Unit testing a message flow in IBM Integration Bus 10.0

2

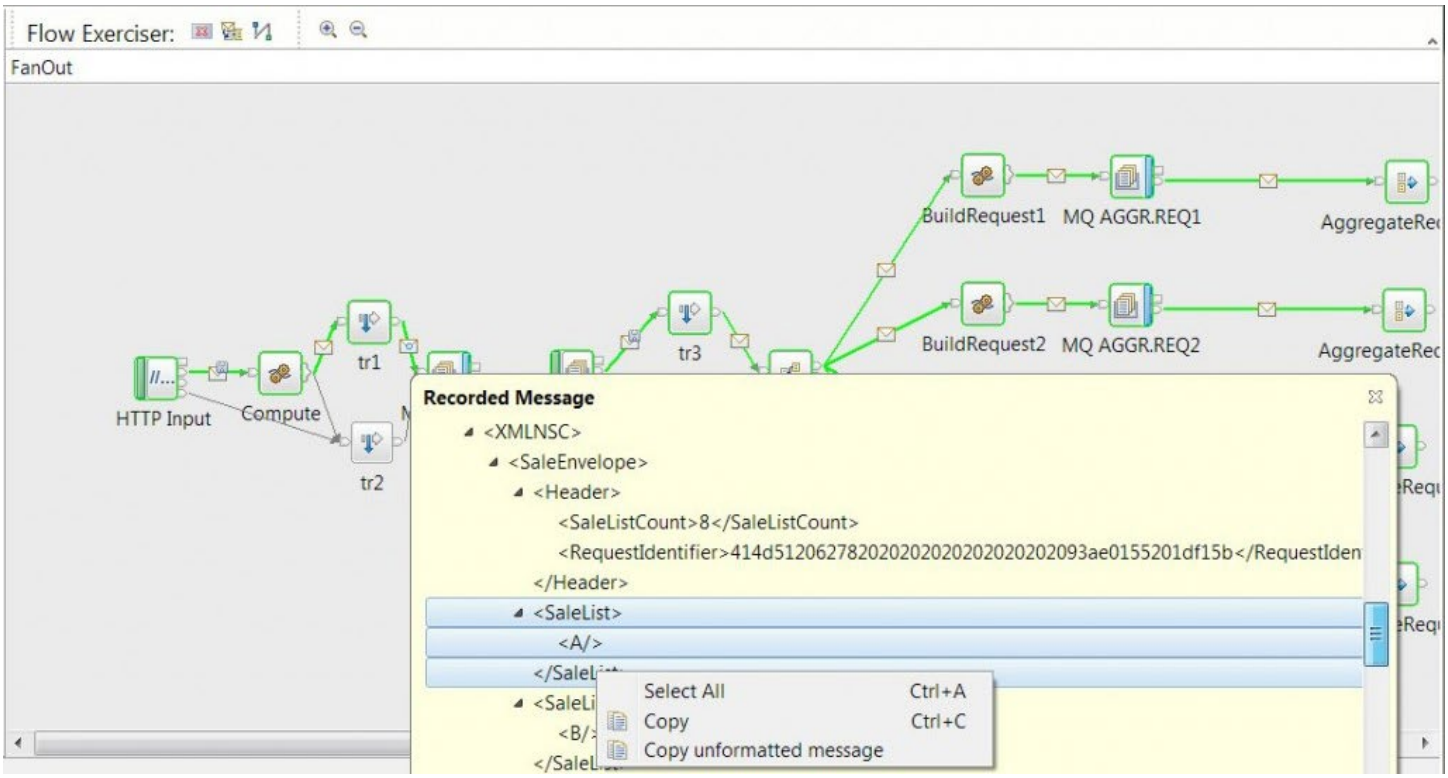
RobHenley
Published on March 27, 2015 / Updated on June 8, 2016

Originally posted March 27 2015 | Visits (3445)

This post describes how to use the APIs provided in IBM Integration Bus Version 10.0 (IIB) to unit test a message flow by checking the logical tree at specified points in the flow.

The Flow Exerciser

You can use the Flow Exerciser in the IBM Integration Toolkit to enable message recording and then inspect the logical tree at any connection in a flow. The Flow Exerciser is described in the following Help topic Testing your message flow by using the Flow exerciser.

