HP Propel

Software Version: 1.10

Service Exchange SDK

Document release date: December 2014

Software release date: December 2014

Legal Notices

Warranty

The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein. The information contained herein is subject to change without notice.

Restricted Rights Legend

Confidential computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Copyright Notice

Trademark Notices

Adobe® is a trademark of Adobe Systems Incorporated.

Microsoft® and Windows® are U.S. registered trademarks of Microsoft Corporation.

Oracle and Java are registered trademarks of Oracle and/or its affiliates.

UNIX® is a registered trademark of The Open Group.

RED HAT READY™ Logo and RED HAT CERTIFIED PARTNER™ Logo are trademarks of Red Hat, Inc.

The OpenStack word mark and the Square O Design, together or apart, are trademarks or registered trademarks of OpenStack Foundation in the United States and other countries, and are used with the OpenStack Foundation's permission.

Documentation Updates

The title page of this document contains the following identifying information:

- Software Version number, which indicates the software version.
- Document Release Date, which changes each time the document is updated.
- Software Release Date, which indicates the release date of this version of the software.

To check for recent updates or to verify that you are using the most recent edition of a document, go to the following URL and sign-in or register: **https://softwaresupport.hp.com/**

Use the **Search** function at the top of the page to find documentation, whitepapers, and other information sources. To learn more about using the customer support site, go to:

https://softwaresupport.hp.com/documents/10180/14684/HP_Software_Customer_Support_Handbook/

You will also receive updated or new editions if you subscribe to the appropriate product support service. Contact your HP sales representative for details.

Support

Visit the HP Software Support Online web site at **https://softwaresupport.hp.com/**.

This web site provides contact information and details about the products, services, and support that HP Software offers.

HP Software online support provides customer self-solve capabilities. It provides a fast and efficient way to access interactive technical support tools needed to manage your business. As a valued support customer, you can benefit by using the support web site to:

- Search for knowledge documents of interest
- Submit and track support cases and enhancement requests
- Download software patches
- Manage support contracts
- Look up HP support contacts
- Review information about available services
- Enter into discussions with other software customers

- Research and register for software training

To learn more about using the customer support site, go to:
https://softwaresupport.hp.com/documents/10180/14684/HP_Software_Customer_Support_Handbook/.

SDK Overview

This chapter describes the content supplied in the SDK package. It also explains how to setup your development environment with the SDK.

SDK contents description

Within the SDK you will find the following directory structure and files:

- doc
  - HP Propel Service Exchange v.1.10 SDK.pdf
- javadoc
- m2-repo
- sx-content
  - JIRA
  - sm-problem
  - sm-problem-cx
- sx-ui-war

javadoc - contains sx-api.jar and sx-adapter-api.jar javadocs. These are the APIs that you can use to develop a custom java adapter to add support for a new backend system type that HP SX interacts with.

m2-repo - contains a maven repository with all the dependencies needed to develop your custom content. The most significant artifacts contained are listed in the table below.

artifactId	description
sx-api	Core HP SX api.
sx-adapter-api	Adapter-related HP SX api.
oo-sx-plugin	This is a crucial artifact for HP OO flows development in HP SX. It contains the implementation for the HP OO operation that enables you to send messages to HP SX AMQP queues within your HP OO flow.
sx-maven-plugin	A useful development tool that allows for deploying development content in an automated fashion.

sx-content - contains source code for the example implementations. It is referenced in the HP Propel Service Exchange v.1.10 SDK. Under each subdirectory you will find an example maven project that builds an HP SX content pack. The content pack is built into content-<project_name> module's target.

sx-ui-war - contains a testing UI web application war file.

Development setup

In this section you will learn how to setup a maven project to develop your custom content.

Prerequisites
• Running HP SX instance - It is presumed that you have access to an HP SX instance dedicated to your development.

Setting up your project

1. **Populate your maven repository with SDK artifacts** - Copy the contents of the m2-repo directory into your local maven repository (it is usually in `<your home directory>/.m2/repository`). By doing so you have maintained all the necessary dependencies to compile your HP SX content pack and to run HP SX development tools.

2. **Setup your maven project** - The specific structure of the maven project depends on the use case that you would like to implement. In the most general case you will setup a maven project that contains three sub modules. The content sub module, oo content pack sub module, and sx adapter sub module. Use the example implementation as a starting point. JIRA provides the most complex example, containing a new backend system adapter implementation. See How to develop an adapter (JIRA).
3. If you need to create your own OO flows in HP OO Studio you have to also populate a OO maven repository with SDK artifacts. Copy the contents of the m2-repo directory into the OO maven repository (it is usually in `<your home directory>/.oo/data/maven`).

Development using sx-maven-plugin

The SDK provided `sx-maven-plugin` artifact is a tool that simplifies content development. It provides an automated way to deploy your content i.e. you do not need to build an HP SX content pack and upload it. Instead you only need to run the maven build and your content is ready to be tested in your HP SX instance.

See How to extend HP SX Content (HP SM Problem entity) where its setup and usage is described in more detail.

HP SX testing UI

In the `sx-ui-war` directory of the SDK package you will find a war file containing a developer testing UI.

Deploy this war into your development HP SX instance jboss application server. The web application will be available under `/sx-ui` context i.e. the following URL:

```
https://sx_host:8444/sx-ui
```

The HP SX testing UI needs to be configured in a similar way as the HP SX administration UI that is part of HP SX itself. Proper configuration is needed for these files:

- `<YOUR_HP_SX_JBOSS_HOME_DEPLOY_DIR>\sx-ui.war\WEB-INF\classes\config\users.json`
- `<YOUR_HP_SX_JBOSS_HOME_DEPLOY_DIR>\sx-ui.war\WEB-INF\classes\config\infrastructure.json`

`users.json` - the configuration is the same as in the native HP SX administration UI. See the HP SX Configuration Guide for details. By default the user admin of the Provider organization has the UI role which enables access to the development features.

The valid roles in the HP SX testing UI are:

- UI
- ADMINISTRATOR

NOTE: When enabling the UI role in the HP SX testing UI you need to enable the role in the native HP SX administration UI as well (i.e. add the role in `users.json` located in `sx.war`.)

`infrastructure.json` - here you need to configure a single value - secretKey. This must match the `secretKey` of the IDM instance that your HP SX instance uses.

infrastructure.json

```
{
    "AUTHENTICATION": {
        "secretKey": "<your_secret_key>"
    }
}
```

HP SX testing UI use cases

Generally it can be said that the HP SX testing UI enables development without the need to enter the Propel portal. It mainly supports the following use cases:

- Submit order requests: test a request to fulfill use case
- Submit ticketing requests: test a ticket management use case
- Notifications monitor and handling: this is where you can monitor the request state and perform approve/deny actions.

Where to go next

In case you need to:

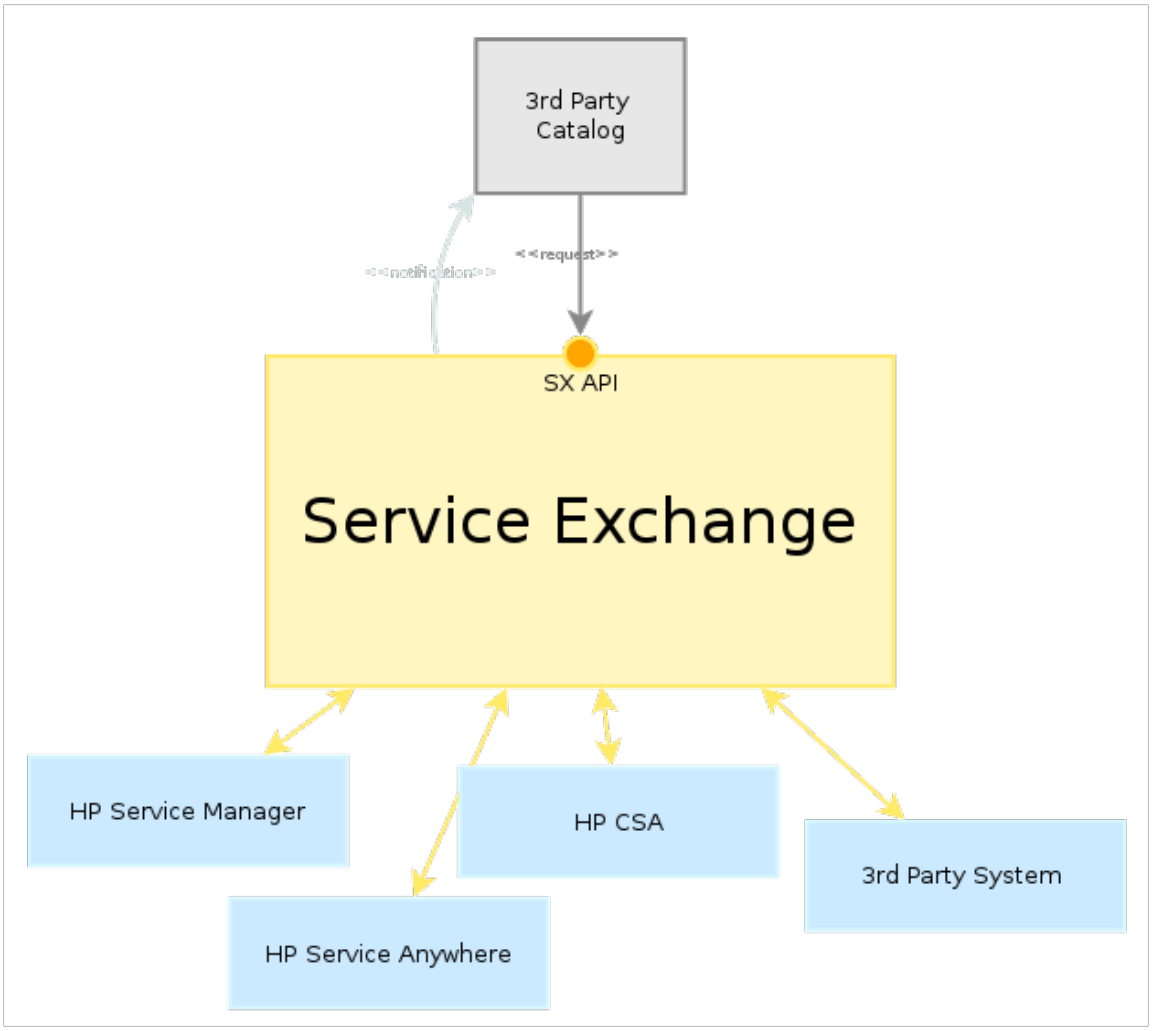
a. **Develop HP SX content for a system already supported by HP SX** (for example HP SM). In this case you do not need any java adapter coding. You will need to provide operation definitions and optionally HP OO flows. Read How to extend HP SX Content (HP SM Problem entity).
b. **Add support for a system type not yet supported OOB**. In this case you need to implement the java adapter for your system, see How to develop an adapter (JIRA). This chapter presumes that you have a basic understanding of the development setup explained in How to extend HP SX Content (HP SM Problem entity).

HP Service Exchange Overview

Service Exchange Architecture

Overview

HP SX is an extensible and customizable framework that allows clients to integrate with any backend system without the need to implement a specific exchange format for each system. Clients communicate with HP SX through SX REST APIs and use a unified data format independent of the target backend system. HP SX then processes the request, transforms it into a backend system-specific data format, and sends it to the system. During the processing of the request, HP SX notifies the client about progress and results.



High level architecture

The powerful HP SX features like extensibility and customizability are achieved through software architecture utilizing "**adapters"** and "**content packs"**.

Adapter role

An HP SX adapter is a component that interacts with a particular *underlying system* and makes this underlying system accessible to HP SX functionality. An example of an underlying system is: HP SM, HP CSA, SAP or any other similar product. Such underlying systems will be called **b ackend systems** in this document. In order to enable a *backend system* - make it accessible by HP SX - one needs to write implement an *adapte r*. The adapter then adapts a particular *backend system* to HP SX paradigms (queues, notifications, operation execution, etc.) As a result, HP SX enables multiple different *backend systems* and makes their functionality available to HP SX clients, for example Propel catalog.

Content packs

HP SX content packs are the key customization components. They contain the high level process definition modeled in HP Operation Orchestration flows (OO flows), and definitions of backend system interactions (operations). They provide business logic to the specific adapter. For example the approval process of an order is modeled in OO Flow. The create order, approve operations etc. must be defined.

OO Flow implementation and the operations that have to be defined depend on the specific features that the content pack supports.

The operations are defined in a special HP SX JSON notation that is interpreted by the adapter's component called ***operation executor***. The operations typically define a set of calls to backend systems' APIs. These calls (steps of the operation) are executed sequentially. The operation

definition framework uses Freemarker templates to compose URLs, request bodies, transform responses and others. The Freemarker templates are also a part of content packs.

Content packs can be deployed into HP SX at runtime.

AMQP

The adapters interact with surrounding components through AMQP. The chosen AMQP implementation is RabbitMQ (http://www.rabbitmq.com/).

