

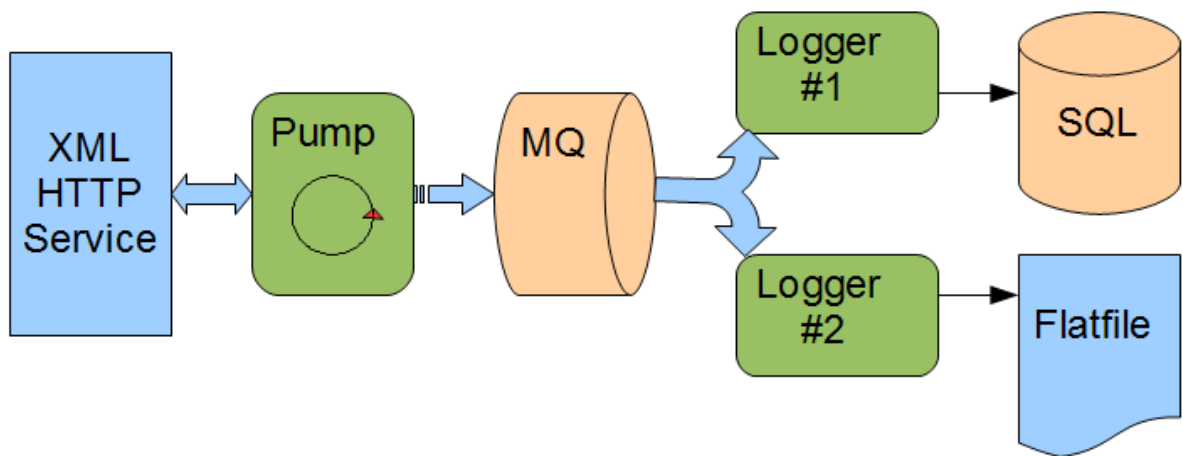
An ESB Implementation for the NextBus Live Data Feed

James R. Doyle

Sept 2012 – Rev 1.1

rockymtnmagic@gmail.com

<http://sourceforge.net/projects/nextbusapi>



Abstract

The NextBus Service provides a near real-time feed of transit service bus traffic in several major Metropolitan areas. This data feed is provided over a classic XML HTTP web service to support a wide number of programming language preferences and platforms for application developers. Following the release of a Java adapter for this XML web service, an Enterprise Service Bus (ESB) implementation was developed using the adapter to demonstrate an Event-Driven, Asynchronous programming model as an alternative approach for building mass-transit applications.

The ESB is provided as a GPL open-source project with a set of pre-built executable binaries capable of customization through external properties files. The implementation is based on Spring Integration and Apache ActiveMQ. The software components consist of a two daemons: a Pumper that feeds a durable Topic and a flexible Logger which acts as a subscriber. Several loggers can be run at once, each capable of tuning into a narrow band of message traffic and feeding the output to the shell console, filesystem, or a database. This simple end-to-end system provides a demonstration case that can be quickly deployed for learning and further used as a starting point for more ambitious projects. A discussion of potential Production Issues is included which can serve as a hands-on planning exercise to see just how ready this code may (or may not) be for going live.

Table of Contents

Abstract.....	1
Overview.....	2
Source Code Availability.....	2
Prebuilt Binaries.....	2
Design.....	3
Packaging Executable Components with Maven Shade.....	3
Message Pump Design.....	4
Logger Design.....	6
Supporting Non-Java (i.e. PHP, Ruby, C#, JavaScript) Clients	7
Setup and Installation.....	7
Setup an Apache ActiveMQ Server.....	7
Setup an Apache Derby DB.....	8
Download Pumper, Edit Properties Files and Start Daemon.....	9
Download Logger Daemon, Edit Properties File and Start.....	10
Next Steps: JDBC and Filesystem Loggers; explore the STOMP protocol.....	10
Further Topics for Consideration and Exploration.....	11
Understanding Performance Tuning Parameters.....	11
Message Selective Subscription with GPS Coordinates.....	12
Exploring Production Readiness and Reliability Issues.....	12
Porting to Mule, Camel, RabbitMQ, JBoss, MySQL, Postgres, etc?.....	13
What about Two Phase Commit in the Logger (JMS to JDBC) ?.....	13
What about security?.....	14
References.....	15
Legal Disclosures.....	15

