**System architecture and requirements**

**The aim of the project is to create the prototype of system for a distributed data management within existing RS Doc application.**

The main technical requirement states to use open source products for enterprise development. In spite of that the products were chosen among others are Spring Core 3.1.3, Spring Integration 2.2.1 modules, ActiveMQ JMS provider 5.8.0 and Postgres 9.2 rdbms.

The functional requirements are:

- Possibility to transfer files between distributed RS doc instances

- divide files by chunks during file transfer process to adjust provider channel speed and quality

- Possibility to call from the third part code due to the provided API, RS doc in our case

- Automatic connection reconnects on channel provider failure with message redelivery

- Possibility to synchronize several slave distributed RS doc instances simultaneously according to the master state

- Possibility for application modules to be loosely coupled, reliable, and asynchronous

- Logging possibilities during message delivery from master to slave’s hosts, with message acknowledge

- Possibility to secure message delivery

According to the requirements the system was built on the top of Spring integration module and utilizes JMS technology as a transport layer between distributed RS Doc instances.

The project utilizes ActiveMQ JMS provider with Postgress database as a pluggable message store. Spring integration [SI] module is used as a free light weight replacement instead of the commercial ESB provider tool.

SI centralizes messaging application business logic on distributed RS DOC instances and provides convenient means to communicate with different kind of data sources, as well as gateway components to communicate with third party applications.

Architecturally the application prototype consists of the following sub modules; according to the set functions they're provided:

a) Sender, responsible for message delivery, splitting in the case of need, routing and logging of out coming messages

b) Receiver, responsible for the incoming message delivery, aggregation, construction of notification message.

c) Monitor, responsible for the calculation of successful and unsuccessful messages delivery between different slave and master nodes, logging undelivered messages to the destinations.

d) JMS message provider

The modules and tasks to be planned and added in the nearest future:

a) The module for a notification on business error arisen under RS doc message processing

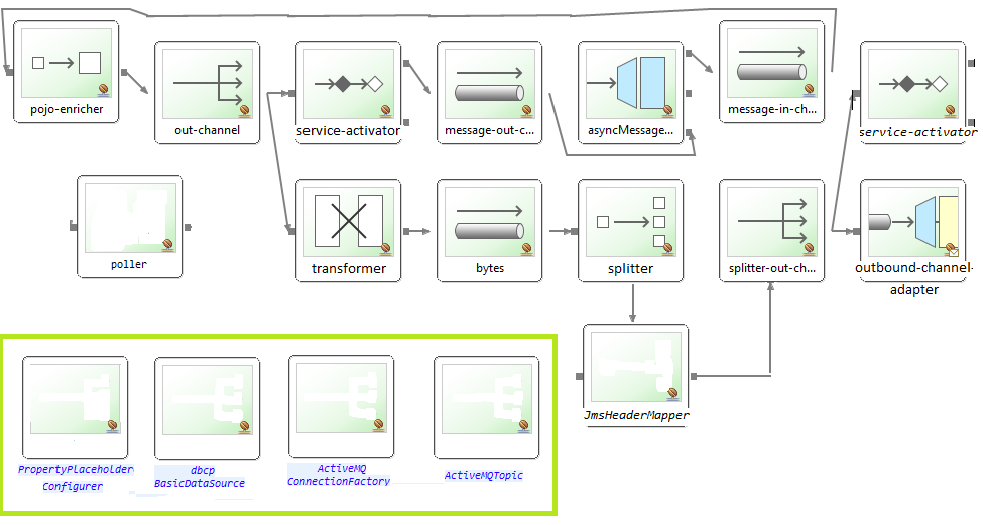
b) The module on custom message resend on undelivered messages

c) The module for monitoring the state of JMS provider, in the case of topic/queue persistent storage limit, etc. with email message notification for tech support

d) Redesign the existing architecture of single JMS provider for all RS DOC instances with standalone for every distributed RS doc instance

e) The module for the existing Web Services module integration in within system, to provide unique interface to the clients

***The Sender component workfllow definition and explanation***

******

The definition of components used:

*- message-in-channel, bytes are channel components to*decouple the sender and receiver components

- out-channel, splitter-out-channel is a publish-subscribe-channel *to*decouple the sender and receiver components

*- asyncMessageGateway is a gateway component that define API interface to expose ESB [Spring Integration] functionality to the external system [RS Doc]*

- *pojo-enricher*is a channel component to enrich the incoming message with an additional information

- messageTransformer is a transformer component, to reorganize message structure

- MessageSplitter is a flow component that provides splitter functionality

- jms -outbound-channel-adapter is component to publish to JMS topic/queue

- service-activator is an endpoint component that invokes a method on a bean when a message arrives at an input channel

-  correlationIdJmsHeaderMapper is a custom mapper to map attributes between internal ESB structure and external JMS provider

- poller is an utility component used to schedule pull data process from inbound components, currently doesn’t used probably as the only inbound component we had previously was inbound-file-adapter component.