HP SX use cases

HP SX is designed to support the following use cases:

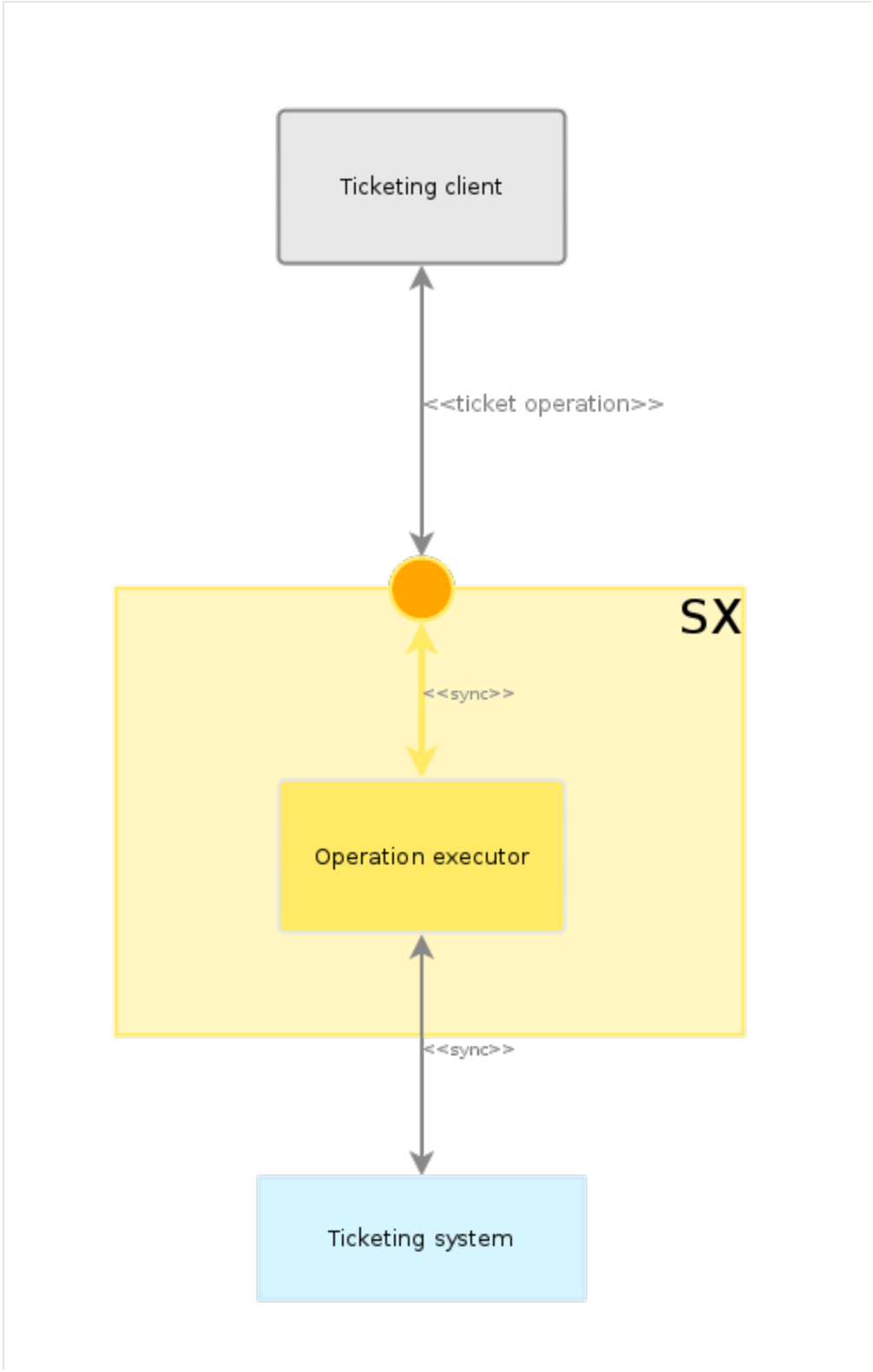
- Ticket managment - this includes routing submit ticket requests to backend systems, listing tickets, adding comments and attachments, closing tickets, watching ticket changes in backend systems.
- Request to fulfill (R2F) - catalog based self-service fulfillment.
- Case Exchange (CX) - exchanging records across multiple systems to facilitate collaboration across silos.

Ticketing

HP SX provides a public Ticketing API to allow clients to interact with ticketing systems such as JIRA, Bugzilla, and HP SM. Ticketing use case requires that the ticketing API is synchronous. This is the only the case where the adapter does not interact through AMQP. The client sends a request to HP SX and waits for the response.

The request is received by HP SX, passed to the *operation executor* component which then executes the given operation against a remote system. The ticketing client using HP SX (for example Propel portal) does not to need know anything about the backend ticketing system's API and data format, it communicates with HP SX via the HP SX unified data format. The data transformation definitions are part of the content pack. HP SX also makes it possible to define ticket properties in the content pack i.e. each backend system type defines its ticket properties.

This use case does not require any interaction state modeling in OO flows. For more details about the API see SX API Docs. For more info about ticketing go to Ticketing use case.



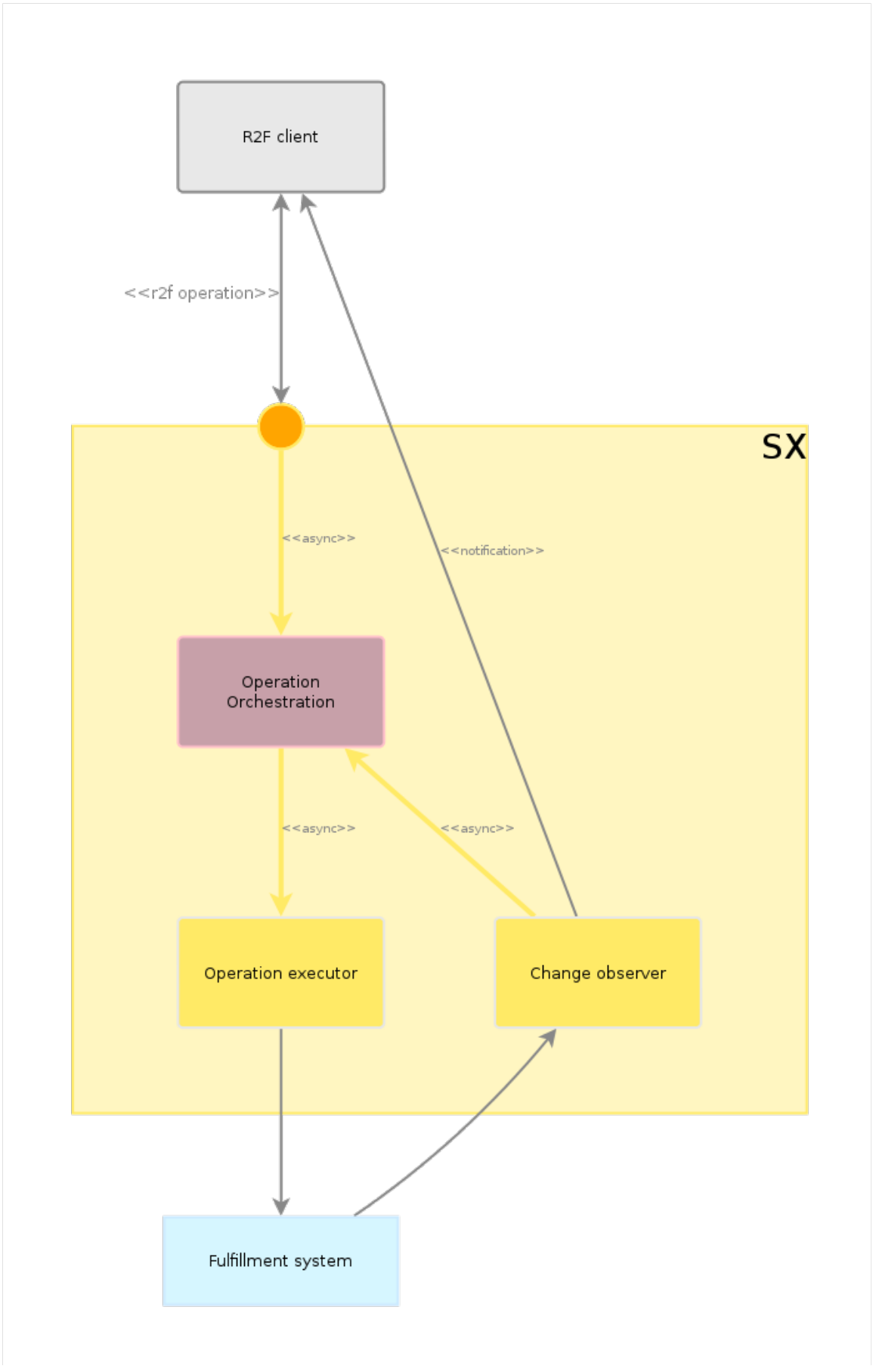
Request to fulfill (R2F)

HP SX provides a public R2F API so that clients (for example Propel portal) may integrate with any fulfillment system supported by HP SX. Fulfilment system here means a backend system as defined above that participates in R2F use case. R2F, unlike Ticketing, is an asynchronous process. The client sends a request containing all the information needed to fulfill the request (fulfillment system instance id, requested item id ...) and HP SX immediately sends the response with the generated request ID. The client then waits for the incoming notifications from HP SX. HP SX accepts the incoming request and sends it to the Operation Orchestrations server, causing the associated OO flow to execute. The OO flow decides which HP SX operation should be executed based on the input data, and instructs HP SX to run the operation by sending it an

asynchronous message. HP SX executes the operation via its Operation Executor causing an interaction with the fulfillment system and invoking the corresponding operation on the fulfillment system side. After the operation is finished, there might be data changes on the fulfillment system that HP SX and the client are interested in. Here the backend system specific Change Observer component becomes involved, detecting such data changes and processing them when they occur. Each data change is handled by the Change Observer in the following way:

1. A notification about the change is sent to the client.
2. The corresponding OO flow is executed to determine if further action is necessary.

For more details about the SX API, see SX API Docs.

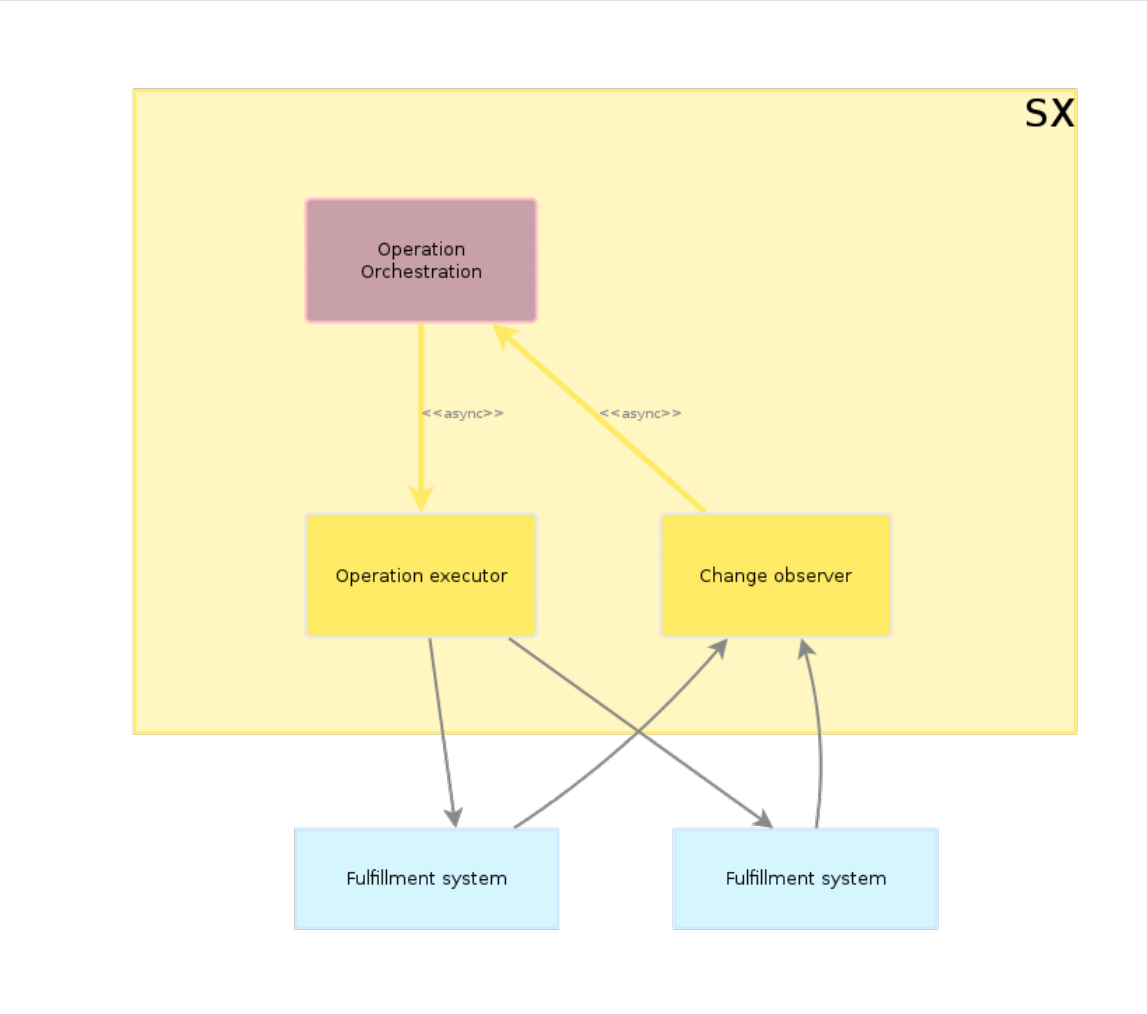

R2F client
<<r2f operation>>
SX
<<async>>
<<notification>>
Operation
Orchestration
<<async>>
<<async>>
Operation executor
Change observer
Fulfillment system

Case Exchange (CX)

HP SX provides a framework to implement automatic data exchange between two or more backend systems.

The typical implementation of this usecase in HP SX enables automatic incident delegation from one service desk system into another service desk system. The automatic delegation here means new linked incident creation in the other service desk system. The linked incidents are then kept aware of each others updates and their states are changed based on the changes occurring in the linked incidents.