Overview

The ESB Project, as well as NextBus XML API Adapter, are hosted on SourceForge. There is a discussion forum and bug tracking database there as well. Please learn more at:

<http://sourceforge.net/projects/nextbusapi/>

Source Code Availability

The Source Code for the XML Adapter, the Pump Daemon and the Logging Daemon is provided under the GNU Public License (GPL). The SVN Source repository for this code is located at:

`svn://svn.code.sf.net/p/nextbusapi/code`

Prebuilt Binaries

Binaries are provided for people who want the “fast-track” experience of getting the ESB up and running in under an hour. Built releases of executable JARs can be downloaded here:

<http://sourceforge.net/projects/nextbusapi/files/releases/>

Design

The ESB uses a JMS Durable Topic to support multiple subscribers against a stream of near-real-time updates on Vehicle Location and Bus Predictions. A Java Adapter to the NextBus XML feed provides native Java Domain classes mapped against the platform-neutral XML wire representation. JMS supports an *ObjectMessage* type which facilitates the transport of *Serialized POJOs* through the ESB. Since we can anticipate downstream Java clients, shipping clean domain objects through the message bus provides a type-safe and easy way to support client applications. Further, since the XML adapter is responsible for XML to Java class translation, downstream applications are less tightly coupled to the XML wire format which is always likely to change in the future. For non-Java applications that participate in the ESB, *JMS Header properties* are set by the pumper to mirror the key domain attributes that are present in the POJO. Since the header attributes are also eligible for message selectors, non-Java developers get a “free ride” without having to be forced to deal with a Java POJO.

Real-time feeds can have special properties over other types of Messaging based integration. New information arrives that replaces old information from the recent past. We use the JMS *Time-to-Live* (TTL) to expel stale and unconsumed messages from the ESB automatically. Listeners joining a topic can expect to not receive any traffic from the past that exceeds whatever “lateness” threshold is needed.

The Pump and Logging Daemon demonstrate many of the key out-of-box components from the Spring Integration toolkit. These include:

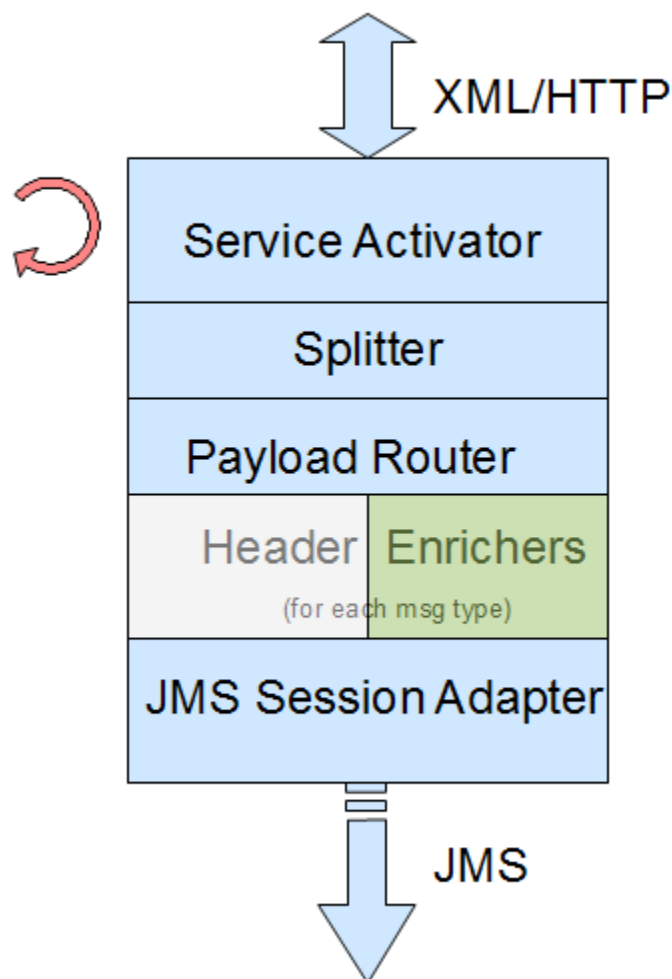
- *Header Enricher* to populate JMS Headers from Payload attributes.
- *Filters* as a mechanism to direct traffic across multiple output channel handlers.
- *Payload router* in the logger to split message traffic streams by Class type
- *Splitter* to fragment messages containing Collections into Individual components.
- *Transformers* to support Java Object Serialization
- Various *Output Adapters* such as Logging, File, STDOUT, JDBC and JMS
- Control of Event Driven Inputs based on Timer/Scheduler
- *Input adapters* including Custom Task and JMS MDB
- Use of Spring Expression Language for minor operations on Payload and Headers