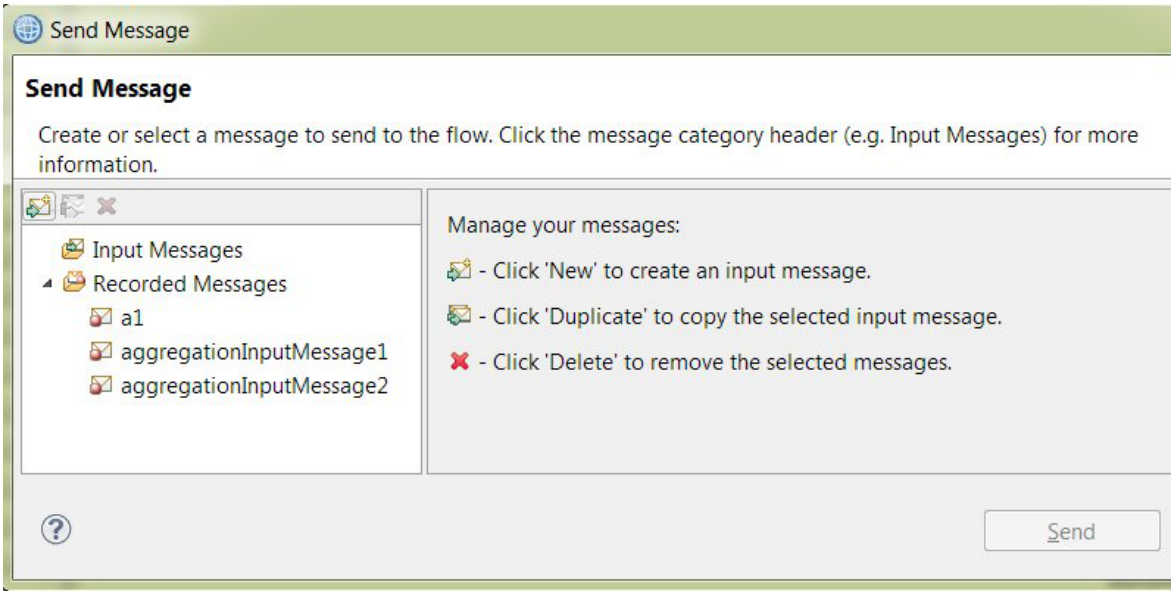


The Recorded Messages are displayed in XML, representing the structure and content of the logical tree at a point in the message flow. The XML displayed (and selectable via Copy on the context menu) is a convenient visualization of the tree, and is a simplified version of the internal XML representation that IIB uses to record a logical tree.

Once you have recorded the run of a flow (as shown by the highlighted paths in green), you can save any input messages for use in exercising the flow in the future. (Candidate input messages have a save icon in the top right.)

When you inject such a message, the flow is initiated directly, without the need for an external transport to send an input message to the node.

These saved messages use the internal XML format mentioned above, and can be viewed under Other Resources in your toolkit Application. (The internal XML format for any Recorded Message can also be viewed by selecting Copy unformatted message from the context menu.)



In summary then, you are able to inspect the execution of a message flow (roughly equivalent to inserting Trace nodes on every connection in your flow), and to re-run that flow efficiently without using an external transport.

Recording and Injecting Messages Programmatically
The ability to record and inject messages is also supported via the IIB REST and Integration APIs. The table below summarizes the APIs, but you should check the product documentation for details.

You can read about the IBM Integration API in the product documentation, which contains a link to the Javadoc.
You can read about the IBM Integration RESTful Api in the html documentation found in the server/docs/REST folder under you product install location.

Description	REST (/apiv1)	Integration API
enable/disable recording mode	/executiongroups/egName ?action=enableTestRecordMode ?action=disableTestRecordMode	setTestRecordMode
enable/disable injection mode	/executiongroups/egName ?action=enableInjectionMode ?action=disableInjectionMode	setInjectionMode
get recorded messages	/test/recordedtestdata ?p1=v1&p2=v2&p3=v3...	getRecordedTestData
get recorded message count	/test/recordedtestdata ?action=count	getRecordedTestDataCount
inject a recorded message	/test?action=inject &p1=v1&p2=v2&p3=v3...	injectTestData injectTestData
clear recorded messages	/test?action=clear	clearRecordedTestData clearRecordedTestData clearRecordedTestData

Test Approach
Using either of the APIs, it is now possible to write ‘unit tests’ for you message flows:
1. inject a previously recorded message to drive the flow
2. record the resulting messages at selected connections and compare with previously captured data

Injection

Invoking a flow over the normal transport (e.g. MQ for an MQInput node) has several disadvantages within a unit test framework: the transport mechanism must be available (e.g. a queue manager or an HTTP listener), your test has to make a transport-specific client call, and the input message is likely to vary from one invocation to the next. Most messages have some fields such as identifiers or timestamps that vary, making it more difficult to compare the resulting logical tree.