CX workflow is similar to R2F workflow except there is no initial request from a client (for example Propel portal.) The initiation comes from a defined change in the backend system that HP SX is configured to watch for. CX boots when HP SX is booted. Special CX components - Change Observers - listen for changes in specified backend systems, and for each change they execute the corresponding OO flow which determines the sequence of operations to be executed. The main difference between R2F and CX is that in CX there is no client request and the CX OO flows have a different logic in them. For more details see Case Exchange documentation.



HP SX lifecycle

In the following it is presumed that HP SX prerequisites are fulfilled. They include:

- running RabbitMQ
- running HP OO

An HP SX instance can be in one of the following stages:

- Starting - in this stage, the following actions are performed:
  - checking if important RabbitMQ queues exist and creating them if they do not
  - starting of RabbitMQ listeners
  - loading and parsing of all available configuration files
  - loading and initialization of content packs

        - content pack initialization includes the check for oo flow versions and possibly the uploading of a new oo flow.
- Running
    - instance is initialized and all APIs listen for incoming requests
    - all change observers wait for events in fulfillment systems.
    - content packs are reloaded dynamically this applies to HP OO flows too they are redeployed into HP OO with the content pack upload.
- Closing
    - closing of RabbitMQ listeners
    - shutdown.
- Stopped - this stage is important for configurations that cannot be loaded dynamically within the running state. The examples follow
    - configure HP SX infrastructure (RabbitMQ, HP OO, SMTP server)
    - configure backend system instances
    - deploy new adapters for backend systems

Adapters in HP SX

Adapters in HP SX

This chapter is an introduction to the HP SX adapter API. You should be able to implement your own custom adapter after reading it.

The main role of an adapter is that it adapts an existing backend system to HP SX paradigms.These are the main components of each adapter:

- **com.hp.ccue.serviceExchange.adapter.Adapter implementation**: adapter interaction manager, capable of boot/shutdown
- **pipeline**: pipelines are used for the processing of AMQP messages, are made of blocks any of which might be custom blocks. Generic blocks are provided.
- **change detector**: detects changes in the backend system and initiates further processing
- **operation executor**: influences operation execution (from operations.json)
- **case exchange adapter**: if you want to enable the backend system for case exchange functionality.

All components except the **com.hp.ccue.serviceExchange.adapter.Adapter implementation** (which glues them together) are optional. But before leaving out a particular component, make sure that it will not be needed once the adapter is booted. Most likely the Operation Executor will be needed - without it an adapter is not able to do anything. There are cases where the pipeline is not needed, mostly where adapters execute operations on request, not as a result of AMQP messages received (these are ticketing adapters.)

Working queues in HP SX

There are three working queues in HP SX :

- CN - queue for submitting catalog notifications. Messages published to this queue are forwarded to catalog (catalog notification.)
- OO - queue for submitting OO flow requests. Messages published to this queue are fowarded to OO (Operations Orchestration engine.)
- SX - queue for passing messages to adapter pipelines. Messages come from: OO, /operation RESTful endpoint and change observers/pollers.

The logic behind CN and OO queues could be invoked directly (the logic which is performed by queue consumers/listeners), but most of the time the natural *asynchronicity* is not a problem. Actually it is desired, and simplifies the whole program workflow. The listener on the SX queue first determines which adapter is interested in the received message, and then passes it to the adapter.

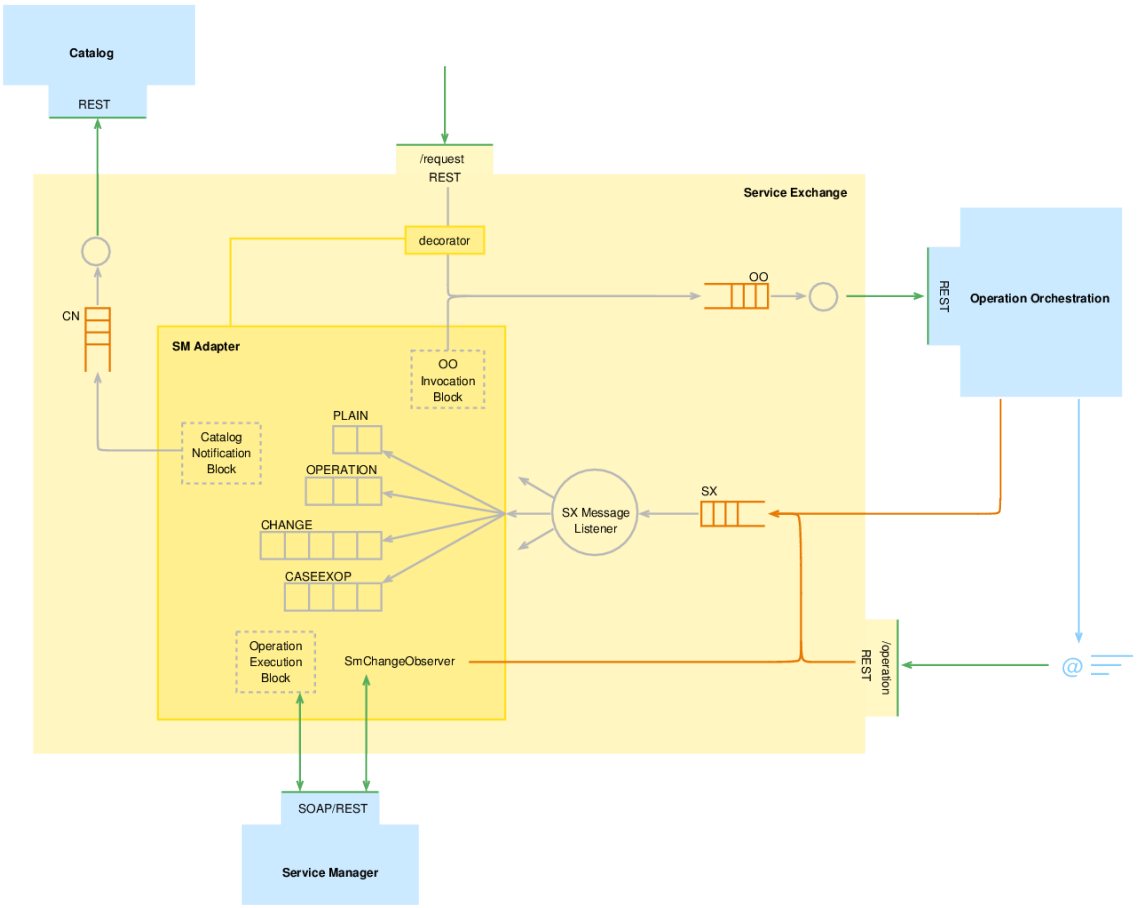
A typical HP SX R2F message flow

To demonstrate the cooperation of the adapter components a description of the message flow for an R2F use case is provided here.

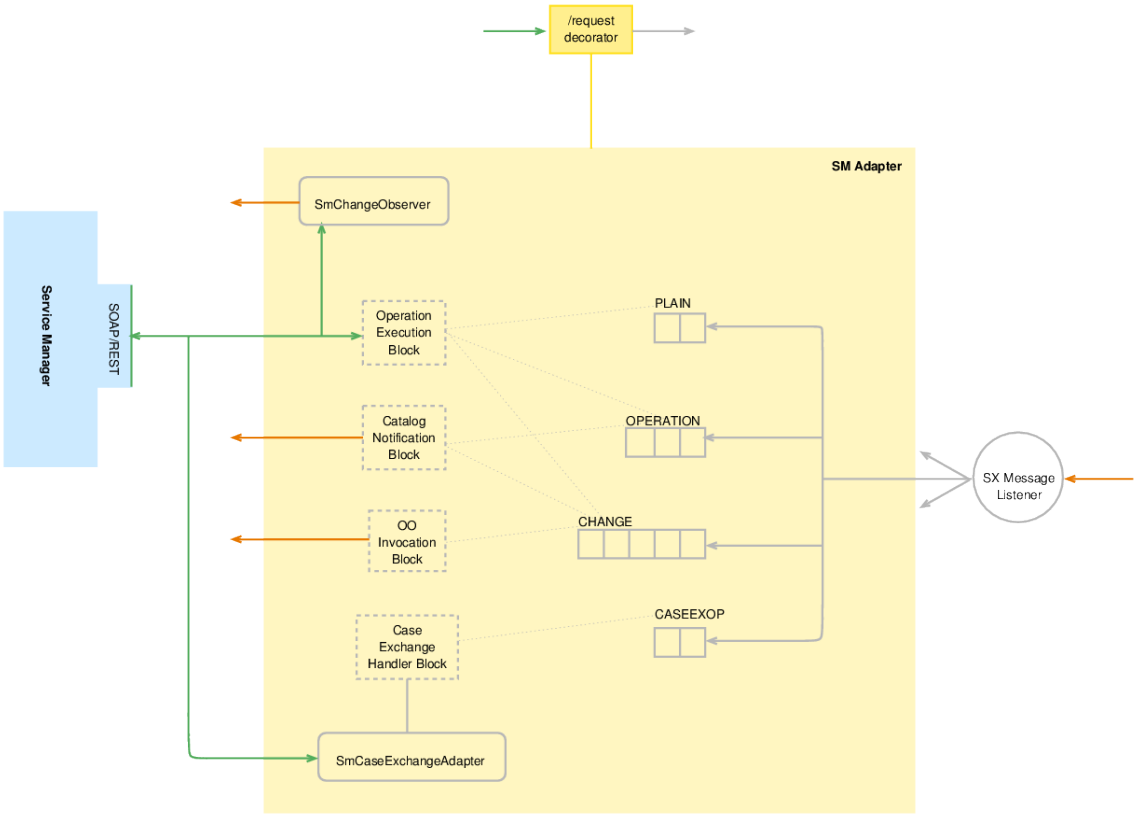
This is the typical flow of an *order* message in HP SX, indicating which queue plays which role:

1. The incoming order request is submitted through the HP SX RESTful interface (/request REST endpoint.) After some decoration it goes directly to the OO queue, and OO decides what to do next.
2. After the message is processed in OO it goes directly to the HP SX queue.
3. The adapter is chosen based on the properties of the incoming message (builtin AMQP property *type*.)
4. The adapter chooses a pipeline for the message processing. For messages from OO, this is usually a *PLAIN* pipeline, which just **executes** an **operation** (via OperationExecutor.) This involves an interaction with the underlying system (for example HP CSA, HP SM) during the operation execution. HP SX remembers that *an entity* is created in the underlying system and sets up constant monitoring of changes of this entity (RESTful polling, RESTful callback notification,...)
5. Whenever a change of the monitored entity occurs, a message is published into the HP SX queue with the type set to '{adapter}:CHANGE'.The message about the change from the HP SX queue is dispatched by executing CHANGE pipeline (based on message properties/type.) This pipeline usually **executes operation**, **notifies catalog** (CN queue), and possibly **invokes OO flow** (OO queue.)
6. [Point 2 extension]: OO might also decide to send an email (approve/reject/close request) instead of publishing a message.
7. Clicking a link in such an email invokes /operation RESTful endpoint, which in turn passes the message to the HP SX queue and sets the message type to '{adapter}:OPERATION'.
8. The OPERATION pipeline usually: **executes operation** and **notifies catalog**.
9. *The steps 2, 3, 4, 5, 6, 7 occur in iterations untill depending on the specific implementation*

This diagram shows the static relationship between queues. It uses the OOB available HP SM adapter as an example:



Here is a detailed diagram of an HP SX adapter, again using the HP SM adapter as an example:



Here you see that an adapter interacts with:

- backend system via REST/SOAP/HTTP (green lines)
- the rest of SX via AMQP queues (orange lines.)

com.hp.ccue.serviceExchange.adapter.Adapter

A fully qualified package name is used here in order to clearly differentiate between the adapter and the Adapter class implementation. The Adapter java class processes incoming AMQP messages, which are pre-processed by the SX Message Listener (the classic implementation of an AMQP listener.)

All adapter instances in the system are available via instances of com.hp.ccue.serviceExchange.adapter.AdapterRegistry. This registry is scanned by SX Message Listener in order to find an adapter which is *interested* in the processing of the received message. Each incoming **AMQP message** has its **type** - sample type values are CSA:PLAIN, CSA:CHANGE, SM:OPERATION. These types must be set by any SX queue publishing code (for example OO.) The first part of *type* (before the colon) is called *major type* throughout the API docs. The default implementation of **isInterested()** returns true if major type matches an adapter name. After the adapter is chosen, its processMessage() method is called.

Here is a non-default implementation of the isInterested() method:

```
@Component
public class OperationCentricAdapter extends AdapterAbstract {

    @Override
    public boolean isInterested(String majorType, MessageProperties properties) {
        // this is more complicated - we inspect operation name (not the majorType)
        final String operationName = extractOperationName(properties);
        return getOperationExecutor().isOperationRecognized(operationName);
    }

    ...

}
```

NOTE: The first adapter in the row is ALWAYS SxInternalAdapter (named SX), which is used to implement core HP SX functionality. It is necessary to not pass the message to other adapters if it is an internal message.

