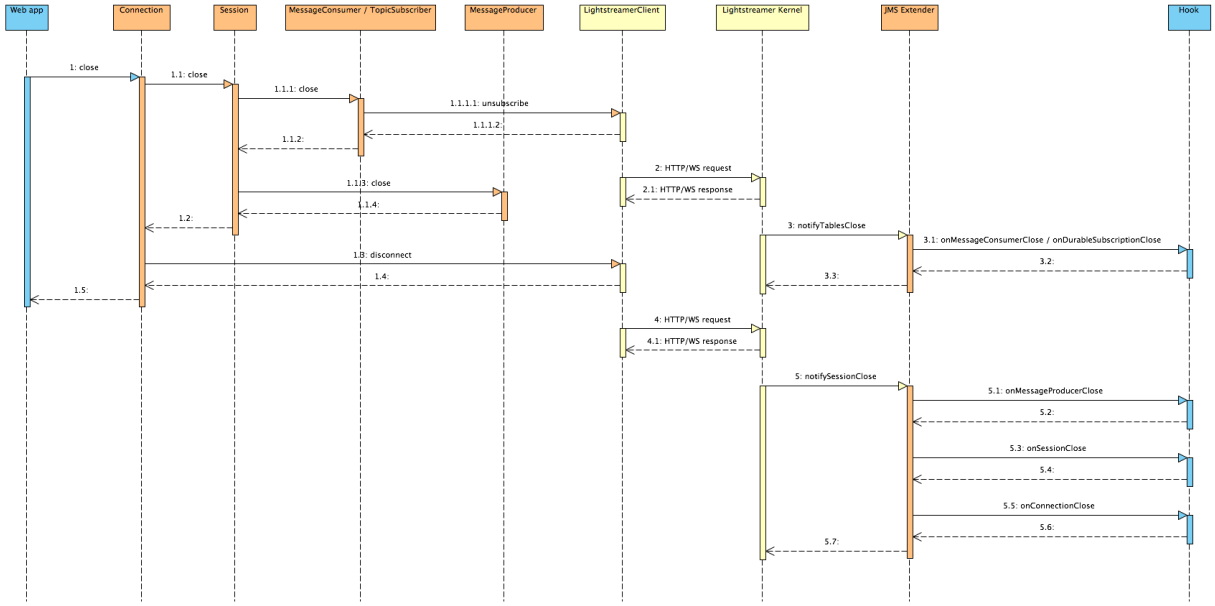
The following diagram shows the sequence of events on both the client side and the extender side when a message producer is created and a message sent:




Similarly to the previous diagram, creating a message producer on the client side does not deliver any notification on the extender side. Only when a message is actually sent the extender is made aware of the operation and calls the hook's methods accordingly.

# Hook Call Sequence for User Disconnection

The following diagram shows the sequence of events on both the client side and on the extender side during a disconnection:




Please note the following details:

- closing a message consumer does deliver the corresponding notification, although asynchronously as in previous cases;
- on the other hand, notifications of a message producer or session closure are delivered only when their connection is closed, and even in this case notifications are asynchronous.

# 6 Special Considerations

The JMS Extender provides an encapsulation mechanism for common JMS semantics, but in fact it's not a JMS broker itself. Due to this, some considerations apply for the following topics:

- use of a shared session with simple topic subscriptions
- use of durable subscriptions and scalability constraints
- acknowledge modes DUPS_OK and INDIVIDUAL_ACK

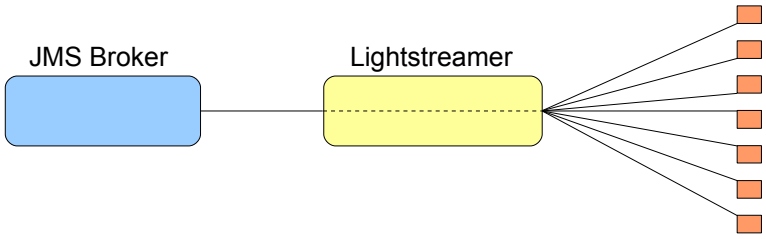
## Use of a Shared Session With Simple Topic Subscriptions

In its default configuration, the JMS Extender is able to do what is called *connection offloading*, i.e. delegate the fanout of topic messages to Lightstreamer instead of the JMS broker. Lightstreamer provides a much improved scalability in this scenario, compared to common JMS broker, since it has been designed to handle a much larger number of connections.