As well as alleviating these issues, injecting a message directly into the flow should be faster than sending a message over a transport layer. When exercising a flow using message injection, reply node output is suppressed. Specifically, HTTPReply and SOAPReply nodes do not attempt to reply to a client (since in this case the client does not exist). However, other output nodes will attempt to interact as normal with external systems – e.g. MQOutput with a queue manager.
The general principle is that when a previously recorded message is injected into a flow, the flow should behave as far as possible in the same way that it would if a message had been sent to the input node over the normal transport mechanism.

Recording

Assuming you want to drive the flow by injecting a message, where do you get the message from?

One way is to write a script to capture an input message. This could be the same script you will use for your unit test, with a flag to 'capture reference data'. When you run the script with this flag it would use an external client (e.g. Apache HTTP client) to drive the flow, and record the logical tree for both the 'input' message and for any other connections where you will want to compare the tree.
Another approach is to use the toolkit Flow Exerciser to drive the flow (e.g. by building a new Input Message), then capture the recorded messages manually by selecting Copy unformatted message on the Recorded Message for each connection you are interested in.

Comparing Messages

Lets say you modify a map or some ESQL in your message flow to improve performance. You inject your recorded input message and record the resulting tree just after your Mapping node or Compute node, giving you 'before' and 'after' messages representing the logical tree after the transformation. How can you compare them to verify that your performance improvements have not changed the important data?

A Recorded Message uses an XML format that, first and foremost, allows IIB to reliably rebuild a logical tree. To make the message as human-readable as possible, the format uses XML tag names to mirror the logical tree elements. However, it is important to realize that this is not always possible: some logical tree element names are not valid XML names (some don't even have names), and sometimes we need to encode values or store additional information such as the logical tree type. In particular, even if your original external message was XML, the internal XML tree representation will be similar but not identical. For example, any attributes in your original XML become elements in the logical tree and in the XML representation.

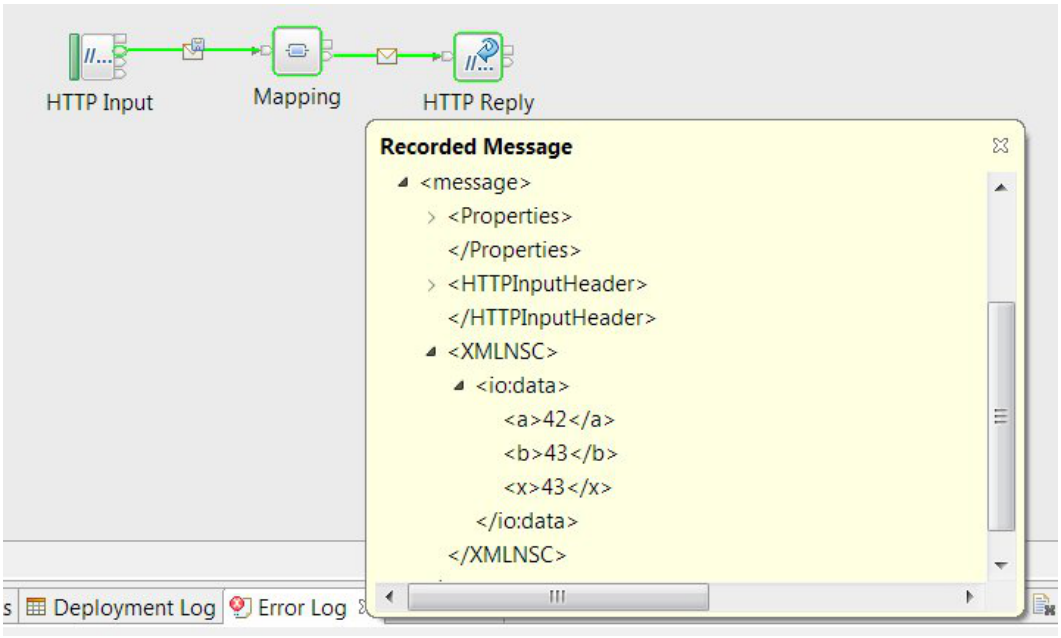
Lets say you have retrieved a Recorded Message that you want to check. There are two main approaches:

1. a string comparison against a previously captured 'reference' message. Ideally you'd want this to report at least the position(s) at which the strings differ.
2. an XML comparison, which could be an element-by element comparison (a tree walk) or a specific path check (for example using XPath).