Regarding AMQP message processing, the entry point to an adapter is the processMessage() method. It is given:

- the body of the AMQP message parsed as JSON
- AMQP message headers
- an initial data context.

```
public interface Adapter {


    /**
     * Handles the given AMQP message.
     *
     * @param properties message properties
     * @param message AMQP/JSON message to be handled/processed
     * @param initialContextData initial data context, may be null
     *
     * @return execution context after the processing finishes
     */
    public ExecutionContext processMessage(MessageProperties properties, Map<String,
Object> messageBody, @Nullable Map<String, Object> initialContextData);

 ...

}
```

Adapter identification and /request message decoration

An adapter instance in the registry is also identified by its **name** or **system type**. Name and system type are usually the same. Name is used mainly for internal SX identification purposes (e.g. HP SM, HP CSA, EMAIL). System type is directly referenced in an incoming RESTful /request message, under a **system_type** key. The /request message uses the identified adapter for /request message decoration. Sample existing **names /system types** are given here:

- SM/SM
- CSA/CSA
- EMAIL/urn:propel:email

Extending AdapterAbstract

When writing a custom adapter, it is recommended to use the provided implementation `com.hp.ccue.serviceExchange.adapter.provided.AdapterAbstract` (see javadoc). This class takes in constructor:

- pipeline builder: see below.
- name: identifies adapter - make sure it is unique.
- [opt] operation executor: see below.
- [opt] subsystem type: identifies adapter in incoming /request message. Same as *name* if not provided.

An adapter implementation should have its `@org.springframework.stereotype.Component` annotation in order to have it spring-enabled. It might then also `@Autowire` its internal fields.

IMPORTANT: It is crucial to not ommit the execution of `afterPropertiesSet()`. This call registers an adapter to the adapter registry.

Here is a simple working adapter implementation:

```
@Component
public class MyAdapter extends AdapterAbstract {

    @Autowired
    public MyAdapter(MyOperationExecutor operationExecutor, MyPipelineBuilder
pipelineBuilder) {
        super(Constants.MY_ADAPTER_NAME, operationExecutor, pipelineBuilder);

setRequestMessageHeaderTemplate("my-r2f/sx/templates/generateMessageHeader.ftl");
    }
}
```

NOTE: The invocation of setRequestMessageHeaderTemplate() signals that this adapter is interested in /request message decoration. The result of decoration should be aligned with message format expectations as described in SX Messages.

com.hp.ccue.serviceExchange.adapter.pipeline.Pipeline

An adapter uses a *pipeline* for message processing, and can have any number of pipelines. Pipelines are an internal abstraction of an adapter. When choosing the appropriate pipeline, by default the adapter just takes the second part of the *type* AMQP message property and uses it for the pipeline name. NOTE: If the type is CSA:PLAIN the CSA adapter extracts the pipeline name 'PLAIN' - this can be overriden in `getPipelineNameForMessage()`).

AdapterPipelineBuilder

If the pipeline is about to be used for the first time, it is built using the constructor-given AdapterPipelineBuilder (lazy construction). Here is a simplistic `buildPipeline()` method implementation:

```
@Component
public class MyPipelineBuilder implements AdapterPipelineBuilder {

    @Override
    public Pipeline buildPipeline(Adapter adapter, PipelineBuilderFactory factory,
String name) {
        // we can build only a single pipeline: OPERATION
        switch (name) {
        case Names.PIPELINE_OPERATION:
            return buildOperationPipeline(factory);
        default:
            return null;
        }
    }

    private Pipeline buildOperationPipeline(PipelineBuilderFactory factory) {
        Builder b = factory.newBuilder(Names.PIPELINE_OPERATION);
        b.addBlock(...);
        b.addBlock(...);
        b.addBlock(...);
  return b.build();
 }

    ...

}
```

com.hp.ccue.serviceExchange.adapter.pipeline.PipelineBlock

Pipelines are intended for linear executions of steps during AMQP message processing. Pipeline elements are called *blocks* and they are **executed linearly,** see PipelineBlock.execute(). If branching is required, it could be implemented (via isBlockInterested()), but consider first if two pipelines sharing some common blocks might be a better solution.

See pipeline block API javadoc (com.hp.ccue.serviceExchange.adapter.pipeline). There are some provided/prebuilt blocks available in package com.hp.ccue.serviceExchange.adapter.provided (See javadoc.)

com.hp.ccue.serviceExchange.adapter.pipeline.ExecutionContext

Before the pipeline's execute() method is invoked, an ExecutionContext is created. It serves for storing the message's processing state. Any message processing state must be stored into the `ExecutionContext` object.

WARNING: Saving the message processing state to a block member variable might lead to concurrency issues. Pipelines and blocks are intended to be ThreadSafe! Always save the state to the execution context.

There is:

- operation name
- arbitrary java data (map):
    - can be pre-initialized from above
    - does not have to be serializable - you can store here a reference to your action classes
    - the default implementation puts AMQP message headers into the data map
- AMQP message as JSON (map):
    - message/value-centric
    - must be serializable to JSON.

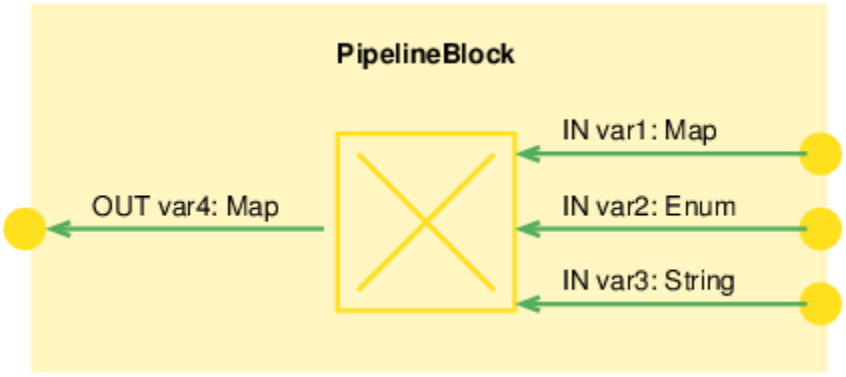
The default implementation of the initial execution context construction is available in the `AdapterAbstract.initContext()` method.

Variables

The order of block execution is determined by their order in the pipeline. Variables are a way to make a block independent, generic, pluggable and thus reusable. There are two kinds of variable:

- **block variable**:
  - has a name (local to a block)
  - is IN or OUT
  - is strongly typed (String, Map,...)

Configured block variables named var1, var2, var3 and var4 are depicted in this diagram:



- **context variable**:
  - points to some place in the execution context (for example 'message' field 'instanceConfig')
  - is strongly typed (String, Map,...)
  - does not have a name but is identified by creating an instance of com.hp.ccue.serviceExchange.adapter.pipeline.ContextVariable
  - once the value is retrieved (looked up in the execution context), it is cached on a specific place in the execution context again
  - uses one of the provided factory methods ContextVariable.newXxx()

Instantiated context variables are depicted in this diagram:

Writing custom blocks

A pipeline block usually does a very simple thing which could be expressed with a few lines of code: it *wraps* a simple Java code. Its uniqueness is that it executes in a pipeline. It therefore needs to adapt pipeline input to input which is expected by the wrapped Java logic. For example:

You want to create a custom block which transforms JSON to JSON via a predefined ftl (Freemarker.) The ftl is the same for all inputs. The Java

code which needs to be executed looks like this:

```
private MessageTransformer mt;

 private final String templatePath;

 ...

 public Map<String, Object> transform(Map<String, Object> messageIn) {
     String transformResult = mt.transformMessage(messageIn, templatePath);
     return JsonUtils.readJsonNothrow(transformResult);
 }
```

To summarize the inputs and outputs needed:

- one input variable typed as Map "messageIn"
- one output variable typed as Map "messageOut"
- one Java member variable (to store template path)
- threadsafe MessageTransformer instance.

When implementing the custom block:

- extend PipelineBlockAbstract: this is the easiest way to implement the custom block
- declare Java fields for inputs common to all invocations
- declare block variables for inputs/outputs specific to the invocation (per ExecutionContext instance)
- bind the block variables to a real place in ExecutionContext (ContextVariable)
- perform the wrapped Java logic in the doExecute() method.

Here is the result of this effort:

MessageTransformBlock

```
package com.hp.ccue.serviceExchange.adapter.provided;

import java.util.Map;
import com.hp.ccue.serviceExchange.adapter.pipeline.ExecutionContext;
import com.hp.ccue.serviceExchange.adapter.pipeline.ContextVariable;
import com.hp.ccue.serviceExchange.adapter.pipeline.impl.BlockVariable.Type;
import com.hp.ccue.serviceExchange.adapter.pipeline.impl.PipelineBlockAbstract;
import com.hp.ccue.serviceExchange.message.MessageTransformer;
import com.hp.ccue.serviceExchange.utils.JsonUtils;

/**
 * Generic message transform block.
 */
public class MessageTransformBlock extends PipelineBlockAbstract {
    // bloc variable names
    public static final String VAR_MSG_IN = "messageIn";
    public static final String VAR_MSG_OUT = "messageOut";

    // members
    private final String templatePath;
    private final MessageTransformer mt;


    protected MessageTransformBlock(MessageTransformer mt, String templatePath,
ContextVariable<Map> messageIn, ContextVariable<Map> messageOut) {
        this.mt = mt;
        this.templatePath = templatePath;
        bindBlockVariable(VAR_MSG_IN, Type.IN, messageIn);
        bindBlockVariable(VAR_MSG_OUT, Type.OUT, messageOut);
    }
    @Override
    public void doExecute(ExecutionContext context) {
        @SuppressWarnings("unchecked")
        String r = mt.transformMessage((Map<String, Object>) getVariable(context,
VAR_MSG_IN), templatePath);
        Map<String, Object> newMsg = JsonUtils.readJsonNothrow(r);
        setVariable(context, VAR_MSG_OUT, newMsg);
    }
}
```

NOTES:

- *bindBlockVariable() method creates the binding between block variable (it creates it) and context variable.*
- *doExecute() method gets/sets the variable value via getVariable() and setVariable().*
- *The entire block is quite universal and therefore easily reusable. The transformation source/target can be customized by providing ContextVariable instances in the Block constructor.*

Default variable binding

If a block variable will be bound to a specific place in the context in most cases, you can also provide *variable default*:

```
/**
 * Generic message transform block.
 */
public class MessageTransformBlock extends PipelineBlockAbstract {

    /**
     * Default input/output message context binding - {@link
ExecutionContext#message}.
     */
    public static final ContextVariable<Map> DEFAULT_MESSAGE_IN =
ContextVariable.newEntireMessage();
    public static final ContextVariable<Map> DEFAULT_MESSAGE_OUT =
ContextVariable.newEntireMessage();

 ...

    protected MessageTransformBlock(MessageTransformer mt, String templatePath,
ContextVariable<Map> messageIn, ContextVariable<Map> messageOut) {
        super(StringUtils.getLastSegment(templatePath, "/"));
        this.templatePath = templatePath;
        this.mt = mt;
        bindBlockVariable(VAR_MSG_IN, Type.IN, messageIn, DEFAULT_MESSAGE_IN);
        bindBlockVariable(VAR_MSG_OUT, Type.OUT, messageOut, DEFAULT_MESSAGE_OUT);
    }

}
```

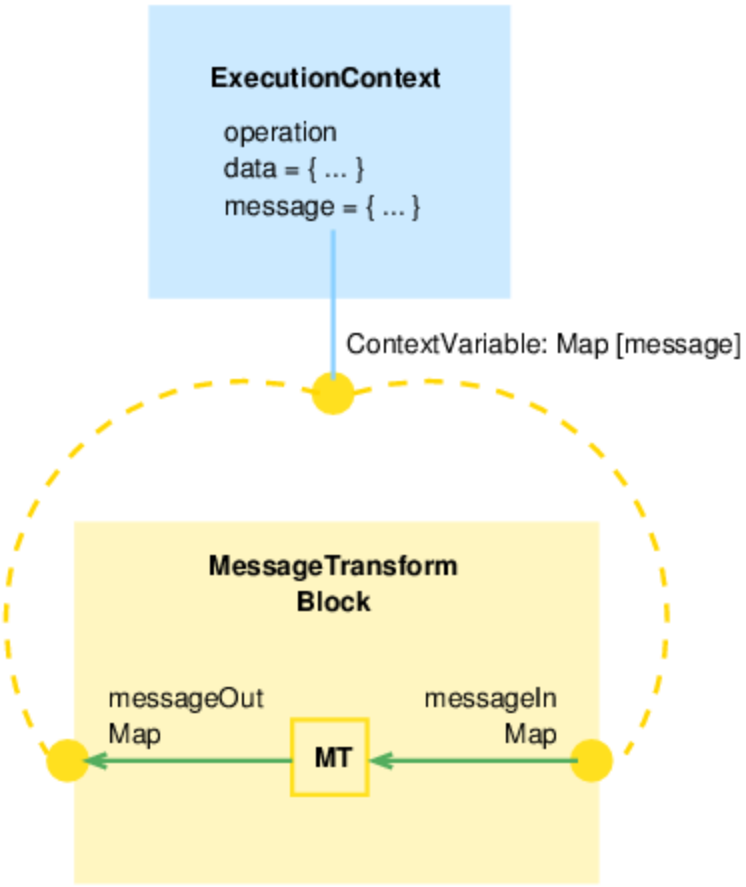
NOTE: The bindBlockVariable() method used checks if the third argument (messageIn, messageOut) is null, and if it is it uses the fourth argument (DEFAULT_MESSAGE_IN, DEFAULT_MESSAGE_OUT), which is considered a default/fallback.

