SSF Tools: Mock Integration Executor User Guide

Document Revision History

Revision Date	Written/Edited By	Comments
May 2018	Matt Shirilla	Initial documentation.
June 2018	Paul Wheeler	Minor amendments for SSD v6.

Table of Contents

Introduction

The Mock Integration Executor is an easy way to simulate a ticket system in your development environment so that you can test integration-related workflows and other integration-dependent features. It can also help you simulate situations where there is a high latency in the integration provisioning methods. It will work with IdentityIQ 6.4 and above.

Installation

The Mock Integration Executor ships with the SSD and is deployed by default as part of the build process if your IdentityIQ version is 6.4 or greater. Deployment of the IntegrationConfig object used by the tool can be switched on or off with the "deployMockIntegrationExecutor" property in the build.properties file. Set this to "true" to enable deployment, or "false" to disable.

Components

The tool includes two Java classes for the Integration Executor. After compilation by the build process, the Java classes can be found here:

- WEB-INF/classes/sailpoint/services/standard/mockintegrationexecutor/MockIntegrationExecutor.class
- WEB-INF/classes/sailpoint/services/standard/mockintegrationexecutor/TicketSystemClient.class

In addition there is an IntegrationConfig object:

- config/SSF_Tools/MockIntegrationExecutor/Source/IntegrationConfig/IntegrationConfig-MockIntegrationConfig.xml

How it Works

The Mock Integration Executor integrates with IdentityIQ workflows in the same way as any other integration. For more information on how to configure workflow options that affect integrations in access request workflows refer to these documents:

Lifecycle Manager Workflows: https://community.sailpoint.com/docs/DOC-4897

LCM Subprocess Workflows: https://community.sailpoint.com/docs/DOC-4917

This is how the Mock Integration Executor is used:

1. An access request is submitted, approved, and the workflow is in the Approve and Provision stage.
    - In the Access Request UI there will be a section for the Mock Integration Executor that shows:
        - The status is pending.
        - The Provisioning Request ID is the IdentityRequest #.
    - The Mock Integration Executor "opens a ticket" by adding to the tickets map in the MockIntegration Custom object.
        - The ticket number is the number of the access request.

    - The ticket has a status of “queued”.
    - If the Custom object does not exist, the Mock Integration Executor will create it, so you never have to.
2. The developer manually edits the status of the ticket in the Custom object by updating it from “queued” to “committed” using the IdentityIQ debug screen.
3. The Perform Maintenance task runs and processes the “Check Status of Queued Items” workflow. In the Access Request UI the section for the Mock Integration Executor shows the status is finished.

Note that by default in IdentityIQ the status of the ticket will only be checked once every hour, even when the Perform Maintenance task is run more frequently. During development and testing stages it is useful to have it check more frequently. This can be done by modifying the script that sets the value of the *provisioningStatusCheckInterval* variable of the “Check Status of queued items” workflow; the integer value returned by the script is the number of minutes it will wait before checking ticket statuses.

Configuration

Mock Integration Executor Specific Configuration

These are settings that are specific to the Mock Integration Executor, configured in the IntegrationConfig object.

assumeSuccessfulTicketOperations

There is an option in the IntegrationConfig to automatically set the status of the ticket to "queued" when it is provisioned and set it status to "committed" automatically when the first check status occurs. This is handy if you want to avoid setting the ticket status manually, yet you want to have the provisioning by integration.

Note that IdentityIQ checks the status of a ticket immediately after provisioning, so if you use this option you are effectively opening and closing the ticket in the provisioning step. The workflow will never go to the check status step if you use this option.

```
<entry key="assumeSuccessfulTicketOperations" value="true"/>
```

customObjectName

The IntegrationConfig object has a customObjectName attribute that sets the name of the Custom object that contains the tickets. The Custom object will be created automatically when the first ticket is generated, if it does not already exist.

```
<entry key="customObjectName" value="MockIntegration"/>
```

simulatedLatency

The Mock Integration Executor simulates the latency of the ticketing system by calling “`Thread.sleep(simulatedLatency);`” at the start of any provisioning or check status call. This is useful for testing how your IdentityIQ environment handles high latency provisioning. The value is in milliseconds.

```
<entry key="simulatedLatency" value="1000"/>
```

IdentityIQ Generic Configuration

These are configuration items that are common to all integrations and can be used with the Mock Integration Executor. These are set in the IntegrationConfig object.

ManagedResources

This property denotes which application(s) will be provisioned via the integration.

```
<ManagedResources>
    <ManagedResource>
        <ApplicationRef>
            <Reference class="sailpoint.object.Application"  name="AppName1"/>
        </ApplicationRef>
    </ManagedResource>
    <ManagedResource>
        <ApplicationRef>
            <Reference class="sailpoint.object.Application"  name="AppName2"/>
        </ApplicationRef>
    </ManagedResource>
</ManagedResources>
```

StatusMap

This attribute maps the status from the ticket system to the internal IdentityIQ status.

```
<entry key="statusMap">
    <value>
        <Map>
            <entry key="closed" value="committed"/>
            <entry key="open" value="queued"/>
            <entry key="retry" value="queued"/>
        </Map>
    </value>
</entry>
```