The order of processing the income call from third party code is next:

1. *‘asyncMessageGateway’ component receives a call from third party code [RS Doc], the component provides API in form of the Java interfaces and transforms parameters in API methods to the internal SI message. On instantiation the component registers itself with two channels for inbound and outbound messages. Outbound message channel serves as confirmation of message processing by SI framework. Currently API provides one method that takes two parameters: String that presents message name in external system and byte array that respectively is a content of the message in external system.*
2. *‘pojo-enricher’ component is used as a means to add new parameter to SI header that denotes message name from Parcel domain object*
3. ‘out channel’ is a SI publish-subscribe channel that delivers internal message to two recipients simultaneously. Here we got message separation in two ways; the first one is used as a feedback confirmation for gateway component and logging facilities and the second as a preparation to the message to delivery via JMS outbound adapter.
4. ‘service-activator component’ [out-channel] is used as test case for figure out SI internal message structure during prototyping, could be removed afterwards in the case of need
5. ‘message-out-channel’ serves as a confirmation end point on message processing to the gateway component.
6. ‘transformer’ components extracts byte array from the incoming SI messages and creates new message with the byte array as a payload and just the same headers.
7. ‘splitter’ component splits incoming message depending on the message and the chunk size configuration parameter. Additionally it adds three header parameters [Sequence number, correlation ID, Sequence size], to denote that message was divided for the aggregator component on the client side.
8. ‘jms-header-mapper’ map message properties from internal SI message structure to the JMS structure, ‘correlation id’ in our case.
9. ‘jms-outbound-channel-adapter’ publishes message to the JMS topic/queue depending on JMS provider configuration.
10. ‘service-activator component’ [splitter-out-channel] is used to store log info in RDBMS layer for every first message in the group, that we get after splitting a big message. Just the same applies for the groups that consist from the one message.

The box on the screenshot below denotes utility beans in SI context:

* ‘PropertyPlaceholderConfigurer’ to provide support for a java.util.Properties inside Spring configuration file.
* ‘dbcp.BasicDataSource’ bean to provide data source to the underlying rdbms provider.
* ‘ActiveMQConnectionFactory’ to provide JMS provider connection factory to use in JMS enabled components
* ‘ActiveMQTopic’ presents dynamically created topic in Active MQ JMS Provider.

The module configuration:

# JMS Topic name

mqTopic=TOPIC.FOO

# JMS Provider host

mqHost=172.16.9.110

# local folder to pick up files to send [do not used any longer]

inputFolder=c:\\tmp\\input

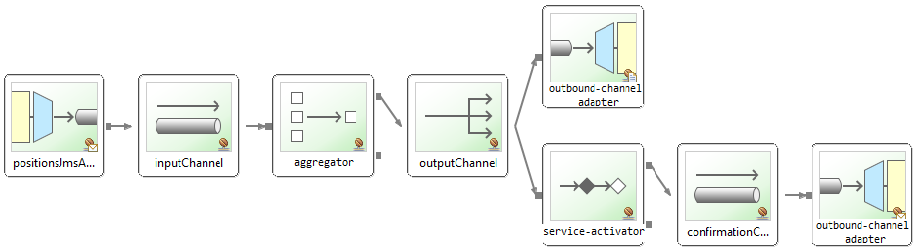
# The size of the chunk to split the whole message

chunkSize=1048576

The packaging process:

Every project is bundled as maven project, the call ‘mvn clean package’ produces distributable zip bundle in the target folder. In the scripts/msdos folder there’s file run.bat to start application in test mode. In the future just the same initialization process would be completed using RS Doc delivery bundle instead of the using ‘ru.rstyle.si.main.Sender’ test case.

***The Receiver component workfllow definition and explanation***



The definition of components used: [for the sake of simplicity only the new kind of components described]

1. ‘PositionsJmsAdapter ‘ – component subscribes on JMS topic, from where incoming messages arrive
2. ‘aggregator’ – component assembles multiple messages to create a single parent message
3. ‘file-outbound-channel-adaper’ – component job is to consume messages from a channel and write them to a file system

The order of processing is next:

1. ‘PositionsJmsAdapter ‘ component receives JMS messages from the topic , where the message could be presented as a single or a group of messages
2. ‘input channel’ is a queue channel, that delivers messages to the aggregator component
3. ‘aggregator’ component job is to assemble multiple messages to create a single parent message, who’s payload forms a byte array presenting the file transmitted
4. ‘outputChannel’ is publish-subscribe channel, that delivers messages simultaneously to tow subscribers.
5. ‘outbound-channel-adapter’ [output-channel] presents itself the component that consumes messages from a channel and write them to a file system. In the future possible to develop the interfaces of different kinds, jdbc, ftp, etc. to accommodate client needs.
6. ‘service-activator’ component get the domain name where the client is resides and create new message with that info as a payload, all headers also copied from the original message, including JMS Correlation ID, that lately used by the Monitor component to identify confirmation message.
7. ‘confirmationChannel’ is a queue channel that provides confirmation messages to the outbound JMS adapter
8. ‘outbound-channel-adapter’ [confirmationChannel] is an outbound JMS adapter to deliver conformation JMS messages to the conformation queue.