Placing a block into a pipeline

When a block executes, it needs to bind its variables to a real place in an execution context: it needs to bind a block variable to a context variable (connect the yellow dots). Presuming the example block is placed in a pipeline like this:

```
private Pipeline buildMsgTransformPipeline(PipelineBuilderFactory factory) {
        Builder b = factory.newBuilder(Names.PIPELINE_MSG_TRANSFORM);
        // just transform the message
   // we need just one ContextVariable - the input and output are the same
  ContextVariable<Map> ctxMsg = ContextVariable.newEntireMessage();
        // now create/configure the block
        b.addBlock(new MessageTransformBlock(mt,
"sx-r2f/sx/templates/ooMsgToOperationMsg.ftl", ctxMsg, ctxMsg);
   // we are done with pipeline building, the result is stored in ctx.message
        return b.build();
    }
```

The block is now configured to transform the entire message in the context (for both input and output). This is what was done:



*NOTE: By default, the presence of input variables is checked during the validate() routine in PipelineBlockAbstract. If an input variable evaluates to a null value (missing/not set), a validation exception is raised. If an input variable is intended as optional, set a boolean **validate** flag of the bindBlockVariable() method to **false**.*

Connecting several blocks together

To connect two blocks together (i.e. to bind their block variables), bind them to the same context variable. Here is an example:

```
ContextVariable<Map> catalogNotificationMessage =
ContextVariable.newDataMap("cnMessage");
        b.addBlock(new PrepareCatalogNotificationMessageBlock(
                ContextVariable.newFixedValue(RequestState.COMPLETED),
                catalogNotificationMessage));
        b.addBlock(new CatalogNotificationBlock(cnPublisher,
                // notification message
                catalogNotificationMessage,
                // entity ID is 'id' in the message
                ContextVariable.newMessageString(MessageConstants.ID),
                // notification type - always request

ContextVariable.newFixedValue(CatalogNotificationMessagePublisher.NotificationType.REQUES
));
```

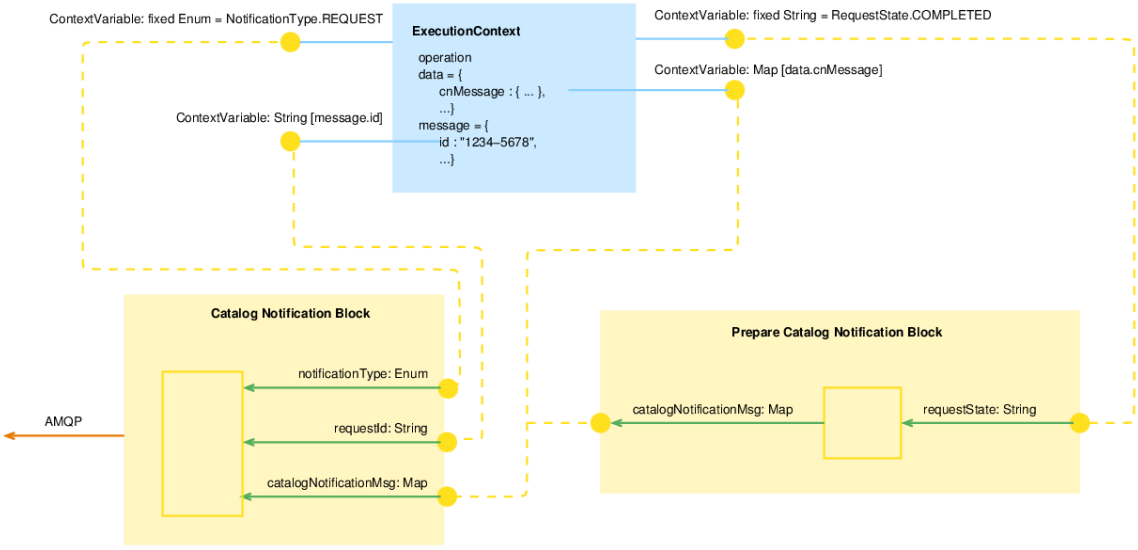
There are two blocks here:

- **PrepareCatalogNotificationBlock**: creates JSON (map) to be published to catalog (notify catalog)
- **CatalogNotificationBlock**: publishes the given message (map) to catalog, providing it also with entity id (string) and notification type (enum).

Clearly the output of PrepareCatalogNotificationBlock needs to be bound to one of the inputs of CatalogNotificationBlock. This is done by declaring the context variable which is passed to both blocks:

- **PrepareCatalogNotificationBlock**: bind to OUTPUT BlockVariable
- **CatalogNotificationBlock**: bind to INPUT BlockVariable.

This diagram depicts the process:



NOTES:

- *The cnMessage property in the data context is initially empty/missing. Once the Prepare Catalog Notification Block sets the value, it is created in the context.*
- *The Catalog Notification Block does not have any output variables, it just initiates catalog notification via AMQP.*
- *If you look at the diagram carefully, you can see that it is possible to define something like a fixedValue ContextProperty. This property would not originate in context values but in your Java code, which is a very useful concept.*
- *Bare in mind that a real PrepateCatalogNotifcationBlock has seven internal variables. Five are omitted in the diagram to keep it simple.*

Change observers

Because of HP SX's asynchronous nature, the adapter needs to detect entity changes in the *backend system*. The detection could be active - polling, or passive - an exposed endpoint receiving notifications. Once the change is detected the adapter acts accordingly. It usually sends an AMQP message to an SX queue to be dispatched by itself, with the type {adapterName}:CHANGE. Despite the fact that change detection can be done in several ways (for example polling or active notification from the backend system), HP SX in particular supports the polling approach in the API.

The change observer implementation needs to implement class `ChangeObserver` (which extends `Runnable` interface):

```
@Component
public class CsaChangeObserver implements ChangeObserver {

    @Value("${adapter.csa.change.observer.interval}")
    private int pollInterval;
    ...
    public int getPollIntervalSec() {
        return pollInterval;
    };
    @Override
    public void run() {
        ...
    }
}
```

NOTES:

- *In the run() routine, the observer should check changes in all configured instances.*
- *Property values have been injected (adapter.csa.change.observer.interval). These properties are taken from `META-INF/adapter.properties`. You can place in this file any custom content and it will be loaded automatically during application boot.*

At this step in the construction of the adapter an instance of change observer needs to be set, so that the adapter schedules or unschedules the observer during the adapter boot and shutdown. Here is an example:

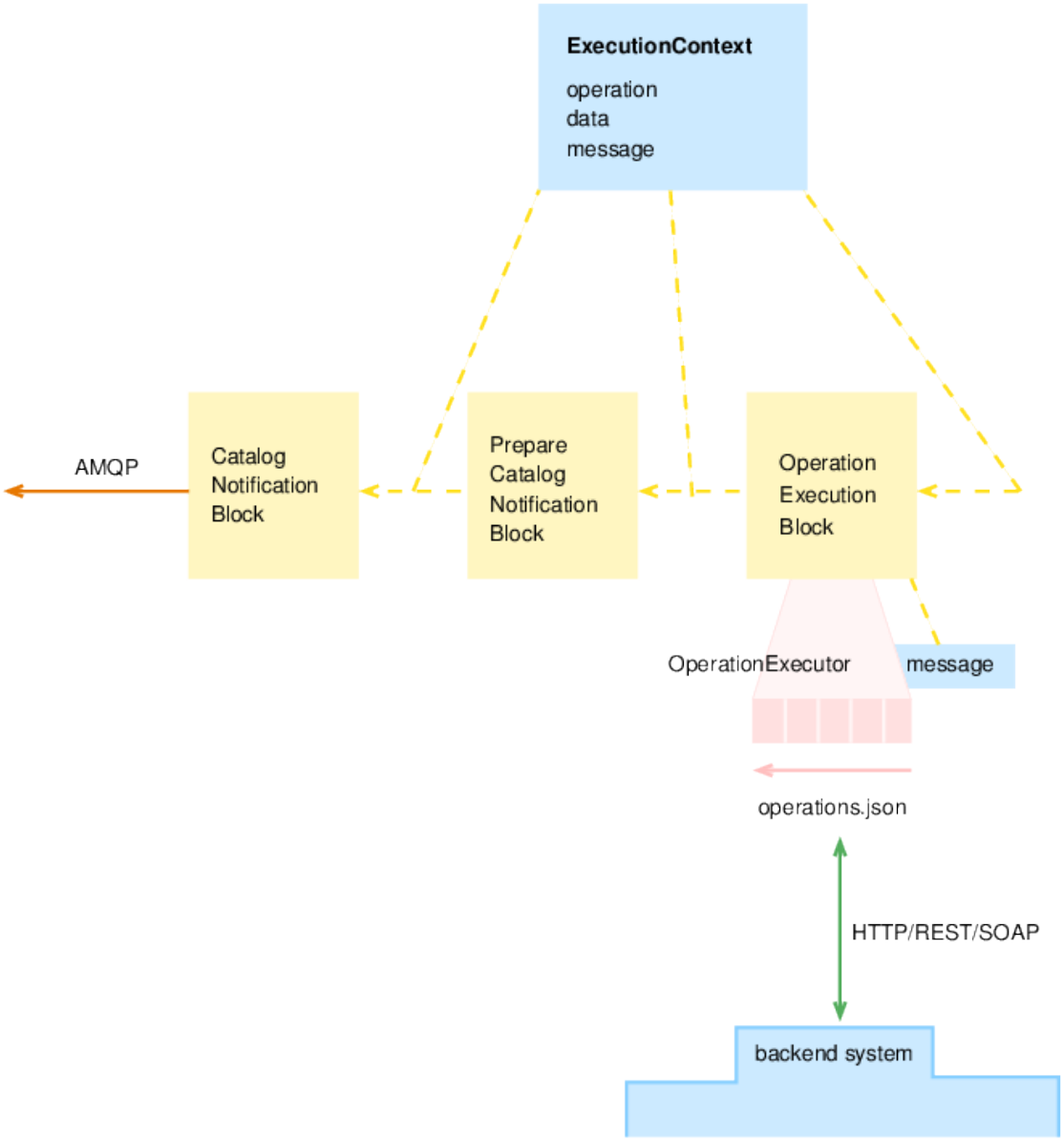
```
@Component
public class CsaAdapter extends AdapterAbstract {

 @Autowired
    public CsaAdapter(CsaOperationExecutor operationExecutor,
   CsaPipelineBuilder pipelineBuilder,
            CsaChangeObserver changeObserver) {
        super(CsaConstants.CSA_ADAPTER_NAME, operationExecutor, pipelineBuilder);
        setChangeObserver(changeObserver);
    }
    ....

}
```

NOTE: Using @Autowired saved some coding.

Operation executors

An adapter is usually accompanied by an *operations.json* file. This file is part of the *content,* containing definiton of operations made of steps to be executed linearly. It is a similar concept to pipelines. However, these operations/steps are mainly concerned with interactions with the backend system and the transformation of sent/received messages. The interaction and message transformation can be easily expressed in operations.json, more easily than in Java. For more on this topic see Operations configuration and templates. The interpretation of operations.json is performed by com.hp.ccue.serviceExchange.operation.**OperationExecutor**.

The OperationExecutor is given a *message* (from the pipeline Execution Context.) The operation execution output (the transformed message) is subject to further pipeline execution. With regard to pipelines, there is a block which is capable of an operation execution called **OperationExecutionBlock**. It wraps the given instance of OperationExecutor and delegates the operation execution to it. A pipeline with an operation execution block might look like this:



For more details on operation executors and how to extend the default HP SX ones, see Appendix B: Operation executors.

Content packs

HP SX Content packs

Overview

Content packs are extension points to HP SX. A typical role of a content pack in HP SX is - in collaboration with adapters - to enable HP SX to communicate with backend systems, for example, HP SM or HP CSA. They also contain the order message lifecycle modeled in HP OO flows in R2F use case.

Technically a content pack is a zipped file containing operation definitions, FreeMarker templates, OO flows and optionally other configuration files. Content packs can be installed/uninstalled into the running HP SX server.

Content Management UI

The HP SX Content Management UI provides an easy interface to view content packs that are currently uploaded in HP SX, download, upload and remove them.

Upload and delete operations include the automatic upload or removal of relevant OO jar files (HP OO content packs), and the merging of HP SX customizations into the running HP SX server.

1. Open the **Content Management** section from HP SX UI.
2. In the **Content Management UI**, view the available content packs with the following details:
    - version numbers
    - which adapter they connect to
    - when last uploaded
    - their high level features
    - the relevant OO content pack name.

Downloading content packs

1. To download a content pack, check the appropriate content pack in the **Id/Name** column.
2. Click the **Download** button.
3. When prompted, **Save** the <*contentpack*>.zip. Depending on your browser settings, select the location through **Save As**… or copy the <*contentpack*>.zip from the Downloads folder to another location. View and customize the files.

Deleting content packs

1. To delete one or more content packs, check the appropriate content packs in the **Id/Name** column.
2. Click the **Delete** button.
3. A confirmation with the number of content packs deleted appears below the buttons on the top of the **Content Management UI**.

Upload content packs

1. To upload a content pack, click the **Upload** button.
2. Locate the .zip or .jar to be uploaded, for example, the **sm-case-exchange.zip** containing a customized **case-exchange.json** file.
3. Select **Open**.
4. It takes a moment for the upload to process. When it is complete a confirmation appears below the buttons on the top of the **Content Management UI** and the Upload Time for the relevant content pack is updated.

Note: When uploading a content pack that was already loaded, HP SX will automatically detect this. The content pack does not need to be selected or specified. HP SX replaces the old version. Content pack are identified by ID attribute provided in their metadata file.

Content Pack Structure

Content packs contain the following folders and files:

- **[oo]** - folder containing the HP OO content pack(s) of custom OO flow(s).
- **[sx]** - folder containing HP SX-specific configuration files.
    - **[templates]** - folder containing Freemarker templates
    - **operations.json** - file containing HP SX operation definitions
    - **flows.json** - file containing the mapping of adapter and message type to OO flow
- **metadata.json** - the content pack description file.

This is an example structure. The structure may differ for specific use cases and adapters. For example HP SM content packs contain an **sm** folder where the customization HP SM unload files are located. Similarly ticketing content packs do not contain OO flows (HP OO content packs) so the **oo** folder and **flows.json** are not present.

