Introduction to Apache ActiveMQ

07/06 Version 1.2

Introduction to Apache ActiveMQ

This document is a brief introduction to Apache ActiveMQ in LogicBlaze FUSE. Apache ActiveMQ is a fast open source Java Message Service JMS 1.1 provider and Message Fabric, supporting clustering, peer networks, discovery, TCP, SSL, multi-cast, persistence, and XA. Apache ActiveMQ integrates seamlessly into J2EE 1.4 containers, light weight containers, and any Java application. Apache ActiveMQ is released under the Apache 2.0 License.

1. ActiveMQ in LogicBlaze FUSE

Inside LogicBlaze FUSE we use Apache ActiveMQ to provide high performance, reliable, JMS-based messaging, plus inter-operate with all standard JMS providers.

Apache ActiveMQ comes preinstalled with the LogicBlaze FUSE distribution. To see a sample application using Apache ActiveMQ, please read the *Loan Broker Tutorial*.

2. Configuration

The LogicBlaze FUSE distribution includes Apache ActiveMQ preconfigured for you. However, if you need to customize the configuration for Apache ActiveMQ, edit this file:

```
[fuse_dir]\components\activemq-xbean.xml
```

where `[fuse_dir]` is the directory in which LogicBlaze FUSE was installed.

After editing the `activemq-xbean.xml` please restart FUSE to make your changes take effect.

LogicBlaze FUSE supports an XML deployment descriptor for configuring the Apache ActiveMQ Message Broker.

There are many things which can be configured, such as:

- transport connectors which consist of transport channels and wire formats
- network connectors using network channels or discovery
- discovery agents
- persistence providers & locations
- custom message containers (such as last image caching, etc.)

XBean is used to perform the XML configuration. If you are already using XBean then you can just mix and match your Spring/XBean XML configuration with ActiveMQ configuration.

See the *XML Reference* at http://incubator.apache.org/activemq/ for details of the XML configuration.

ActiveMQ has many strategy pattern plugins for transports, wire formats, persistence, and many other things, therefore, we wanted to leave the configuration format open so that you the developer can configure and extend ActiveMQ in any direction you wish.

We use the Spring XML configuration file format, which allows any beans or POJOs to be wired together and configured. However, Spring's XML can be verbose, so we have implemented an ActiveMQ extension to the Spring XML which knows about the common, standard ActiveMQ things you're likely to do, for example, tags such as `connector`, `wireFormat`, `serverTransport`, `persistence`. At any time you can fall back to the normal Spring way of doing things, with tags such as `bean`, `property`, etc.

This means there is an ActiveMQ DTD which extends the Spring DTD. Therefore, DTD-aware XML editors can help you edit configuration files without you having to look at Java code to remember property and class names, etc.

3. Monitoring a Broker

To monitor a message broker:

1. Run a broker setting the broker property `useJmx` to true.

   - For an XBean configuration:

     ```
     <broker useJmx="true"brokerName="BROKER_1">
     ...
        <broker>
     ```

   - For url configuration

     ```
     broker:(tcp://localhost:61616)?useJmx=true
     ```

   In the `[fuse_dir]\components\activemq\activemq-xbean.xml` file the `useJmx` property is set to true. You could also set this property in a custom configuration file.

2. Run a JMX console, for example, `JConsole`. `JConsole` is a JMX console included in the `JDK <JAVA_HOME>/bin/jconsole.exe`.

3. Connect to the given JMX URL:

   ```
   jconsole service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi
   ```

4. Additional Resources

For more information on Apache ActiveMQ: http://incubator.apache.org/activemq/.

For more information on the Java Message Service (JMS): http://java.sun.com/products/jms/.