# The JMS topic name to subscribe for incoming messages

mqTopic=TOPIC.FOO

# The JMS durable subsciption name

durable.subscription.name=R-Style.com

# The JMS durable client ID

client.id=R14942.R-Style.com

# The JMS queue name to send confirmation messages

mqConfirmQueue=CONFIRM.FOO

# The host name where JMS provider is deployed

mqHost=172.16.9.110

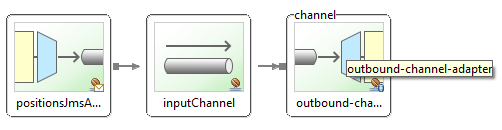
# The local folder name to accomodate arriving files

outputFolder=c:\\tmp\\output

The packaging process:

Every project is bundled as maven project, the call ‘mvn clean package’ produces distributable zip bundle in the target folder. In the scripts/msdos folder there’s file run.bat to start application in test mode. In the future just the same initialization process would be completed using RS Doc delivery bundle instead of the using ‘ru.rstyle.si.main.Receiver’ test case.

***The Monitor component workfllow definition and explanation***



The definition of components used: [for the sake of simplicity only the new kind of components described]

1) outbound-channel-adapter component presents itself jdbc adapter to execute sql queries on the database when the message arrives

The order of processing is next:

1. ‘positionsJmsAdapter’ subscribes at the queue where the confirmation messages are arrived
2. ‘inputChannel’ is a queue channel to deliver confirmation messages to the JDBC outbound adapter
3. ‘outbound-channel-adapter’ executed delete operation on the ‘status\_table’ for those messages that have just the same Correlation ID and domain.name. We have to remember that throw out JMS message life cycle beginning from message creation in the sender module and further in the receiver/monitor modules, correlation ID is just the same.

Below provided DDL scripts for Postgresql 9.2, that creates two tables, the status table to hold info about unconfirmed messages and domain table to hold info about target destinations domain names that correspond client domain names where durable topic clients are resides.

-- Table: status\_table

-- DROP TABLE status\_table;

CREATE TABLE status\_table

(

correlation\_id character varying NOT NULL,

parcel\_name text,

domain\_id integer

)

WITH (

OIDS=FALSE

);

ALTER TABLE status\_table

OWNER TO postgres;

-- Index: fki\_dmain\_id\_fk

-- DROP INDEX fki\_dmain\_id\_fk;

CREATE INDEX fki\_dmain\_id\_fk

ON status\_table

USING btree

(domain\_id);

-- Table: domain

-- DROP TABLE domain;

CREATE TABLE domain

(

id integer NOT NULL,

domain\_name character varying,

description character varying,

CONSTRAINT domain\_pk PRIMARY KEY (id)

)

WITH (

OIDS=FALSE

);

ALTER TABLE domain

OWNER TO postgres;

***The security implementation***

Currently security implementation schema consists of three layers:

1. Authentication [JMS client credentials under attempt of connection to JMS queue or topic]
2. Authorization [JMS client permissions under attempt to read, write from topic or queue, or attempt of create ESB managed objects]
3. Stored passwords encryption on JMS broker
4. Adding corresponding properties to the ActiveMQConnectionFactory on the JMS client side.

Next we briefly describe each of them.

1)

All the security concepts in activeMQ are implemented as plugins, so the Authentication plugin in correspondence to our existing module structure looks like that:

<simpleAuthenticationPlugin>

<users>

<authenticationUser username="system" password="${activemq.password}" groups="users,admins"/>

<authenticationUser username="user" password="${guest.password}" groups="users"/>

<authenticationUser username="guest" password="${guest.password}" groups="guests"/>

<authenticationUser username="api2queue" password="${api2queue.password}" groups="producer"/>

<authenticationUser username="queue2api" password="${queue2api.password}" groups="consumer"/>

<authenticationUser username="monitor" password="${monitor.password}" groups="control"/>

</users>

</simpleAuthenticationPlugin>

Here we added three users ‘api2queue’, ‘queue2api’, ‘monitor’ additionally to the predefined users. The user names correspond to the logical module structure described earlier. Additionally we have groups attribute, whose purpose is to define the group names the user to belongs to.

Groups used as markers in Authorization plugin that we describe next.

2)

The current configuration states that users that belong to the group admins, have possibility to read, write and manage all the topics and queues.