File metadata.json

Sample `metadata.json`

Sample metadata.json

```
{
    "id": "sm-r2f",
    "name": "SM request to fullfilment",
    "description": "",
    "version": "1.0.0",
    "adapter": "SM",
    "features":[
        "r2f",
        "sm-r2f"
    ],
 "files": [
        {
            "path": "sm/SXR2FCustomizations.unl",
            "version": "1.01.1",
            "type": "sm_unload"
        },
        {
            "path": "sm/SXR2FDB.unl",
            "version": "1.01.1",
            "type": "sm_unload"
        },
        {
            "path": "sm/SXR2FExtAccess.unl",
            "version": "1.01.2",
            "type": "sm_unload"
        }
    ]
}
```

File **metadata.json** is a description file of the content pack. It contains the following information:

- **id** - content pack's unique ID, limited to 30 characters.
- **name** - content pack name.
- **description** - content pack description.
- **version -** content pack version. IMPORTANT: If you make changes to a content pack increase the version number before upload.
- **adapter** - defines the adapter the content pack is created for.
- **features** - list of the basic HP SX use cases that the content pack supports. When developing your custom content pack specify all use cases that you implement (*r2f, ticketing, case-exchange*)
- **files** - list of adapter-specific files where there is a need to specify a type and a version; currently used for HP SM unload files only. The version specified here is checked by HP SX self-test.

List of currently defined features in HP SX are:

- **r2f** - general requests for fulfilment
- **ticketing**
- **case-exchange**

NOTE: Customers can define new features. This is useful if a custom SX adapter wants to check whether a content pack with a specific feature is deployed.

File sx/flows.json

File `flows.json` contains the mapping of the pair - adapter type and message type to OO flow - and the definition of parameters that will be passed to the OO flow.

Structure of flows.json

```
{
    "<adapterType>": {
        "<messageType>": {
            "flowId": "<flowId>",
            "compressMessage": true|false,
            "parameters": [
                {
                    "name": "<parameterName>",
                    "valueSelector": "<valueSelectorExpression>",
                    "source": "<sourceType>"
                }
            ]
        }
    }
}
```

This file specifies the following:

- **adapterType** - same as the *adapter* in the content pack description.
- **messageType** - HP SX can receive several types of messages. A user can specify different flows for different types - they are matched to the messageType received in http request body on the /reqeust REST endpoint. See Appendix A: Service Exchange - API
- **flowId** - identifier of OO flow that is invoked to process a message. The HP OO content pack containing the flow is located in the **/oo** folder of the content pack.
- **compressMessage** - tells HP SX whether it should compress a message before sending it to OO.
- **parameters** - array of input parameters of flow:
    - **parameterName** - parameter name, the flow receives the parameter with this name during invocation.
    - **valueSelectorExpression** - expression that references the value of the configuration. It uses dot notation to get subproperties (i.e. JsonPath).
    - **sourceType** - source of the value, which can be one of:
        - ***infrastructure*** - find value in configuration file: **sx.war/WEB-INF/classes/config/infrastructure.json**
        - ***oo-properties*** - find value in configuration file: **sx.war/WEB-INF/classes/config/oo/properties.json**
        - ***message*** - find value in the input JSON message to be sent to OO.

Here is an example:

Sample flows.json

```
{
    "SM": {
        "order": {
            "flowId": "95b152da-9666-4c05-883c-593e45bffaa5",
            "compressMessage": true,
            "parameters": [
                {
                    "name": "sxConfiguration.jmsBroker",
                    "valueSelector": "$.JMS_BROKER.endpoint",
                    "source": "infrastructure"
                },
                {
                    "name": "sxConfiguration.smtpServer",
                    "valueSelector": "$.smtpServer",
                    "source": "oo-properties"
                },
                {
                    "name": "orderInfo.id",
                    "valueSelector": "$.orderInfo.id",
                    "source": "message"
                }
            ]
        }
    }
}
```

File sx/operations.json

File `operations.json` contains operation definitions that are interpreted by the adapter's operation executor. An operation is list of step definitions. Invoking an operation means invoking all the steps the operation consists of. The steps are invoked sequentially in the order they are defined.

Structure of operations.json

```
{
    "<operation_1>": [
        <stepDefinition_1>,
        ...
        <stepDefinition_n>
    ],
    "<operation_2>": [
        <stepDefinition_1>
    ]
}
```

Step definition format can differ depending on the adapter type, but the `operationName` must be unique to the adapter type. There is though a base implementation that adapters extend. See Appendix B: Operation executors for the operations definition format provided by the base implementation.

Example:

Sample operations.json

```
{
    "checkSubscription": [
        {
            "label": "Retrieve subscription",
            "requestUrlTemplate": "subscriptionUrl.ftl",
            "responseTemplate": "subscriptionResponse.ftl",
            "method": "GET"
        },
        {
            "label": "Retrieve service instance",
            "requestUrlTemplate": "serviceInstanceUrl.ftl",
            "responseTemplate": "serviceInstanceResponse.ftl",
            "method": "GET"
        },
        {
            "label": "Retrieve root component",
            "requestUrlTemplate": "rootComponentUrl.ftl",
            "responseTemplate": "rootComponentResponse.ftl",
            "method": "GET"
        },
        {
            "label": "Create subscription notification",
            "resultTemplate": "subscriptionNotification.ftl"
        }
    ]
}
```

Custom operations.json files for specific instances

HP SX allows you to change the behavior of any operation for specific instances, by overriding operations. For more info see:

Appendix D: Per instance operation definition

FreeMarker templates

FreeMarker templates are used to transform messages to and from the HP SX JSON format. Templates are written in FTL (FreeMarker template language).

Example of a template transforming the JSON format to XML (In the example it is presumed that an HP SX order message is transformed):

Sample template: JSON to XML

```
<#switch orderInfo.orderType>
    <#case "change">
        <#if orderInfo.subscription.subscriptionId??>
        <Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"><Body>
          <RetrieveSXSubscriptionRequest xmlns="http://schemas.hp.com/SM/7">
            <model>

<keys><subscriptionID>${orderInfo.subscription.subscriptionId}</subscriptionID></keys>
              <instance/>
            </model>
          </RetrieveSXSubscriptionRequest>
        </Body></Envelope>
        </#if>
        <#break>
    <#case "quote">
        <#break>
</#switch>
```

Example of a template transforming XML to the JSON format (The output of this transformation is HP SX order message):

Sample template: XML to JSON

```
<#ftl ns_prefixes={
    "soap":"http://schemas.xmlsoap.org/soap/envelope/",
    "sm":"http://schemas.hp.com/SM/7"}
>
<#switch message.orderInfo.orderType>
    <#case "change">
{
    "orderInfo":{
        "subscription": {
            <#if
doc["soap:Envelope/soap:Body/sm:RetrieveSXSubscriptionResponse/@returnCode"]=="9">
                "status": "Deleted",
                "displayName": ""
            <#else>
                "status":
"${doc["soap:Envelope/soap:Body/sm:RetrieveSXSubscriptionResponse/sm:model/sm:instance/sm
"displayName":
"${doc["soap:Envelope/soap:Body/sm:RetrieveSXSubscriptionResponse/sm:model/sm:instance/sm
</#if>
        }
    }
}
        <#break>
    <#case "quote">
        <#break>
</#switch>
```

Special message directives

HP SX also supports a few special JSON directives. These can influence message processing in the following operation steps when set in the

output JSON message.They are also useful to influence the message processing in the adapter's pipeline blocks that follows the operation executor block executing the operation.

These directives are:

- **skipProcessing** - JSON boolean property. If in the JSON message the step output is skipped, both the following steps of the operation and pipeline processing are skipped.
- **skipProcessingReason** - JSON text property. Describes the reason why the processing was skipped.
- **stopListening** - JSON text property. If set in the JSON message then HP SX will stop listening for changes of this entity. This is mainly used by EntityChangeCleanupBlock (see javadoc).
- **skipFlowRun** - JSON boolean property. If set in the JSON message the OO flow is not called at the end of the operation.

HP OO content packs

HP SX content packs usually come with custom HP OO flows, which are designed in **HP Operation Orchestration Studio**. The output of HP OO Studio is the HP OO Content Pack which is a jar file. This jar file must be put into the HP SX content pack's **oo/** folder. When an HP SX content pack is uploaded to the HP SX server the HP OO Content Packs are automatically deployed on the HP OO server defined in `infrastructure.json`.

IMPORTANT: If you make changes to an HP OO Content Pack you need to manually increase its version number. This is required for the content pack changes to be detected and re-deployed to the HP OO sever. The content pack version is kept in a `contentpack.properties` file that can be found in the HP OO studio project folder.

Function sendMessageToMQ

To send messages from OO flow back to HP SX use function `sendMessageToMQ` from plugin `oo-sx-plugin`. This plugin adds a new message into HP SX's AMQP queue.

This function's input properties are:

- **brokerUrl** - set value to **${sxConfiguration.jmsBroker}** (passed to OO flow after defined in flows.json)
- **brokerUsername** - set value to **${sxConfiguration.jmsBrokerUsername}** (passed to OO flow after defined in flows.json)
- **brokerPassword** - set value to **${sxConfiguration.jmsBrokerPassword}** (passed to OO flow after defined in flows.json)
- **queueName** - assign from input parameter **queueName** (always passed to OO flow by HP SX)
- **operationName** - name of the HP SX operation
- **messageText** - JSON message to be sent as message body
- **messageCompressed** - whether the messageText should be zip compressed.

Limitations

Content packs of the current HP SX version have the following limitations:

Java code can be changed only by writing a new HP SX Adapter

Currently there is no way to customize the Java implementation of existing HP SX adapters. That means if you want, for example, to extend the HP SM adapter with a feature which is totally different from creating an order or from case exchange then you cannot do it without defining a new adapter.

Creating new content packs

Creating a new HP SX content pack involves these steps:

1. Create the content pack structure
2. Define `metadata.json`
3. Define operations:
    a. Declare in `sx/operations.json`
    b. Create FreeMarker templates in `sx/templates/`
4. Create the OO flow:
    a. Design the OO flow in HP OO Studio
    b. Define `sx/flows.json`
5. Zip the content pack and upload to the HP SX server.

The steps in detail:

1. Create content pack structure

Create the content pack folder structure based on Content Pack Structure.

2. Define metadata.json

Create the **metadata.json** file in the root folder of your content pack and fill in the fields according to File metadata.json. Make sure that you choose an ID which is unique in the whole HP SX system. Select the HP SX features your content pack will implement. If you are writing a new content pack for a new adapter, you can also define a new feature (with a unique name).

3. Define operations

First create a new **sx/operations.json** file and according to the specifications in File sx/operations.json define your new operations and operation steps. Then create the referenced FreeMarker templates in the **sx/templates** folder.

4. Create the OO flow

a. If your new content pack requires an OO flow too, use HP OO Studio to design the OO flow.
b. When complete, export your design into the HP OO content pack (which is a jar file), and copy that file into the oo/ folder of the content pack. Use the **Create new content pack** menu item in HP OO Studio.
c. Next, map your new OO flow to an existing adapter and message type in the `sx/flows.json` file. Create the file according to `sx/flows.json` specifications.
d. Set the **flowId** to the ID of your new OO flow (find it in the root element of the flow XML file, or the **Properties** tab in HP OO Studio), and fill the input parameters to be passed to the OO flow.

5. Zip the content pack and upload

Zip your content pack into one file and using the HP SX Content Management UI upload it.

Creating content packs in development

When developing your content you do not need to follow the above manual process. A more convenient approach for the content developer is the setup maven build that uses HP OO SDK maven plugin to create the HP OO content pack, and packs it together with SX files into the HP SX content pack archive. See Creating an HP SX content module in How to extend HP SX Content (HP SM Problem entity) for a detailed description of how to setup such a maven project, or see the example content implementations provided with the SDK package.

See also

How to extend HP SX Content (HP SM Problem entity)

SX HP OO plugin

SX HP OO plugin

Introduction

SX HP OO plugin (oo-sx-plugin artifact provided in SDK's maven repository) is a key component for OO flows in HP SX. It provides a way for the OO flow to interact with SX by sending a message into SX Rabbit AMQP.

It is an extension to HP OO, implementing a single `sendMessageToMQ` operation. Its parameters are listed in the table below:

parameter	description
brokerHostname	JMS broker URL. The Rabbit AMQP used by your HP SX.
brokerUsername	Rabbit AMQP user.