In the code examples that follow I show one way of running an XPath assertion against the XML representation of the logical tree.

Integration API Example

As an example we'll use a simple flow with a mapping node. The flow takes an XML message where a and b have integer values, and returns a similar value with an additional child element that is set to max(a,b).



For simplicity, the code that follows shows a sequence of calls in a single Java method. The code assumes that the flow you want to test has been deployed to a running integration server.

These are the key imports we need for our Java progam:

```
import java.io.*;
import java.util.List;
import java.util.Properties;
// the IIB Integration API
import com.ibm.broker.config.proxy.*;
// for parsing the returned XML
import com.ibm.broker.config.common.*;
import javax.xml.xpath.*;
import org.w3c.dom.*;
```

This is a single Java method that makes all the required Integration API calls:

```
 static boolean testOnServerUsingInegrationAPI(String integrationNodeName, String integrationServerName)
  {
    boolean result = false;
    try {
      // Integration API initialization
      BrokerProxy nodeProxy = BrokerProxy.getLocalInstance(integrationNodeName);
      if (!nodeProxy.isRunning()) return false;

      ExecutionGroupProxy serverProxy = nodeProxy.getExecutionGroupByName(integrationServerName);
      if (!serverProxy.isRunning()) serverProxy.start();


      // ENABLE injection and recording mode
      serverProxy.setInjectionMode(AttributeConstants.MODE_ENABLED);
      serverProxy.setTestRecordMode(AttributeConstants.MODE_ENABLED);

      // the calls above are asynchronous. We could implement a listener to check when the
      // calls have taken effect but to keep the code simple we'll include a 1 second sleep.
      try   { Thread.sleep(1000); } catch (InterruptedException e) { }


      // INJECT a previously-recorded message to drive the flow
      // you can cut and paste this from the Recorded Message in the Flow Exerciser, or obtain it programm
atically
      // using similar code to this. In this case the behaviour of the flow does not depend on the content
 of the
      // Properties or HTTPInputHeader, so these are omitted. Likewise we haven't supplied data for the
      // LocalEnvironment or Environment trees (which you would need to associate with their respective pr
operties).
```

```
      String message = "4243";

      Properties injectProps = new Properties();
      injectProps.setProperty(AttributeConstants.DATA_INJECTION_APPLICATION_LABEL,  "MapApp");
         // Application name
      injectProps.setProperty(AttributeConstants.DATA_INJECTION_MESSAGEFLOW_LABEL,  "simplemap");
         // Flow name
      injectProps.setProperty(AttributeConstants.DATA_INJECTION_NODE_UUID,          "simplemap#FCMComposit
e_1_1"); // input node
      injectProps.setProperty(AttributeConstants.DATA_INJECTION_WAIT_TIME,          "60000");
      injectProps.setProperty(AttributeConstants.DATA_INJECTION_MESSAGE_SECTION,    message);

      // The node uuid consists of the flow name, appended to the node id that you can find in the '.msgfl
ow' in the bar file

      boolean synchronous = true; // making the call synchronous means we don't need to sleep afterwards

      result = serverProxy.injectTestData(injectProps, synchronous);

      // RETRIEVE recorded message
      // because we enabled recording, IIB has recorded the logical tree at each connection that
      // the injected message travelled through. Now we retrieve the message(s) we want by specifying the
      // set of properties that define those connections. In this case I just specified the mapping
      // node as the source of the connection I want to check, but you can specify more properties
      // if required such as the target node, terminal etc.
      Properties filterProps = new Properties();
      filterProps.put(Checkpoint.PROPERTY_SOURCE_NODE_NAME,  "simplemap#FCMComposite_1_3");

      List dataList = serverProxy.getRecordedTestData(filterProps);
      // why do we get a list back? If the flow has been run more than once since recording was enabled, o
r
      // if the flow contains a loop, then multiple messages will be recorded.

      // TEST the message against some criteria
      boolean usingREST = false;
      result = runAssertionsAgainstData(dataList, usingREST);

      // clear recorded data and reset server to turn off recording and injection
      serverProxy.clearRecordedTestData();
      serverProxy.setInjectionMode(AttributeConstants.MODE_DISABLED);
      serverProxy.setTestRecordMode(AttributeConstants.MODE_DISABLED);
    }
    catch (ConfigManagerProxyPropertyNotInitializedException e) { System.out.println("ERROR: "+e); }
    catch (ConfigManagerProxyLoggedException e)                 { System.out.println("ERROR: "+e); }
    finally {
      return result;
    }
  }

  // helper method to run assertions against the recorded message you retrieve - this is used for the REST
 example too
  static boolean runAssertionsAgainstData(List dataList, boolean useREST)
  {
    // check that at least one recorded message was retrieved
    if (dataList == null || dataList.size() == 0) { return false; }

    boolean result = true;
    try {
      // check that the message looks ok. You could make a string comparison against a
      // previously captured reference message. Or, as here, check the value of specific
      // elements in the message that you are interested in.
      String xPathAssertion1 = "/message/XMLNSC/data/x[.='43']";

      for (RecordedTestData data : dataList) {
        String messageData = data.getTestData().getMessage();
        // see section on REST API if (useREST) { messageData = unwrapXML(messageData); }

        ByteArrayInputStream bais = new ByteArrayInputStream(messageData.getBytes(CommsMessageConstants.TE
XT_ENCODING));
        Document logicalTree      = XMLHelper.parse(bais);
        Element  logicalTreeRoot  = logicalTree.getDocumentElement();

        // evaluate XPath expression against the parsed XML
        XPath xPath = XPathFactory.newInstance().newXPath();
        NodeList nodes = (NodeList)xPath.evaluate(xPathAssertion1, logicalTreeRoot, XPathConstants.NODESET
);
        if (nodes.getLength() != 1) {
          result = false;
          System.out.println("ERROR: no elements found that match XPath "+xPathAssertion1);
        }
      }
```

```
    finally {
      return result;
    }
  }
```