Packaging Executable Components with Maven Shade

The source base relies on the Maven build framework and specifically the Maven Shade plug-in to generate tightly packaged standalone Java executable JARs. Configuration is accomplished using Java Properties files that are read from outside the packaged JAR. This approach creates simplicity as there is no need to set environment variables, write driver shell scripts, or transport dozens on dependency JAR files to the point of deployment. Each daemon can simply be run as `$ java -jar file.jar` with all configuration variables coming from properties files.

Message Pump Design

The Message Pump is a stack of basic Spring Integration components with only two custom-written Java classes needed to meet specific requirements for the project. While most of the Plumbing related burdens were entirely solved with out-of-the-box Spring Integration components, the Spring Philosophy is to provide extensibility through Composition and/or Inheritance so that specialization and oddball corner-case requirements can be accomplished without modifying base classes. As it turns out, there were some specific requirements from the Nextbus Webservice Service Level Agreement that required custom code ; these are summarized following the illustration for the Pump Daemon



Key Requirements

- The pump needs to *self-configure* its work list ; At startup time, this is done by traversing the Route and Stop catalog from the NextBus RouteConfig RPC to discover all routes and stops to monitor.
- The pump needs to maintain a time-expiry based policy for retrieving VehicleLocation data based on a ***Bandwidth Service Level Agreement*** with NextBus. NextBus requires that we update Vehicle Location state no more frequently than every 5 minutes and Stop Prediction no more than every minute.
- ***Division of workload***: The number of stops and routes on large transit networks is quite large ; upwards of 100 Routes on the MBTA and 300 Stops. To maintain freshness within the 5 minute window, we'd need to make hundreds of calls every 5 minutes. Further, Nextbus enforces a burst bandwidth SLA term of no more than 2 Mbytes every 2 seconds. To meet these requirements, the workload must be cutdown into pieces that spread into a many frequent calls inside a 5 minute outer window.

To meet these needs, we use the base component – a Service Activator driven periodically by a Timer to couple to a custom Task Driver.

- The Task Driver configures its own work schedule by interrogating the NextBus XML service to discover the needed Routes and Stops to manage.
- The Task Driver tracks completed work based on timestamps and uses time expiration to refresh data from the near-real-time feed without excessively updating or exceeding the SLA with Nextbus.
- The Task Driver delegates the gritty details of actually getting the data refresh to Worker classes that are responsible for the details of obtaining data for each possible data type (Prediction, Location, Service Messages, etc).
- The Task Driver further implements its own throttling scheme that divides the network traffic refresh workload into small but continuous requests rather than bursty and chunky behavior.

