A Catch-All Approach to Failed Instances

Author: Rex Townsend

Date: 2/24/2023

Salient Process
Your process. Our passion.

Contents

Introduction

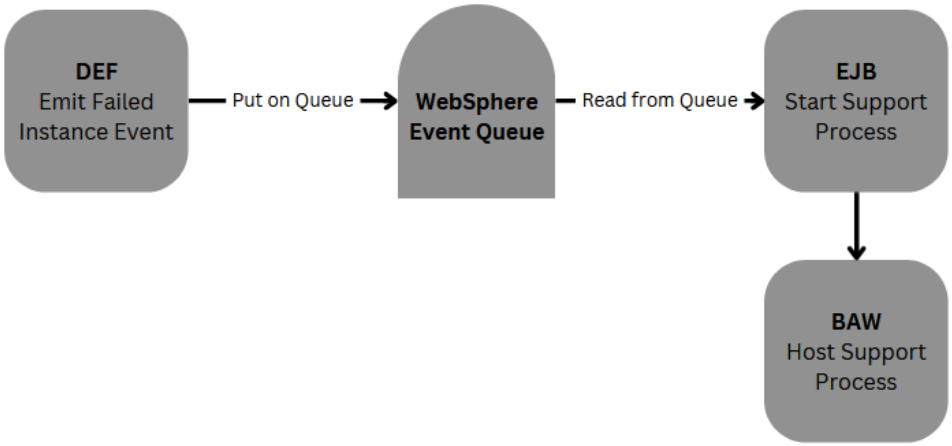
In every application that is promoted to a production environment, we need to consider what happens when an instance goes into a failed state. This could be handled directly in the code by attached error events and special routing or by a support team who is monitoring for failures. There are a few issues that arise with these approaches:

- Nobody is alerted when an instance goes into a failed state. Generally, they are discovered by manual check by someone with access to production Process Inspector, or an incident reported by a user (in many cases, the user would not know)
- Handling process level errors is difficult to do without cluttering the diagram and sacrificing human-readability for what can be an artifact exposed to your users (via Process Portal or API)
- Handling process level errors needs to be done for each individual process; no application-wide catch all exists out of the box
- Handling process level errors is not all encompassing, gaps will be left in what is handled and something uncaught will inevitably arise

By relying on the Dynamic Event Framework and a couple simple components, we can apply application-wide error handling that plugs the gaps between specifically handled scenarios and leveraging the process/task framework to notify the appropriate team with a task when a process failure is encountered so that it is known and can be handled appropriately.

Note: This approach is used to complement the above approaches and can be used completely in tandem with them or as your sole method of handling errors that make it up to the process level unhandled.

General Description of the Implementation by Component



Dynamic Event Framework

- Turn on the Dynamic Event Framework
- Configure the event filters to listen for all process failures for a given application

A Support Process

- Create a process that will be started when a process failure is detected

Create and Install EJB to WAS

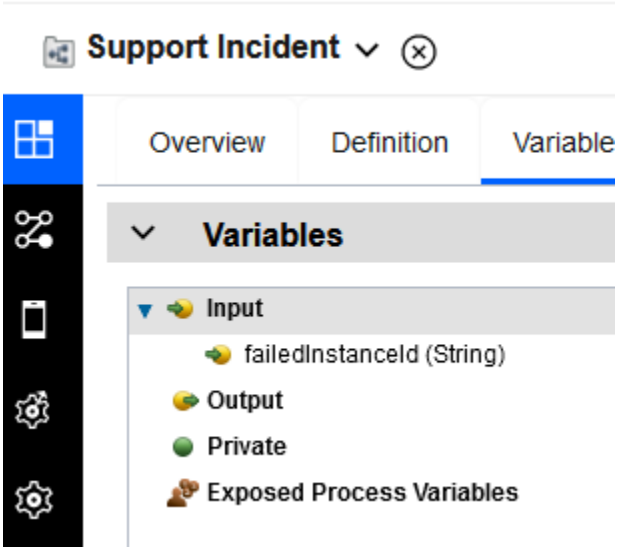
- This is the component that will act when there is a new event emitted