To take advantage of connection offloading, some conditions apply:

1. the special flag **topic_session_sharing** on the connector must be set to **true** or not set at all (by default it is enabled);
2. the topic must be subscribed with a simple subscription; durable subscriptions can't take advantage of connection offloading.

Under these conditions, the JMS Extender connector will set up a shared session on which all simple topic subscriptions will be done. This will ensure that just one session is established between the JMS Extender and the JMS broker for topic message transport, greatly reducing load on the broker. At the same time, it will be up to Lightstreamer to deliver the topic messages to all subscribers.
See the picture below:

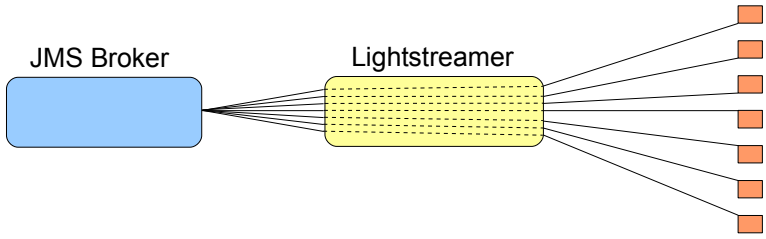



JMS Extender with connection offloading

Moreover, under these conditions, the topic subscription will be done using the special *PRE_ACK* acknowledge mode, whatever acknowledge mode the client specifies (recall that acknowledge mode is session-wide, this override applies only to topic subscriptions). This mode pre-acknowledges messages before they leave the JMS Extender, ensuring better performances and lower overhead. Anyway, consider that JMS brokers are free to dismiss any unacknowledged message on a topic subscription by JMS specifications (see *Creating Robust JMS Applications* on Oracle's web site). Thus, this forced acknowledge mode does not change significantly the semantics from a pure JMS connection.

On the other hand, disabling connection offloading by setting **topic_session_sharing** to **false** will mean the fanout of topic messages will be provided by the JMS broker, and each and every topic subscription will rely on different JMS sessions.

See the picture below:




JMS Extender without connection offloading

In case of hundreds of thousands of users this may put the JMS broker under excessive load and bring it to a halt. If you think this configuration is better suited for your needs, we invite you to experiment a bit before turning it on in a production environment.

## Use of Durable Subscriptions and Scalability Constraints

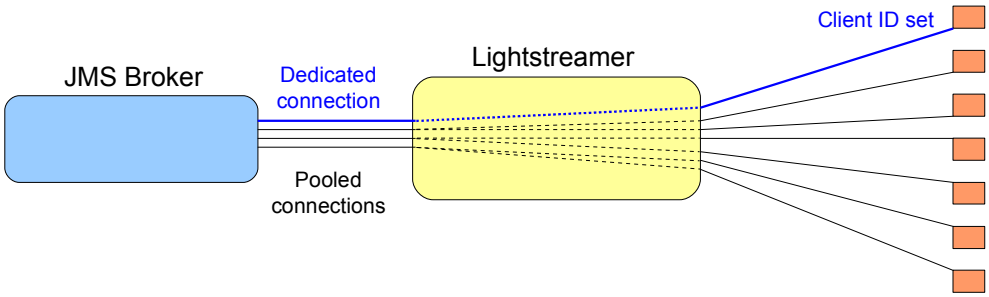
Durable subscriptions present a particularly challenging feature to provide, in a context like that of the JMS Extender. Since each durable subscription is identified by a pair of mandatory informations:

- the connection's client ID, and
- the subscription's name,

there is a problem associating a client ID with a pooled connection that may be chosen randomly and that is shared between different clients.