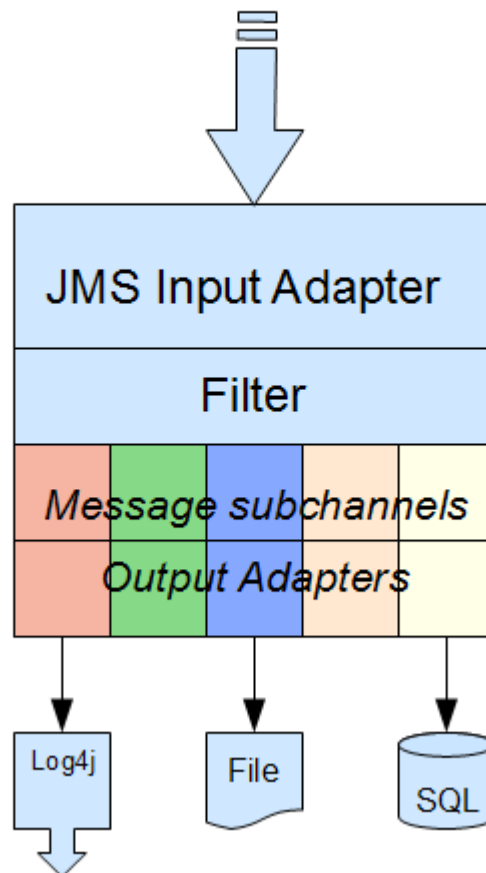
Once the Task Driver class was coded, the rest of the work to achieve the design fell nicely into place by simply wiring up existing Spring Integration components: The Scheduler and Task Executor are used to drive the Pump Task POJO on a one second interval ; a Splitter carves up the Collections returned by the NextBus XML adapter into individual instances, A Payload Router divides the traffic stream by class type into separate streams, two Header Enrichers populate the JMS Headers so that Topic Subscribers may “tune into” specific slices of the traffic stream. Finally, a JMS adapter is configured to combine the separately processed message subchannels and stream the headers and payload to a JMS Topic. The out of box JMS Outbound Adapter handles all of the plumbing required to send to JMS : JNDI lookup, caching a Session instance, handling JMS Connection setup and connection recovery when network outages intrude.

The bulk of the functionality is entirely achieved without any custom coding, rather simply wiring Spring Integration components together in an XML config file. Finally, the complexity of executable packaging ; Distilling classes from a dozen JAR dependencies ; is done transparently by the Maven Shade plugin.

Logger Design

The Logger implements a JMS Message Driven Bean with zero custom Java code. Properties file settings are used to configure the logger's behavior. The logger can do several things, each of which may be separately enabled:

- Print each received message to STDOUT
- Print each message to the log4j output stream – which may separately be configured in log4j.properties to filter and process log traffic.
- Insert into a SQL database various properties of the Message Headers
- Copy to a SQL database the Serialized POJOs in a BLOB column.
- Copy to the Filesystem the Serialized Java POJOs that have arrived.



An input listener capable of using Message Selectors is used to subscribe to the JMS Topic. This message stream is immediately split using a Payload Router – as different target output adapters are needed for each data type when logging into the database.

Message Filters are used in their simplest form to act as a valve to steer traffic to an output adapter based on how logging output is configured from the Properties file.

Finally, the logger uses a combination of Logging, File and JDBC based adapters to copy the payload out of the ESB and into their external destinations.

Supporting Non-Java (i.e. PHP, Ruby, C#, JavaScript) Clients

If you configure ActiveMQ to support the STOMP protocol as well as OpenWire, you can very quickly open the Message Bus up to non-Java clients. In fact, an example of how to subscribe to the bus using just a TELNET client is covered in another PDF Document for the impatient developers out there.

Since STOMP carries the JMS Message Headers, you'll have full access to route, timestamp and GPS location information without needing to deal with the payloaded Java POJO. For Web Developers who prefer to develop in Ajax, the STOMP protocol option will allow them to use the the ESB pumper on a Topic to explore Ajax-based consumption of message driven events.

Setup and Installation

Setup an Apache ActiveMQ Server

Setting up ActiveMQ is quite simple ; Download and unzip the distribution on some server. Since the default installation listens on non-privileged ports, it does not have to run as root.