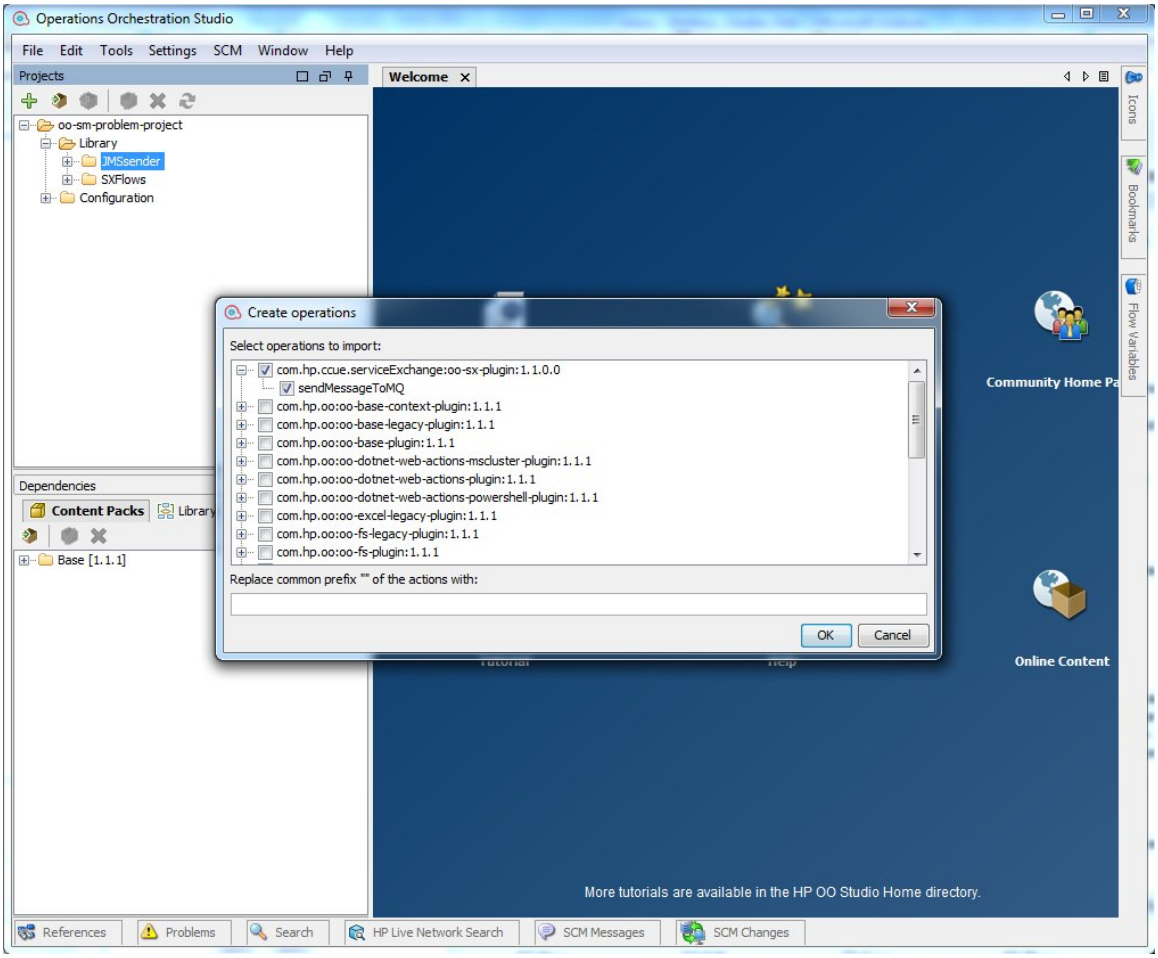
brokerPassword	Rabbit AMQP password.
queueName	Name of the Rabbit queue the message will be sent to, it is based on the configuration in `sx.properties`.
messageType	Usually in the format 'ADAPTER_NAME:TYPE', typically this parameter is used to determine the pipeline that will process the message (for example SM: PLAIN will be handled by PLAIN pipeline.)
operationName	Optional - put into AMQP message header when there is content.
messageText	The message itself when compression is not used (development.)
messageCompressed	The message itself Base64 encoded.

The parameters brokerHostname, brokerUsername, brokerPassword, queueName are HP SX configuration parameters.

Configuring the SX HP OO plugin in HP OO Studio

To use the `sendMessageToMQ` plugin operation in HP OO Studio when designing your OO flows, follow these steps.

1. Add the plugin into the maven repository of your HP OO Studio instance. The simplest method is to copy the contents of the m2-repo directory from your SDK package into the HP OO Studio repository: (`<your_home_dir>/.oo/data/maven`).
2. Perform the operation import into your specific OO flow project.
    a. Under the **Library** folder of your OO project, create a new folder for the operation. It is named JMSsender in the example below.
    b. In the created folder right click **> New > Operation**.
    c. Find com.hp.ccue.serviceExchange:oo-sx-plugin and sendMessageToMQ in the list. See the screenshot below.
    d. Click **OK**.



Case Exchange

Overview

CX is a subsystem of HP SX, designed for exchanging entity data between two or more external systems. The aim is to have some entity data, for example Incidents, automatically synchronized between two different systems without the need for human intervention.

CX does all the work of data transformation including connecting systems of different types, for example HP SM and HP SAW. In addition, CX removes the need to setup the two systems to communicate directly with each other, which helps simplify the security and environment setup. Instead of having to provide an adapter for each possible system-type pair combination, it is sufficient to implement CX between system A and HP SX, and system B and HP SX.

CX works in the following way:

1. A pairing between source and target system is defined.
2. The source system is observed for changes CX is interested in.
3. Once an interesting entity change is detected (Creation, Update, Status change), CX performs the following:
    a. Retrieves any important entity data from the source system.
    b. Transforms the entity data to the canonical model representation.
    c. Changes the data of a connected entity on a target system in the way defined by the configuration.

Example:
There is an HP SM instance called SM03 and an HP SAW instance called SAW02.

To set up CX to clone any new Incident created on SM03 to SAW02 systems:

1. Create a CX pairing (see External systems and entities pairing) between SM03 and SAW02, where SM03 is a source system and SAW02 is the target system.
2. Set up cloning of new incidents for the pairing.

Once finished with the configuration, any new Incident created on SM03 is automatically cloned to SAW02.

When a new system type adapter (for example for Remedy) is being written, the adapter can be implemented to support CX. What needs to be done to accomplish this is described in the following example, see Case exchange use case. Once CX is enabled for the new system type, it can participate in CX together with all the other systems supporting CX, without any need to change any existing systems.

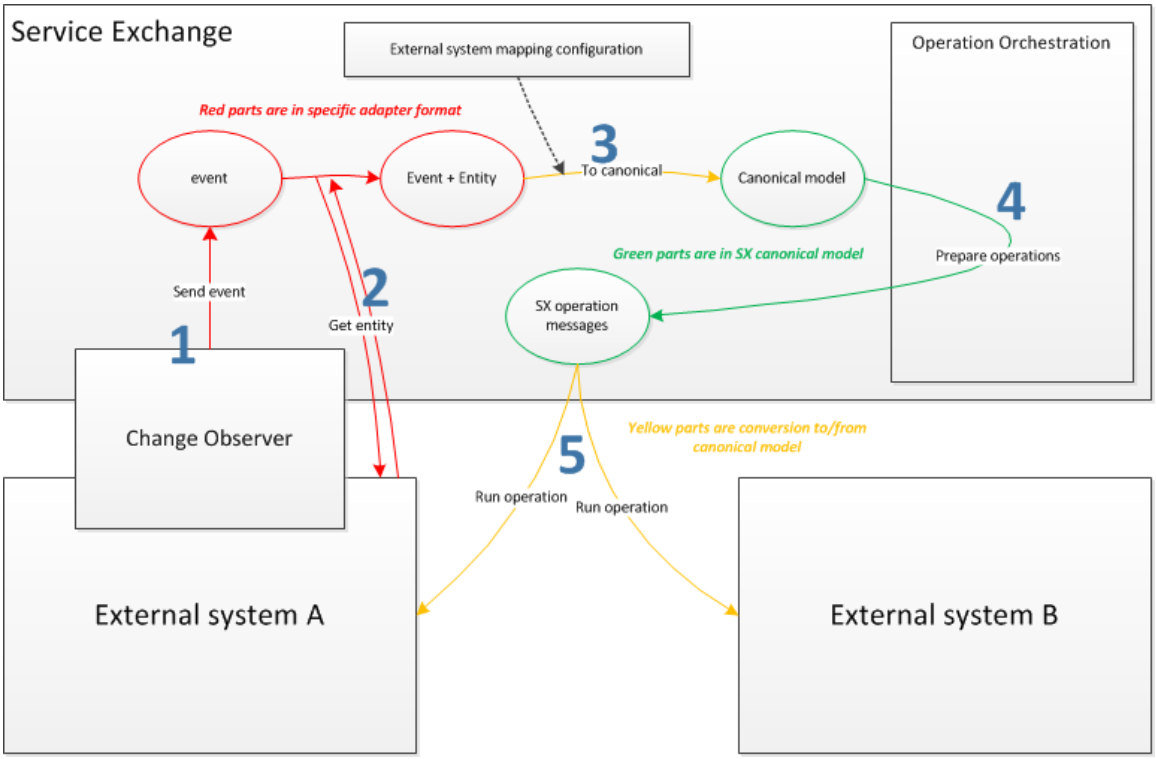
Concepts

- Canonical Model
    - Canonical Model Format
        - Common canonical model parts
        - Entity specific canonical model parts
    - Canonical Model Translation
        - Example mappings
            - ES property name -> SX property name
            - ES property value -> SX property value
            - external system alias name -> SX external system name
            - SX entity name -> ES entity name
            - SX property name -> ES property name
            - SX property value -> ES property value
- Single or Multiple linkedEntity
- External systems and entities pairing
- Change Observation
- Symmetry

Canonical Model

HP SX can exchange entities (incidents, tasks, etc) between different external systems (HP SM, HP SAW, JIRA, etc.) Every system uses its own entity format. HP SX translates from these formats to a universal HP SX format - a canonical model. The canonical model contains information about entities, external systems and

event.



How it works:

A change observer detects a change in external system A. It creates an event, for example, incidentResolved. The format of an event message is adapter/system specific. A set of possible events is the same for one entity type (for example, Incident.)

1. HP SX retrieves the entity from system A. The entity is in an adapter-specific format.
2. HP SX translates the entity + event message + information about external systems to the canonical model.
3. The OO flow decides which operations will be called in external systems A and B.
    - The most frequent scenario:
    a. Copy some properties of the entity to system B (first operation.)
    b. Update the external status of entity in system A (second operation.)
    - Operation messages contain two main parts: entity and linkedEntity both represented in the canonical model.
        - The entity part contains information about the source system (A), and about the entity in the source system.
        - The linkedEntity part contains information about the target system (B), and the entity in the target system
4. HP SX runs the operations. The canonical model is transformed to the adapter-specific format during the operations.

Canonical Model Format

Canonical model Example

```
{
    "event":"incidentResolved",
    "entity":{
        "instanceType":"SAW",
        "instance":"msalb003sngx",
        "entityType":"Incident",
        "entityId":"19423",
        "properties":{
            "Title":"Window was broken",
            "Description":"An angry employee broke the window",
            "Urgency":"U3",
            "Status":"Resolved",
            "Impact":"I4",
            "Solution":"We buy new window and installed it.",
            "CompletionCode":"Resolvedbyfix",
            "Attachments":[
                {
                    "id": "3987",
                    "name": "window",
                    "type": "image/png",
                    "size": 723454
                },
                {
                    "id": "3987",
                    "name": "window_detail",
                    "type": "image/png",
                    "size": 901211
                }
            ],
            "Comments":[
                {
                    "id": "121",
                    "author": "jim.breaker",
                    "content": "I send you picture of the window."
                }
            ]
        }
    },
    "linkedEntity":{
        "instanceAlias":"supportSM",
        "instanceType":"SM",
        "instance":"mpavmsm12",
        "entityType":"Incident",
        "entityId":"IM12391",
        "properties":{
            "Status":"Complete"
        }
    }
}
```

Common canonical model parts

These parts are the same for all entities (incidents, tasks, problems, ...). All fields except "linkedEntity.entityId" are mandatory. Each entity type is a common name for the entity, for example probsummary in HP SM is not represented as probsummary, but Incident, as this is the canonical

model system-independent name for the entity.

json field	description
event	Event name must exist in a set of events for the given entity type.
entity	Describes the source system and entity which was changed.
entity.instanceType	Type of source system/adapter (e.g. HP SM, HP SAW, JIRA.)
entity.instance	ID of the specific source external system. HP SX has mapping from this ID to URLs for communication with the system.
entity.entityType	Entity type (for example, Incident, Task, Problem.) Every entity type has its own set of events and specific properties.
entity.entityId	ID of entity in source system.
linkedEntity or linkedEntities	Describes the entity and target system, where the entity will be cloned or refreshed.
linkedEntity.instanceAlias	User of source system uses this alias to identify the target system, HP SX has mapping between aliases and real external systems.
linkedEntity.instanceType	Type of target system/adapter (for example, HP SM, HP SAW, HP JIRA.)
linkedEntity.instance	ID of the specific target external system. HP SX has a mapping from this ID to URLs for communication with the system.
linkedEntity.entityType	Entity type in the target system. It can be different from entity.entityType. For example: Incidents from HP SAW are cloned as Tasks to HP SM.
linkedEntity.entityId	ID of entity in the target system. It can be empty before cloning.

Entity specific canonical model parts

These parts are specific to each entity (incident, task, problem, ...)

json field	description
entity.properties	Set of properties, specific for a given entity type.
entity.properties.attachments	List of attachments which were added from the last event. HP SX copies it to the target external system. Now attachments are used for the Incident entity type only, but it can be used elsewhere.
entity.properties.attachments.id	ID of the attachment in the external system.
entity.properties.attachments.name	Name of the attachment.
entity.properties.attachments.type	MIME type, for example, "image/gif" or "application/xml".
entity.properties.attachments.size	Attachment size in bytes.
entity.properties.comments	List of comments which were added from the last event. HP SX copies them to the target external system. Now comments are used for the Incident entity type only, but they can be used elsewhere.
entity.properties.comments.id	ID of the comment in the external system.
entity.properties.comments.author	Author of the comment.
entity.properties.comments.content	The comment.

linkedEntity.properties	Set of properties. They can be specific for a given entity type, but typically there is only "Status". These properties are only copied to external reference fields, see pairing.

HP SX content developer note: If you add a new adapter (for example Bugzilla), but you work with an existing entity (for example **incidents**) you have to conform to both common and entity-specific parts of the canonical model. If you are newly adding support for an entity (for example **change requests**), and there is no canonical model specified for it yet, you need to specify the entity-specific part of the canonical model.

Canonical Model Translation

HP SX translates entity names, property names, property values and external system names *to* and *from* the canonical model.