Support Process

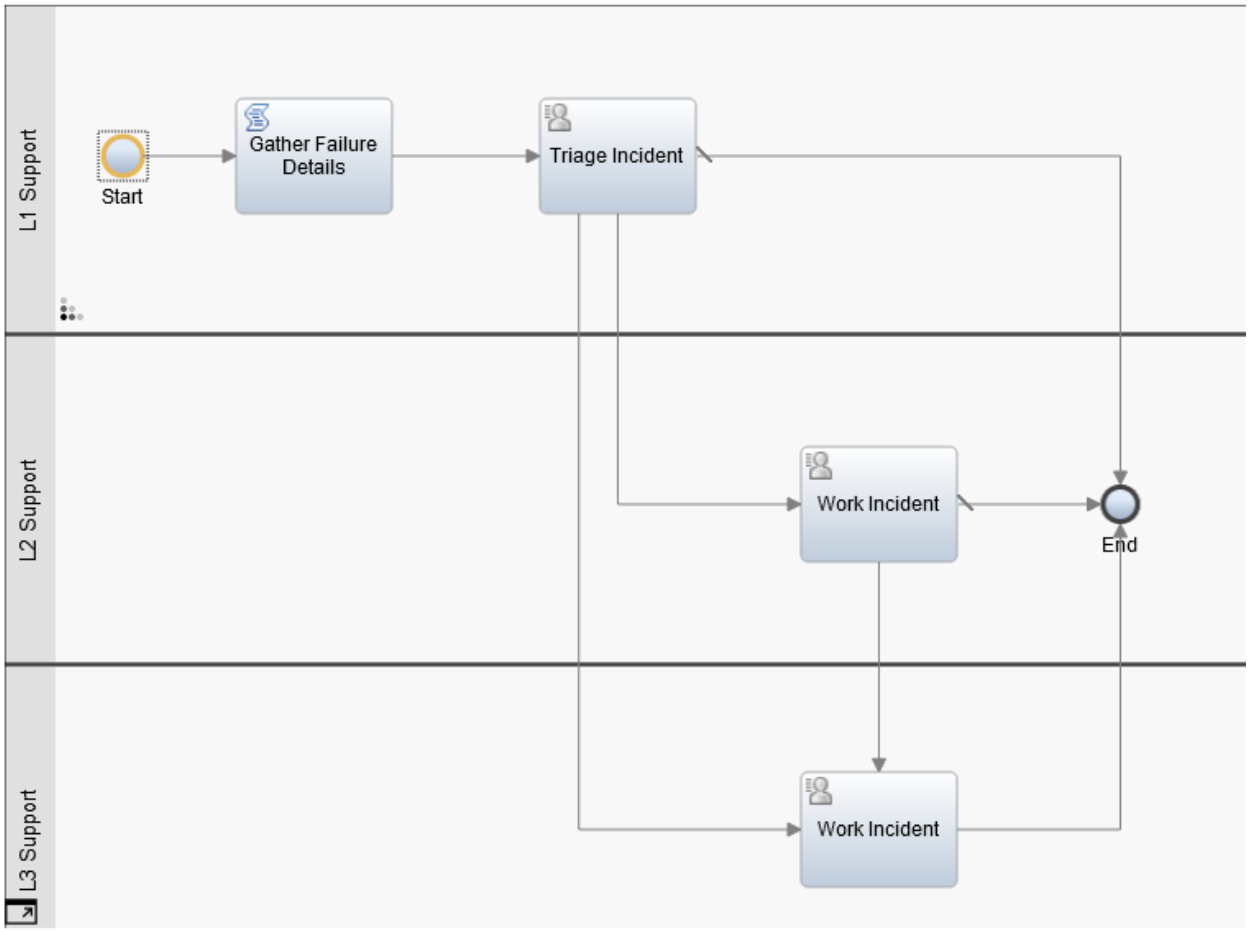
The first step in creating our centralized approach to process level errors is to create a process that is started when a failure occurs so that the proper people are at least aware of the failure and can take action. Preferably this process will exist in a separate application from the one we are supporting, but this is not required.

Below is an example of what our Support process could look like, but this will vary based on the organization. At the very least, we will want to map in a variable for the failedInstanceId so that the person attending to the failed instance know which instance failed. We will be using this variable in a later step.

Support process input variables



Example Support process diagram



It is important to consider what would happen if the Support instance itself failed before it gets to the human task “Triage Incident”. Another failed instance event would be emitted if it were in the same application that you are designing this to support.