These are not official performance figures, but a rough measurement based on repeated runs of this inject-retrieve cycle on a laptop. Each run of the method `testOnServerUsingInegrationAPI` took approximately 4 seconds, broken down as:

0.5s – initialization
1s – enabling record and injection mode (this is the sleep, so this overhead could be reduced by polling)
2s – injection
0.5s – retrieval

Of course, in practice the initialization and enablement calls would be done once for a set of 'tests' leaving us with a cost of roughly 2.5 seconds per test, 80% of this being the (synchronous) injection call.

REST API Example

We'll use the same flow example as above with the single mapping node. The REST program needs an HTTP client. I show how you could use the Apache Fluent API but any similar API can be used.

By comparison with the Integration API example, note that:
1. the REST API may be a little slower.
2. the injection data is complicated slightly by the need to escape XML markup. You can see this in the example hard-coded XML in the example below.
3. similarly, the retrieved recorded messages include XML entities.

If you look in the `runAssertionsAgainstData` method above, the call to `unwrapXML(messageData)` is needed to convert:

```
<message>&lt;x&gt;X&lt;/x&gt;</message>
```

to

```
<x>X</x>
```

which is in a form we can then parse and run XPath assertions against. (I have not included the implementation of unwrapXML here, because I wanted to focus on the IIB APIs. If you find you do need an implementation of this please let me know.)

```
import org.apache.http.*;
import org.apache.http.client.fluent.*;

  static boolean testOnServerUsingREST(String host, String port, String integrationServerName)
  {
    boolean result = false;
    try {
      String webadminURL = "http://"+host+":"+port+"/apiv1/";
      int maxTimeToEstablishConnection       =  2000; // ms
      int maxTimeToWaitForDataOnConnection   = 10000; // ms

      // ENABLE injection and recording mode - these are the HTTP client calls
      Request.Put(webadminURL+"executiongroups/"+integrationServerName+"?action=enableTestRecordMode")
        .connectTimeout(maxTimeToEstablishConnection).socketTimeout(maxTimeToWaitForDataOnConnection).exec
ute();
      Request.Put(webadminURL+"executiongroups/"+integrationServerName+"?action=enableInjectionMode")
        .connectTimeout(maxTimeToEstablishConnection).socketTimeout(maxTimeToWaitForDataOnConnection).exec
ute();

      // INJECT a previously-recorded message to drive the flow
      // as for the Integration API case you can cut and paste the data from the Recorded Message in the F
low Exerciser, or
      // obtain it programmatically. Unfortunately, for the REST API the XML markup has to be represented
using the XML
      // entities < etc. For the purposes of this example I've just hard-coded the data, again using the m
inimal message
      // body, but you can add localenvironment, environment etc in the same way.
      // The # character is encoded for use in the URL using the standard URLEncoder (and becoming %23)
      String message = "";
      message += "<message xmlns:iib='http://com.ibm.iib/1t/1.0' iib:parser='WSRoot'><Properties/><XMLNSC
iib:parser='xmlnsc'><ns1:data xmlns:ns1='http://ns1'><a iib:valueType='INTEGER'>42</a><b iib:valueType='IN
TEGER'>43</b></ns1:data></XMLNSC></message>";