The JMS Extender chooses between two different strategies:

1. if the JMS broker **does not support** multiple connections with the same client ID, each time the client specifies a client ID a dedicated connection is created, and it is assigned the requested client ID; i.e. no use of the connection pool is made:




JMS Extender with a dedicated connection

2. if the JMS broker **supports** multiple connections with the same client ID, a randomly chosen connection from the pool is assigned to the client (since all connections will have the same configured client ID), and the subscription name is modified so that it contains both the requested client ID and the requested subscription name (to maintain its uniqueness).

Connections for which the client did not set a client ID (i.e. connections not used for durable subscriptions) remain pooled in both cases. Please note that some JMS brokers allow the creation of durable subscriptions without specifying a client ID (i.e. with client ID not set), but this is **not** supported by the JMS Extender as of today.

In case 1, a scalability problem arises. The JMS Extender relies on Lightstreamer, which easily supports hundreds of thousands of connections, but if each connection uses a durable subscription

with its own client ID then there will be a corresponding number of connections between the JMS Extender and the broker. The latter may not scale so well with these numbers, even with low traffic.

In case 2, everything will scale well, but keep in mind that the JMS broker will not see the durable subscriptions with the same name that is specified on the client side. In case of auditing, for instance, this should be taken into account.

The modified name is a juxtaposition of the client ID and the subscription name:

```
<requested client ID>-<requested subscription name>
```

For instance, if a client sets the client ID to *"customer1"* and requests a subscription with name *"portfolio"*, the durable subscription will be created on the JMS broker with the name:

- *"customer1-portfolio"*.

Recall that in case 2 the actual client ID is what has been set in the connector configuration's **client_id_prefix** parameter (see *Configuration and Deployment*).

Besides these details, durable subscriptions work out of the box. But to give even more flexibility to the system integration, you may use the hook, that provides an opportunity for name mangling and/or decoration for both client IDs and subscription names. This hook is configured through the parameter **hook** of the common configuration section (see *Common Configuration Parameters*) and provides two specific methods for this purpose:

- *getDedicatedBrokerConnectionName* and
- *getDurableSubscriptionName* (see *The JMS Extender Hook*).

### Precautions About Client ID Uniqueness

You have to take particular precautions when your JMS broker does not support multiple connections with the same client ID (i.e. case 1), as there are a number of situations where client ID uniqueness may become a problem, e.g.:

- two clients may ask for the same client ID on the same server;
- two clients may ask for the same client ID on two different nodes of a cluster;
- a client may ask for a client ID actually in use by a connection pool;
- a badly configured cluster node may try populate its connection pool with client IDs in use by another node;
- etc.

In all these cases the JMS broker won't open the connection (as per JMS specifications), causing service denial to the client or even to the node.

The hook has been designed to give you an opportunity to prevent some of these problems: it gives you a chance to add something user-specific or node-specific to the client ID and/or to the subscription name at the moment they are asked, so that you can maintain their uniqueness. Design your system keeping in mind these scenarios, as the JMS Extender cannot provide different service requirements than those of the underlying JMS broker.

## Acknowledge Modes

The JMS Extender supports 5 acknowledge modes:

- *PRE_ACK*: messages are acknowledged before leaving the extender, or even before leaving the broker if the **pre_acknowledge_value** parameter has been specified in the connector configuration (see *Configuration and Deployment*);
- *AUTO_ACK*: messages are acknowledged automatically when the client has finished processing them;

- *CLIENT_ACK*: messages are acknowledged manually by the client by calling the *acknowledge* method on the *Message* object; as per JMS specifications, acknowledgement is **cumulative**: all messages received from the session tare acknowledged with a single call;
- *DUPS_OK*: messages are acknowledged automatically at lazy time, thus there is a chance you may receive a message more than one time if a session recovery is started;
- *INDIVIDUAL_ACK*: messages are acknowledged manually by the client by calling the *acknowledge* method on the Message object; this mode is not part of original JMS specifications but is nevertheless supported by many modern JMS brokers; with this mode acknowledgement is **not cumulative**: you may acknowledge a single message and be sure that every other message will be redelivered upon a session recovery.