1. Edit **conf/activemq.xml** and enable the OpenWire connector to listen on all network interfaces. You might also enable the STOMP connector while under the hood.

```
<transportConnectors>
    <transportConnector name="openwire" uri="tcp://0.0.0.0:61616"/>
    <transportConnector name="stomp" uri="stomp://0.0.0.0:61613"/>
</transportConnectors>
```

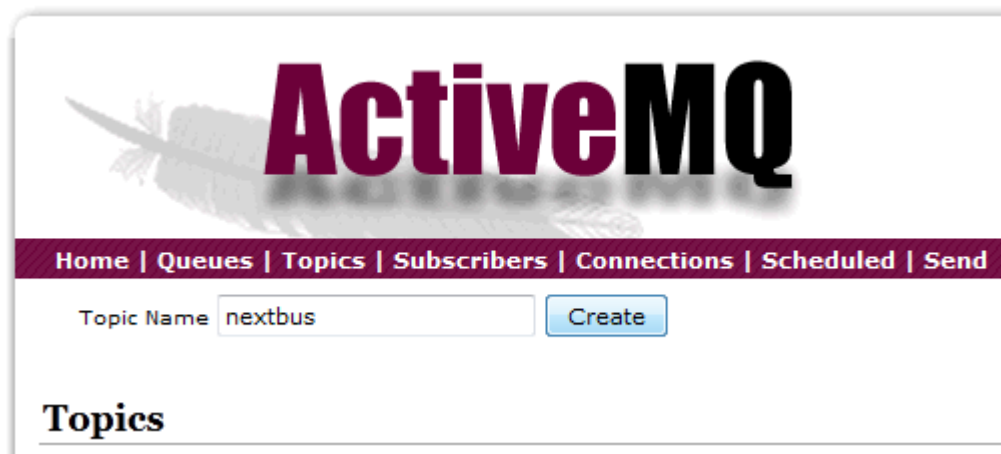
2. Start the activemq server, listening on all network interfaces (not just LOCALHOST/127.0.0.1)

\$./bin/activemq start

3. Using your browser connect to the ActiveMQ Console wherever the AMQ server is running:

<http://localhost:8161/admin>

Create a new Topic for the nextbus pump, call it “nextbus” as it will work with the default daemon configuration properties.



Setup an Apache Derby DB

Apache Derby is far easier to use for fast demos than MySQL or PostGres because security does not get in the way. Derby also supports XA very well, but that is another matter for another discussion.

1. Download Apache DB to some box where you want it to run and unzip the distribution.
2. Create an initial username-password pair of sa/sa by creating a **derby.properties** file in the unzipped base directory.

```
$ echo "derby.user.sa=sa" >> derby.properties
```

2. Start the Network Server and connect with the Derby client tool 'ij'

```
$ ./bin/startNetworkServier -h 0.0.0.0
```

```
$ ./bin/ij
```

3. Next, create an Empty database and Schema for the demo. Creating new databases on derby is remarkably simple. Once you have the target schema created, cut and paste the SQL DDL from the logger file 'nextbus-logger-ddl.sql' into IJ. This will create three tables:

PREDXN - a transcript of Prediction events that have arrived

VEHLCN - a transcript of Vehicle Location events

SERIALIZED - A table with a BLOB column that contains byte-stream serialized Java classes.


```

ij> connect 'jdbc:derby:foobar;create=true';
ij> create schema NEXTBUS;
0 rows inserted/updated/deleted
ij> CREATE TABLE NEXTBUS.VEHLCN (
>     AGENCY  CHAR(8) NOT NULL,
>     ROUTE   CHAR(12) NOT NULL,
>     VEHICLE CHAR(12) NOT NULL,
>     .      .      .
ij> CREATE TABLE NEXTBUS.SERIALIZED (
>     SERIALIZED BLOB NOT NULL,
>     CLASS VARCHAR(64) NOT NULL,
>     CRTIME TIMESTAMP DEFAULT CURRENT_TIMESTAMP
> );
>     .      .      .
0 rows inserted/updated/deleted
ij> exit;

```

You should now be able to connect to this new database remotely. Most IDE's include Derby JDBC support, as well as a SQL Browser GUI to explore the database. The connect string will look something like below, you'll need to put that in other properties files later for the Logger daemon:

`jdbc:derby://192.168.11.2:1527/nextbus`

Download Pumper, Edit Properties Files and Start Daemon

The pumper needs an empty directory for its two properties file. It's best to put the JAR file in the same directory as well. Download the pumper and the JAR files from the SourceForge drop point. Edit the nbpumper.properties file ; what you'll particularly need to set is:

- Hostname of your ActiveMQ Server host (leave port 61616 in place as this is standard fport)
- The transit agency ID that you'd like to traverse (i.e. mbta)
- Any specific routes to poll.. If you leave routes empty, the pumper will discover and pump ALL ROUTES in the agency.
- Whether to pump Vehicle Location, Predictions, or both.

Starting the Daemon is very simple! No Environment Variables or crazy command line options:

`$ java -jar nextbus-jms-pump-1.0.1-RELEASE.jar`

Give the Pumper about 30 seconds to a minute to initialize. Go back to the Active MQ Admin webapp and watch the Status Panel for “Topics”. Refresh this page every once in awhile. You should see the message count begin to rise once the pumper begins moving messages.

If you suspect setup problems, edit log4j.properties and restart the pump. Typically, you'll want to increase the log visibility for the ApacheMQ client, as well as possibly Spring's JMS and/or Integration package hierarchy.

Download Logger Daemon, Edit Properties File and Start

Start off with a Single Logger. You can add a second or third later as this is JMS Topic – Publish/Subscribe. *EACH LOGGER needs its own working directory* – because it will have its own unique nblogger.properties file. You should also run the Logger in a separate directory than the Pumper.

Download the Logger JAR and the startup properties files from the SourceForge download drop.

Edit the nblogger.properties file and add the following:

- Enable the STDOUT logger as your first step, Disable JDBC and Filesystem logger for now.
- Set your JDBC connect information for now, even though you have yet to turn on JDBC support.
- Set the Connect URL information for Apache ActiveMQ

Start the Logger

\$ java -jar nextbus-logger-1.0.1-RELEASE.jar

Wait awhile.. Since this logger is printing the Message channel to STDOUT, you can expect the toString() method to fire for VehicleLocation. serialized POJO. And this is what you soon see in the

```
VehicleLocation{vehicle=Vehicle{id=0874}, parent=Route{agency=Agency{id=mbta, title=MBTA, shortTitle=, regionTitle=Massachusetts}, tag=455, title=455}, directionId=455_1_var1, predictable=true, locationAtLastTime=Geolocation{latitude=42.5241478 N, longitude=-70.8961658 W}, speed=0.0, heading=356.0}
```

logger output.


Next Steps: JDBC and Filesystem Loggers; explore the STOMP protocol

Once your STDOUT logger is verified to be working, go setup the JDBC logger or the Filesystem logger. The JDBC logger creates a nice way to collect statistics on the Transit route's performance.

You can also use the STOMP protocol via a TELNET client to port 61613 of the ActiveMQ Server. This will let you hand type commands to SUBSCRIBE to the messaging server's destinations.

Once you have multiple subscribers configured, you'll be able to watch their status on the ActiveMQ Admin Webapp. Look at (Connections). Here we see the Pumper connected, 2 Logger daemons

concurrently listening, as well as a STOMP protocol user with a Telnet client.



Home | Queues | Topics | Subscribers | Connections | Scheduled | Send

Connections

Connector openwire

Name ↑	Remote Address	Active	Slow
nextbus-logger-57133-1347260101222-0:1	/127.0.0.1:64188	true	false
nextbus-logger-65364-1347596263530-0:1	/192.168.11.1:65365	true	false
nextbus-pumper-8954-1347260490374-0:1	/127.0.0.1:38447	true	false

Connector stomp

Name	Remote Address	Active	Slow
ID:localhost-10360-1347259234238-2:1	/127.0.0.1:48853	true	false

Further Topics for Consideration and Exploration

Understanding Performance Tuning Parameters

The pumper daemon relies on an internal throttling strategy to manage the request workload against the NextBus XML Service. NextBus limits the traffic for your connection to no more than 2MB every 2 seconds. There are further guidelines for the frequency for requesting predictions and vehicle locations.