Users that belong to the consumer group have permission to write to the ‘CONFIRM.FOO’ queue and read only users that belong to control group.

Users that belong to the producer group have permission to write to the TOPIC.FOO and read only the users that belong to consumer group.

Besides we setup additional configuration that grants needed rights to the Advisory objects.

That way we implement fine grained access control based on the groups, that grant or not certain kind of permissions on managed objetcs.

<authorizationPlugin>

<map>

<authorizationMap>

<authorizationEntries>

<authorizationEntry queue=">" read="admins" write="admins" admin="admins" />

<authorizationEntry queue="CONFIRM.FOO" read="control" write="consumer" admin="admins" />

<authorizationEntry topic=">" read="admins" write="admins" admin="admins" />

<authorizationEntry topic="TOPIC.FOO" read="consumer" write="producer" admin="admins"/>

<authorizationEntry topic="ActiveMQ.Advisory.>" read="guests,users,control,producer,consumer" write="guests,users,control,producer,consumer" admin="guests,users,control,producer,consumer"/>

</authorizationEntries>

</authorizationMap>

</map>

</authorizationPlugin>

3)

Encryption of stored passwords on JMS provider side.

With the help of encrypt command listed below we generate encrypted passwords.

$ bin/activemq encrypt --password activemq --input mypassword

Encrypted text: eeWjNyX6FY8Fjp3E+F6qTytV11bZItDp

Encrypted passwords could be stored in ordinary properties files, using used ENC() to wrap them.

The sample configuration file, for our application listed below:

activemq.username=system

activemq.password=ENC(m41w1yTgQCO4I7XDX8loBZ9zX75GCWrQ)

guest.password=ENC(w6TUxdh+ABCRY50eb6u/CA==)

queue2api.password=ENC(wkdEVZXGNqyMpUcO7Bt6Bjby7w/Dwxuy)

api2queue.password=ENC(v+OR6pohoQvoh617WQ3fsiTkRQvvdU2D)

monitor.password=ENC(/9WiCvevcDyiOXNf+d3xdQ==)

Additionally we ought to provide the additional configuration instead of existing one, in JMS broker configuration file used to launch it.

Currently the file name is activemq.xml, inside it we have to replace the next code fragment :

<!-- Allows us to use system properties as variables in this configuration file -->

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

<property name="locations">

<value>file:${activemq.conf}/credentials.properties</value>

</property>

</bean>

with a such one:

<!-- Allows us to use encrypted system properties as variables in this configuration file -->

<bean id="environmentVariablesConfiguration" class="org.jasypt.encryption.pbe.config.EnvironmentStringPBEConfig">

<property name="algorithm" value="PBEWithMD5AndDES" />

<property name="passwordEnvName" value="ACTIVEMQ\_ENCRYPTION\_PASSWORD" />

</bean>

<bean id="configurationEncryptor" class="org.jasypt.encryption.pbe.StandardPBEStringEncryptor">

<property name="config" ref="environmentVariablesConfiguration" />

</bean>

<bean id="propertyConfigurer" class="org.jasypt.spring.properties.EncryptablePropertyPlaceholderConfigurer">

<constructor-arg ref="configurationEncryptor" />

<property name="location" value="file:${activemq.conf}/credentials-enc.properties"/>

</bean>

That gives us the possibility to use encrypted property files.

The main trick is that, we need to supply in runtime the argument used to decrypt passwords, the argument name is ACTIVEMQ\_ENCRYPTION\_PASSWORD. Please see the example below for the Unix environment:

* Set environment variable:

$ export ACTIVEMQ\_ENCRYPTION\_PASSWORD=activemq

* Start the broker:

$ bin/activemq start xbean:conf/activemq-security.xml

* Unset the environment variable:

$ unset ACTIVEMQ\_ENCRYPTION\_PASSWORD

Without environment variable set we just got some kind of error message during the JMS broker startup process.

1. Adding additional properties on the client side, using Spring configuration style:

<!-- connection factory for ActiveMQ -->

<bean name=*"connectionFactory"* class=*"org.apache.activemq.ActiveMQConnectionFactory"*>

<property name=*"brokerURL"*>

<value>failover:tcp://${mqHost}:61616</value>

</property>

<property name=*"userName"* value=*"${amq.username}"* />

<property name=*"password"* value=*"${amq.password}"* />

</bean>

Here we added to new properties username and password, to authenticate user during handshake process.

**Additional options to be applied**

It’s possible to achieve just the same functionality using JAAS pluggable modules that included in Apache MQ distribution; currently the built-in implementations include (property files, LDAP, SSL certificates).

If we have additional requirements that do not cross with built-in JAAS modules we could implement it using just the same JAAS API under request.

Also there’re possibilities to unite existing Spring Security implementation with spring integration module to restrict user access to certain kind of managed objects inside ESB (SI in our case).