While first 3 modes are always available, modes *DUPS_OK* and *INDIVIDUAL_ACK* rely on the availability of INDIVIDUAL_ACK mode on the JMS broker. If the broker does not support it, these modes can't be provided.

The JMS Extender can try to discover automatically if *INDIVIDUAL_ACK* is available, but JMS broker implementors have shown a good amount of fantasy when it is time to choose this mode name. Sometimes it is called "individual", others it is called "explicit" or something else. If you know for sure your broker supports this mode, set its numeric value in the **individual_acknowledge_value** parameter on the connector (see *Configuration and Deployment*). This will make sure *DUPS_OK* and *INDIVIDUAL_ACK* will be available.

## Clustering

*For general informations on the configuration of a Lightstreamer Server cluster, please refer to the included Lightstreamer Clustering document. Details on how to configure the load balancer, how to obtain session stickiness or how to leverage wildcard HTTPS certificates, can be found there. The following information apply specifically to the JMS Extender.*

For common use of **topics and queues**, configuring multiple JMS Extenders as part of a cluster presents no challenges. Clients consuming messages from the same queue on different nodes will simply instruct the JMS Extender to create different queue receivers, and messages will round-robin between consumer as they would do with direct JMS connections. Correspondingly, clients subscribing to the same topic will instruct the JMS Extender to create different topic subscribers, and messages will fan out to subscribers as they would do with direct JMS connections.

For these use cases, the same configuration may be cloned on each node, taking care only of setting the appropriate user name and password (and/or principal and credential) if they must be different between nodes. The *client_id_prefix* parameter, which is related only to durable subscriptions, should be commented out to avoid client ID conflicts.

**Durable subscriptions**, on the other hand, present some challenges. Most of JMS brokers require that clients provide a unique client ID to reconnect a durable subscription appropriately. The client ID may either be provided by JMS Extender clients autonomously or by the JMS Extender itself by its configuration (via the *client_id_prefix* parameter). In the second case, when a client roams between different nodes, they may get different client IDs and fail to reconnect their durable subscriptions.

Moreover, depending on a number of factors, each client may be assigned a pooled connection or a dedicated connection. In the second case, scalability constraints may arise. For more information regarding connection pooling and its relationship with client IDs and durable subscriptions, please check out the paragraph Use of Durable Subscription and Scalability Constraints. You are advised to read it before proceeding.

In order to understand how to configure your cluster and what kind of support you may have for durable subscriptions, you should ask yourself the following questions:

1. Does my JMS broker support multiple connections with the same client ID?

    - If in doubt, set the logging category *JMSExtenderLogger.autodiscovery* at *INFO* level, start the JMS Extender and check the log for the following string:

    ```
    Autodiscovery for Support for Multiple Connections with Same Client ID succeeded
    ```

2. Do my clients always provide a unique client ID via the *setClientID* API when connecting to the JMS Extender?

3. Does my hook implement the *getDedicatedBrokerConnectionName* method and/or the *getDurableSubscriptionName* method to ensure uniqueness of the client ID/subscription name pair?

With answers at hand, find your entry in the table below:

<table>
<tr><th>Q1</th><th>Q2</th><th>Q3</th><th>Support for Durable Subscriptions</th><th>client_id_prefix</th></tr>
<tr><td>No</td><td>No</td><td>No</td><td rowspan="2">Not supported: the JMS Extender will assign a pooled connection with an unpredictable client ID.</td><td rowspan="4">Configure with a different value for each node.</td></tr>
<tr><td>No</td><td>No</td><td>Yes</td></tr>
<tr><td>No</td><td>Yes</td><td>No</td><td rowspan="2">Supported with scalability constraints: the JMS Extender will provide a dedicated connection to each client. 10.000 clients will require 10.000 connections to the JMS broker (i.e. not only to the JMS Extender)</td></tr>
<tr><td>No</td><td>Yes</td><td>Yes</td></tr>
<tr><td>Yes</td><td>No</td><td>No</td><td>Not supported: the JMS Extender will assign a pooled connection with a common client ID. There is no way to match a durable subscription with the client that created it.</td><td rowspan="4">Configure with the same value across all nodes.</td></tr>
<tr><td>Yes</td><td>No</td><td>Yes</td><td>Supported: the JMS Extender will assign a pooled connection with a common client ID, but the hook ensures each durable subscription has a unique subscription name.</td></tr>
<tr><td>Yes</td><td>Yes</td><td>No</td><td rowspan="2">Supported: the JMS Extender will assign a pooled connection with a common client ID, but the additional client ID provided by the client is juxtaposed to the subscription name to build a unique subscription name.</td></tr>
<tr><td>Yes</td><td>Yes</td><td>Yes</td></tr>
</table>

Note that each condition applies to a different component of the system: the JMS broker, the client, the the hook. If the outcome is not what you expected, you can change the conditions by intervening in the appropriate component, e.g. adding unique client ID generation to the client's code or implementing the appropriate callbacks in the hook.

# 7 JMS Broker Examples

The JMS Extender has been tested with most mainstream JMS brokers. Some advanced features may or may not be available, based on the version of each broker. The table below shows some examples of tested versions and resulting features.

Please consider that this is not an exhaustive compatibility list. In other words, older and newer versions of each broker will probably work seamlessly with the JMS Extender. But versions might impact on available features.

| Broker | Base ack modes | Adv. ack modes | Durable subscriptions | Transactions | Sync reading | Messages types | Temp. topics and queues |
|---|---|---|---|---|---|---|---|
| **HornetQ 2.2.14** | ✓ | ✗ | ✓ | ✓ | ✓ | All | ✓ |
| **HornetQ 2.4** | ✓ | ✓[1] | ✓ | ✓ | ✓ | All | ✓ |
| **Apache ActiveMQ 5.6** | ✓ | ✓[2] | ✓ | ✓ | ✓ | All | ✓ |
| **JBossMQ** | ✓ | ✗ | ✓ | ✓ | ✓ | All | ✓ |
| **TIBCO EMS 7.0** | ✓ | ✓ | ✓ | ✓ | ✓ | All | ✓ |
| **IBM WebSphere MQ 7.5** | ✓[3] | ✗ | ✓ | ✓ | ✓ | All | ✓ |
| **Oracle WebLogic 12c** | ✓ | ✗ | ✓ | ✓ | ✓ | All | ✗[4] |
| **Apache Qpid 0.22** | ✓ | ✗ | ✓ | ✓ | ✓ | All | ✓ |

Advanced acknowledge modes are *DUPS_OK* and *INDIVIDUAL_ACK*, which depend on the native support of *INDIVIDUAL_ACK* mode on the JMS broker (see *Acknowledge Modes*).

The following notes apply:

1. HornetQ, as of version 2.3, when using advanced acknowledge modes fails to report the *redelivered* flag on recovered messages;
2. ActiveMQ, as of version 5.6, does not support the use of advanced acknowledge modes with durable subscriptions, although they can be safely used with common queues;
3. WebSphere MQ, as of version 7.5, does not support the use of synchronous operations on a session which has already been used for asynchronous operations; for this reason, topic session sharing must be disabled (see *Use of a Shared Session With Simple Topic Subscriptions*);
4. WebLogic, as of version 12c, is not able to properly recreate a destination from its own name (e.g.: *session.createQueue(queue.getQueueName())* returns a different queue or fails); for this reason, temporary topics and queues cannot be used with JMS Extender, as are some other operations that require this mechanism (e.g.: using *message.getJMSReplyTo()* to get the destination for a reply); operations where the destination name is set on the client are exhempt from this problem.