Dynamic Event Framework

The Dynamic Event Framework (DEF) is what will provide us with a central point where we can listen to all events of a specific type. In our case, we are interested in all process events that have a status of failed. The DEF will be responsible for emitting events of the subscribed type to the JMS queue you define in the steps linked below.

If the environment you are implementing this in does not already have the Dynamic Event Framework configured, refer to Capturing Business Automation Workflow events for external consumption for instructions.

Defining subscriptions will be another edit in the SampleConfigureEventsToJMS.py file. By default, you will be looking for a line that looks like this, indicating that you are subscribing to all events.

```
subscriptions=[
'*/*/*/*/*/*/*'
]
```

The 7 asterisks represent: Application Name / Version / ComponentType / Component Name / Element Type / Element Name / Nature

So if our application is Hiring Sample (HSS) and we want to listen to all process failures, our subscription would look like this:

Subscriptions example

```
subscriptions=[
'HSS/*/BPD/*/*/*/FAILED'
]
```

Run the two commands below to apply the update to the filters

Reload DEF commands

3. Run the script to configure DEF.
   a. At a command line, go to the `bin` directory under your deployment manager profile home directory. Run the `SampleConfigureEventsToJMS.py` script.

```
wsadmin -lang jython -f c:\SampleConfigureEventsToJMS.py
```

   b. Run the `SampleReloadDEF.py` script to cause the DEF to refresh dynamically.

```
wsadmin -lang jython -f c:\SampleReloadDEF.py
```

Create and Install an EJB

Now we need a mechanism to actually listen to the queue and process the messages. This will take place in a Message Driven Bean. You can use any IDE to do this, but we used IBM Integration Designer.

Listening to the queue

In an Enterprise Java Bean project, create a Message Driven Bean. Your newly created file should have a stubbed method called onMessage. This method will be called whenever there is a new message on the queue, in our case based on how we set up the filters, this method will be called every time an process instance goes to a failed state.

```
public void onMessage(Message message) {
    // TODO Auto-generated method stub

}
```

Use a library of your choice to parse the message which can be XML or JSON depending on how you configured the dynamic event framework. At the very least, we want to supply our Support process with the instance ID of the failed instance so that we can gather more information on the failure manually, or even provide some mechanisms to the team assigned the Support task such as retrying via API, moving token via API, reassigning etc. To learn more about the structure of the event itself, refer to Activity monitoring events.

Note: If you change the filters to include any additional events, additional checking will need to be added when introspecting the events so that you do not start Support processes for other event types.

Starting the Support process

Next, we will need to make a REST call to the BAW REST API in order to start our Support process. Depending on how you created your Support process, the URL parameters may vary, but at the very least you need to supply the BPDID and Process Application ID of the Support process and the app it resides in. In our example, we also add…

```
&params={"failedInstanceId": "12345"}
```

…in order to map to the input variable for our process with the same name of String type.

An important call out here is that if you created your Support process in the same application that you are listening for failed instances in, then a failure in the Support process itself will trigger another Support process to be created. This could cause Support processes to be started in quick succession until manually stopped. If you do have your Support process in the same application, create an exception in the code so that a Support process is not started if the failure comes from the Support process itself.

```
// Make sure failed process is not the designated Support Process
if (type.equals(EVENT_TYPES.ProcessFailed) && category.equals(EVENT_CATEGORIES.BPD) && !SUPPORT_BPD_ID.equals(event.processId))
{
    startSupportProcess(instanceObj.processId);
}
```

Event Processing Framework

While this article is specific to handling process failures, it could be extended to support any kind of event the Dynamic Event Framework is able to submit and could even include custom events that you may define in your codebase. If after reading this you feel that your application will need a lot more event processing, consider inquiring about Salient Process' Event Processing Framework by reaching out to us via our Contact Us page. This framework breaks down event handling into its core components: Event flows, event processors, and routes in order to be more modular, promote reuse and allow developers to focus on higher level thinking rather than dealing with the plumbing. Use of this framework would replace the EJB mentioned in this article.

Event Processing Framework Case Study

Experience the power of our Event Processing Framework in action! See how it transformed the operations of the Global Automotive Leader and enabled efficient event-triggered notifications, streamlined processes, and enhanced communication. Dive into our detailed case study to witness the remarkable results firsthand. Click here to explore the case study and discover the potential of our Event Processing Framework for your business.