      message += "";

      String injectionURL = webadminURL+"test?action=inject"+
                            "&applicationName="+"MapApp"+
                            "&messageFlowName="+"simplemap"+
```

```
                        "&nodeName="+URLEncoder.encode("simplemap#FCMComposite_1_1", "UTF-8");

      int statusCode = Request.Post(injectionURL)
            .version(HttpVersion.HTTP_1_1)
            .bodyString(message, ContentType.APPLICATION_XML)
            .execute().returnResponse().getStatusLine().getStatusCode();

      // RETRIEVE resulting recorded messages
      // as for the Integration API example I'm just specifying the source node (the mapping node) to
      // define the flow connection I want the data for. Additional properties can be added in
      // the same way (with the 2nd and subsequent parameters introduced by & rather than ?)
      String retrieveURL = webadminURL+"test/recordedtestdata"+
                           "?sourceNodeName="+URLEncoder.encode("simplemap#FCMComposite_1_3", "UTF-8");

      String retrievedData = Request.Get(retrieveURL)
            .connectTimeout(maxTimeToEstablishConnection)
            .socketTimeout(maxTimeToWaitForDataOnConnection)
            .execute().returnContent().asString();

      // build data set from returned XML using ITB-supplied helper method
      List dataList = RecordedTestData.getRecordedTestDataFromXML(retrievedData);

      boolean usingREST = true;
      result = runAssertionsAgainstData(dataList, usingREST);

      // clear recorded data and reset server to turn off recording and injection
      Request.Put(webadminURL+"test/recordedtestdata?action=clear")
        .connectTimeout(maxTimeToEstablishConnection).socketTimeout(maxTimeToWaitForDataOnConnection).exec
ute();
      Request.Put(webadminURL+"executiongroups/"+integrationServerName+"?action=disableTestRecordMode")
        .connectTimeout(maxTimeToEstablishConnection).socketTimeout(maxTimeToWaitForDataOnConnection).exec
ute();
      Request.Put(webadminURL+"executiongroups/"+integrationServerName+"?action=disableInjectionMode")
        .connectTimeout(maxTimeToEstablishConnection).socketTimeout(maxTimeToWaitForDataOnConnection).exec
ute();
    }
    finally {
      return result;
    }
  }
```

Summary

I've outlined one approach to using the APIs provided in IIB v10 for testing your message flows. I hope you've found this brief introduction useful. It would be interesting to know if you think the APIs could be useful to you, and how you think they could be improved. Thanks for reading!

TAGS MQSI, IIBV10, API

RobHenley

---

2 comments on"Unit testing a message flow in IBM Integration Bus 10.0"

k ravi January 27, 2017

Hello Team,

Have a nice day.

Please can any one provide the list of check list for verifying my interfaces all are working fine after fix pack installation.

currently I am using IIB fixpack v9003, i am planning to update to new fix pack version.

Reply (Edit)

Petar June 09, 2016

Thanks for this article. It was helpful for me to find out ways how can be flow tested.

I have two questions.
1. Could you recommend this approach as automatically test (run on Jenkins as JUnit test)?
2. If the flow contains some logic for changes on database, would you recommend as a part of this approach to approve also those data also?

Thanks and best regards,
Petar

Reply (Edit)