For very large transit agencies, such as the MBTA in Boston, with nearly 200 Routes and Thousands of stops, it's conceivable to heavily load the XML endpoint if a Pumper is set to probe and ship the entire

Route system. To manage the traffic load, but balance the need for ESB data that is as close to real time as reasonable. It's best to determine what the minimum level of precision is needed for you application, so you can reduce the workload on NextBus and not waste provisioned capacity.

Message Selective Subscription with GPS Coordinates

Message Selectors provide a means to narrow a subscriber's view on the traffic stream. The JMS Provider will honor the Message Selector predicate and only copy and schedule traffic to your connection that you want to receive. Here's an example of how to use a Message Selector on the JMS Header properties. Lets tune into any VehicleLocation events that happen roughly within ¼ Mile of Fenway Park. Here's the GPS location of Center Field from Google Earth:

42.346700° N -71.097131° W

As a rough approximate, lets put Center Field at the center of a GPS Square with border points +/- 0.001000 degrees Latitude and Longitude from away the center. While this not might be exactly ¼ mi – it illustrates how one can use a rectangle or square to tune into Geocoded messages. So, here are the vertices of this square:

42.347700	-71.098131
42.347700	-71.106131
42.345700	-71.098131
42.345700	-71.106131

Now it's very simple to create a Message Selector expression that uses the corner coordinates of this box to dial into a GPS Grid:

Latitude > 42.345700 AND Latitude < 42.347700 AND Longitude > -71.098131 AND Longitude < -71.106131

For the Logging Daemon, you could set this selector in the nblogger.activemq.message_selector attribute of the nblogger.properties file and see for yourself. You could also specify this selector in the SUBSCRIBE command of the STOMP protocol.

Exploring Production Readiness and Reliability Issues

Now that you've deployed this ESB on your own and relish in the ease in which you can build applications around JMS Topics, you might think you have a system very close to Production capability that can serve thousands of SmartPhone users. **Not quite!** Here are some questions to think about regarding the Production Readiness of this implementation:

- Do the Producer and Consumer Components recover lost connections automatically when a participating resource trips out of service, or is manual intervention to revive the ESB? Transient failures that would occur in production include periodic outages of the NextBus Web Service, small network hiccups between components, and server restarts for ActiveMQ and the database.
- Are faults properly recorded in log4j output and do the fault messages logged there make sense and lead to a clear action for an application administrator? If a daemon crashes, does it leave a

useful UNIX Exit Code to allow the OS management tooling to take auto-restart action?

- Are any **MESSAGES LOST** when an ESB component crashes? What are the consequences of lost messages on the overall correctness of the system. For bus data, it might be only that an end-user misses a bus about to arrive and needs to wait for another turn. But for other types of systems, the consequences of a lost message might be more impactful.

You can find answers to these by “kicking the tires” of the completely integrated system. You might find that the outcomes of these experiments might warrant additional code changes to the project. It's up to you find out, and ultimately, a key step to building is to thoroughly consider whether what you've built will be **RELIABLE** and **PREDICTABLE** against the backdrop of likely, and expected issues in the Production Environment.

Some Fun Things to try on your own

- Simulate an ActiveMQ Crash and Database Crash (for the logger) by halting and restarting these components while traffic is in-flight. You can either shutdown the database and/or ActiveMQ with the start-stop scripts, or try a Unix KILL 9 to force them into post-restart crash recovery.
- Try setting up the File-based logger in a situation where file-system permissions will not let the daemon write, or, create an intentionally full filesystem. Are messages acknowledged? Would message traffic be “lost” once the filesystem access problems are resolved?
- What happens when the HTTP endpoint for NextBus goes down? How will we know that it has gone down other than the traffic stream suddenly goes dry. What happens when it comes back up? You could simulate this by tweaking firewall rules on your router and blocking out traffic. You can also change the URL for NextBus in the pumper properties file and point it to a dead endpoint (i.e. <http://localhost:9999>), or, to some other Website that accepts connections, but does not talk the XML wireformat the adapter expects (i.e. try <http://www.cnn.com>)

Porting to Mule, Camel, RabbitMQ, JBoss, MySQL, Postgres, etc?

Apache ActiveMQ and Apache Derby were chosen, respectively, because they can be downloaded and setup to go with very little housekeeping chores. Porting to other databases as well as other messaging providers is very simple because of Spring dependency injection and the Maven build platform:

- Adjusting pom.xml dependencies to reach the Client Runtime libraries of the desired JDBC Datasource or JMS Connection Pool classes.
- Modifying the Spring Context XML files as needed to provide any additional support (such as JNDI template for the JBoss container)

Maven Shade will package the nested dependencies for you.

What about Two Phase Commit in the Logger (JMS to JDBC) ?

In the previous section we posed the question is it possible to lose messages? YES! Furthermore, the implementation of the *Logger daemon is not fit to be used as a template for high-reliability requirements requiring strict message semantics.*

It's possible for the Logger to lose message if something gets in the way of the Database insert: A transient network outage, a database restart, a filled up database log or tablespace. In such a case, the arriving messages will be auto-acknowledged then lost by the database adapter.

To solve this problem, the JMS Acknowledge and the RDBMS Insert must be bound together by two phase commit. There are several ways to do this:

- Use a Standalone Transaction Manager such as Atomikos TransactionsEssentials® within the Logger daemon. You will have to convert to using XAConnectionFactory for JMS, XADataSource for your database as well as actually start the TP Monitor from Spring during bootstrapping.
- Get rid of ActiveMQ and use a Full Java EE Container with it's built in JMS Provider. Move the database insert code into a Stateless Session Bean and code up a Message Driven Bean that listens on topic. You will still have to setup XA resources for DataSource and JMS Connection Factory, but the Container's transaction manager will coordinate the 2PC for you.

Not every application will suffer if a few messages are lost. Again, it's a matter of quantify the impact and then determining if small risks and consequences can be tolerated. In such cases where small, transient failures are acceptable, there are other ways to risk mitigate:

- Modify the Logger such that database commits happen before acknowledging messages from JMS. This mitigates a lot of risk without requiring a TP monitor. However, there is still the possibility of a “lost acknowledgement” which would result in message redelivery and possibly a double insert to the database. Consider this as a plausible option if a full TP monitor seems overkill.

What about security?

For simplicity and fun, the Pumper is designed to stream messages without authentication in ActiveMQ. ActiveMQ is delivered out of box with no security – to ease prototyping and learning by new developers. Since the bus data is open on the internet, there is little need to protect the ESB other than to insure there are no malevolent publishers of bad information. So, try this as your first security task:

- Require password authentication for Topic Publishers
- Continue open, unauthenticated connections for Topic Subscribers

SSL is also possibility, however, both the Pumper and Logger Daemons would need additional modifications to support keystore/truststore.

References

Nextbus XML Feed Specification

<http://www.nextbus.com/xmlFeedDocs/NextBusXMLFeed.pdf>

Spring Integration 2.1.2 Release Reference Materials

<http://static.springsource.org/spring-integration/docs/2.1.2.RELEASE/reference/html/>

Atomikos TransactionEssentials Documentation

<http://www.atomikos.com/Documentation/ConfiguringTransactionsEssentials>

Legal Disclosures

NextBus® is a registered trademark of Webtech Wireless Inc.

Learn more at www.nextbus.com/about.

The Terms and Conditions for using the NextBus XML feed are provided in the Spec guide.

The bus data itself is Copyright the respective Transit Authority Agencies using Nextbus.

TransactionsEssentials® is a registered trademark of Atomikos BVBA.

Learn more at <http://www.atomikos.com/Main/TransactionsEssentials>

SpringSource is a registered trademark and division of VMWare