The following table shows where and how HP SX does these translations (note that for simplicity, ES replaces 'external system' in the table):

Translation	Translation executed by ...	Example
ES entity name -> SX entity name	ChangeObserver.	JIRA ChangeObserver listens for change of issues and creates an Incident event
ES property name -> SX property name	convert*ToCanonicalModel freemarker template.	see example
ES property value -> SX property value	convert*ToCanonicalModel freemarker template. It uses *-mappings.json configuration.	see example
external system alias name -> SX external system name	convert*ToCanonicalModel freemarker template. The template uses the FindExternalSystemForAlias class.	see example
SX entity name -> ES entity name	freemarker template during running operation.	see example
SX property name -> ES property name	freemarker template during running operation.	see example
SX property value -> ES property value	freemarker template during running operation. It can use *-mappings.json configuration.	see example
SX external system name -> external system alias name	freemarker template, before it updates external references in the source system. The template uses the FindAliasForExternalSystem class.	prepareInputAfterOperation.ftl in content-sm-case-exchange

Example mappings

ES property name -> SX property name

convertIncidentToCanonicalModelResult.ftl

```
<#assign
loadConfig='com.hp.ccue.serviceExchange.adapter.freemarker.LoadConfig'?new()/>
<#assign
findKey='com.hp.ccue.serviceExchange.adapter.freemarker.FindKeyForValue'?new()/>
<#assign
findExtSystemForAlias='com.hp.ccue.serviceExchange.caseex.freemarker.FindExternalSystemFo
/>
<#assign sawMapping=loadConfig(context.contentStorage,
"saw-case-exchange/saw-mappings") />
<#assign
findExtSystemForAlias='com.hp.ccue.serviceExchange.caseex.freemarker.FindExternalSystemFo
/>

<#assign secondExternalSystem=findExtSystemForAlias(context.appContext,
entityChange.instanceType, entityChange.instance, data.externalInstanceAlias)!"" />
{
  "event": "${message.entityChange.changeReason}",
  "entity": {
    "instanceType": "${message.entityChange.instanceType}",
    "instance": "${message.entityChange.instance}",
    "entityType": "Incident",
    "entityId": "${message.entityChange.entityId}",
    "properties": {
        "Title":"${message.entityChange.data.response.properties.DisplayLabel}" //
property DisplayLabel in SAW format is mapped to property Title in canonical model
        ,"Description": "${message.entityChange.data.response.properties.Description}"
        ,"Urgency": "${findKey(sawMapping.Incident.Urgency,
message.entityChange.data.response.properties.Urgency)}"
        ,"Status": "${findKey(sawMapping.Incident.Status,
message.entityChange.data.response.properties.Status)}" // property Status has same
name in SAW format and canonical model
  }
  },
  "linkedEntity": {
    "instanceAlias": "${message.entityChange.data.externalInstanceAlias}",
    "instanceType": "${secondExternalSystem.targetInstanceType}",
    "instance": "${secondExternalSystem.targetInstance}",
  },
}
```

ES property value -> SX property value

It translates by the convertIncidentToCanonicalModel freemarker template, and uses the saw-mappings.json configuration.

convertIncidentToCanonicalModelResult.ftl

```
//function for loading config from content pack
<#assign
loadConfig='com.hp.ccue.serviceExchange.adapter.freemarker.LoadConfig'?new()/>
//load value mapping config (SAW<->SX) to hashMap
<#assign sawMapping=loadConfig(context.contentStorage,
"saw-case-exchange/saw-mappings") />
//function for finding key by value in hashMap
<#assign
findKey='com.hp.ccue.serviceExchange.adapter.freemarker.FindKeyForValue'?new()/>
<#assign
findExtSystemForAlias='com.hp.ccue.serviceExchange.caseex.freemarker.FindExternalSystemFo
/>
<#assign
findExtSystemForAlias='com.hp.ccue.serviceExchange.caseex.freemarker.FindExternalSystemFo
/>

<#assign secondExternalSystem=findExtSystemForAlias(context.appContext,
entityChange.instanceType, entityChange.instance, data.externalInstanceAlias)!"" />
{
  "event": "${message.entityChange.changeReason}",
  "entity": {
    "instanceType": "${message.entityChange.instanceType}",
    "instance": "${message.entityChange.instance}",
    "entityType": "Incident",
    "entityId": "${message.entityChange.entityId}",
    "properties": {
        "Title":"${message.entityChange.data.response.properties.DisplayLabel}"
        ,"Description": "${message.entityChange.data.response.properties.Description}"
        ,"Urgency": "${findKey(sawMapping.Incident.Urgency,
message.entityChange.data.response.properties.Urgency)}" //convert Urgency SAW value
to Urgency SX value by sw-mapping.json config
        ,"Status": "${findKey(sawMapping.Incident.Status,
message.entityChange.data.response.properties.Status)}"//convert Satus SAW value to
Status SX value by saw-mapping.json config
  }
  },
  "linkedEntity": {
    "instanceAlias": "${message.entityChange.data.externalInstanceAlias}",
    "instanceType": "${secondExternalSystem.targetInstanceType}",
    "instance": "${secondExternalSystem.targetInstance}",
  },
}
```

saw-mapping.json

```
{
    "Incident": { //property value mapping for entity Incident
        "Status": { //mapping for Status property, key is value in SX, value is value
in SAW
            "Open": "Ready",
            "WorkInProgress": "InProgress",
            "PendingChange": "Pending",
            "PendingOther": "Suspended",
            "Complete": "Complete"
        },
        "Urgency": { //mapping for Urgency property, key is value in SX, value is
value in SAW
            "U4": "NoDisruption",
            "U3": "SlightDisruption",
            "U2": "SevereDisruption",
            "U1": "TotalLossOfService"
        }
    }
}
```

external system alias name -> SX external system name

It translates by the convertIncidentToCanonicalModel freemarker template, and the template uses the FindExternalSystemForAlias class. The class is found in external-system.json.

convertIncidentToCanonicalModelResult.ftl

```
<#assign
loadConfig='com.hp.ccue.serviceExchange.adapter.freemarker.LoadConfig'?new()/>
<#assign
findKey='com.hp.ccue.serviceExchange.adapter.freemarker.FindKeyForValue'?new()/>
<#assign
findExtSystemForAlias='com.hp.ccue.serviceExchange.caseex.freemarker.FindExternalSystemFo
/>
<#assign sawMapping=loadConfig(context.contentStorage,
"saw-case-exchange/saw-mappings") />
//this method find in of second external system by its alias in first external system
<#assign
findExtSystemForAlias='com.hp.ccue.serviceExchange.caseex.freemarker.FindExternalSystemFo
/>
//find id of second external system by its alias in first external system
<#assign secondExternalSystem=findExtSystemForAlias(context.appContext,
entityChange.instanceType, entityChange.instance, data.externalInstanceAlias)!"" />
{
  "event": "${message.entityChange.changeReason}",
  "entity": {
    "instanceType": "${message.entityChange.instanceType}",
    "instance": "${message.entityChange.instance}",
    "entityType": "Incident",
    "entityId": "${message.entityChange.entityId}",
    "properties": {
        "Title":"${message.entityChange.data.response.properties.DisplayLabel}"
        ,"Description": "${message.entityChange.data.response.properties.Description}"
        ,"Urgency": "${findKey(sawMapping.Incident.Urgency,
message.entityChange.data.response.properties.Urgency)}"
        ,"Status": "${findKey(sawMapping.Incident.Status,
message.entityChange.data.response.properties.Status)}"
  }
  },
  "linkedEntity": {
    "instanceAlias": "${message.entityChange.data.externalInstanceAlias}",
    "instanceType": "${secondExternalSystem.targetInstanceType}",//using of found
secondExternalSystem by alias
    "instance": "${secondExternalSystem.targetInstance}",//using of found
secondExternalSystem by alias
  },
}
```

external-systems.json

```
 {//this file is used by findExtSystemForAlias method from
convertIncidentToCanonicalModelResult.ftl
  "externalSystemAliases": [
    {
      "sourceInstanceType": "SAW",
      "sourceInstance": "msalb003sngx",
      "targetInstanceType": "SM",
      "targetInstance": "mpavmsm10",
      "targetAlias": "SM"
    }
  ]
}
```

SX entity name -> ES entity name

resolvedIncidentUrl.ftl

```
<#escape x as x?url>
<#noescape>${instanceConfig.endpoint}</#noescape>
/9/rest/sxce_incidents/${message.args.linkedEntity.entityId}/action/update
//sxce_incident url part says that entity Incident in SX is mapped to entity
sxce_incidents in SM
</#escape>
```

SX property name -> ES property name

resolveIncidentRequest.ftl

```
<#assign
loadConfig='com.hp.ccue.serviceExchange.adapter.freemarker.LoadConfig'?new()/>
<#assign smMapping=loadConfig(context.contentStorage, "sm-case-exchange/sm-mappings")
/>
<#-- @ftlvariable name="message" type="java.util.Map" -->
<#assign entity=message.args.entity/>
{
    "Incident" : {
        "Status": "Resolved"
        ,"ClosureCode":
"${smMapping.Incident.CompletionCode[entity.properties.CompletionCode]!""}"
//CompletitionCode property in SX is mapped to ClosureCode property in SM
        ,"Solution": "${entity.properties.Solution}"
        ,"Category":"incident"
    }
}
```

SX property value -> ES property value

resolveIncidentRequest.ftl

```
//function for loading config from content pack
<#assign
loadConfig='com.hp.ccue.serviceExchange.adapter.freemarker.LoadConfig'?new()/>

//load value mapping config (SM<->SX) to hashMap
<#assign smMapping=loadConfig(context.contentStorage, "sm-case-exchange/sm-mappings")
/>
<#-- @ftlvariable name="message" type="java.util.Map" -->
<#assign entity=message.args.entity/>
{
    "Incident" : {
        "Status": "Resolved"
        ,"ClosureCode":
"${smMapping.Incident.CompletionCode[entity.properties.CompletionCode]!""}"//convert
CompletionCode SM value to CompletionCode SX value by sm-mapping.json config
        ,"Solution": "${entity.properties.Solution}"
        ,"Category":"incident"
    }
}
```

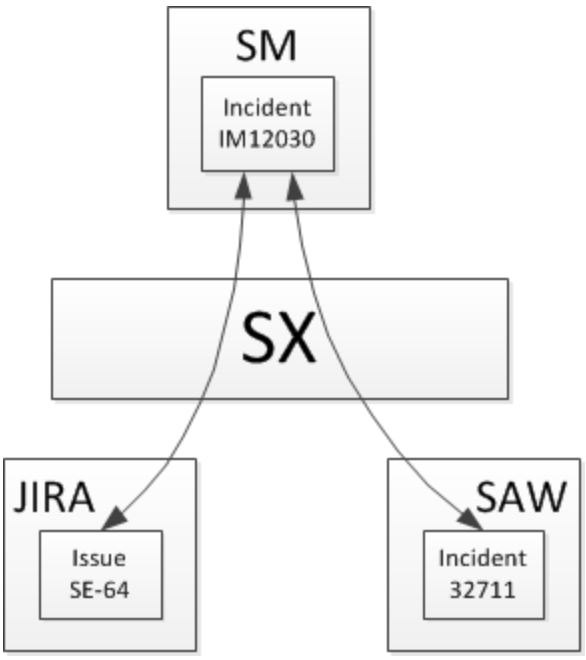
sm-mapping.json

```
{
    "Incident": { //property value mapping for entity Incident
        "CompletionCode": { //mapping for CompletionCode property, key is value in SX,
value is value in SM
            "SuccessfulDiagnosis": "",
            "NoFaultFound": "",
            "NoUserResponse": "",
            "NotReproducible": "Not Reproducible",
            "OutofScope": "Out of Scope",
            "RequestRejected": "Request Rejected",
            "Resolvedbyfix": "Solved by Change/Service Request",
            "ResolvedWorkaround": "Solved by Workaround",
            "UnabletoSolve": "Unable to solve",
            "WithdrawnbyUser": "Withdrawn by User",
            "SolvedByUserInstruction":"Solved by User Instruction",
            "AutomaticallyClosed":"Automatically Closed"
        }
    }
}
```

Single or Multiple linkedEntity

One entity in an external system can be mapped to several entities in other external systems.

Example:



There are two ways to handle this in HP SX. If you write a new adapter you need to choose one method:

1. If an incident changes in system A, ChangeObserver generates two events: one for system B, one for system C. The rest of the scenario is the same. There is still one linkedEntity section in the canonical model. This is the method used in the SAW adapter.
2. If an incident changes in system A, ChangeObserver generates one event. HP SX receives the entity with several external references and transforms them to the canonical model with several linkedEntities. Then HP SX automatically converts it to two canonical models with one linkedEntity and calls the OO flow twice. This is the method used in the SM adapter.

The following example shows the canonical model with several external references:

Canonical model with several external referencies

```
{
    "event":"incidentResolved",
    "entity":{
        "instanceType":"SAW",
        "instance":"mpavmsm12",
        "entityType":"Incident",
        "entityId":"IM12030",
        "properties":{
            "Title":"Window was broken",
            "Description":"An angry employee broke the window",
            "Urgency":"U3",
            "Status":"Resolved",
            "Impact":"I4",
            "Solution":"We buy new window and installed it.",
            "CompletionCode":"Resolvedbyfix",
            "Attachments":[],
            "Comments":[]
        }
    },
    "linkedEntities":[
        {
            "instanceAlias":"externalJIRA",
            "instanceType":"JIRA",
            "instance":"kpj00765a",
            "entityType":"Incident",
            "entityId":"SE-64",
            "properties":{
                "Status":"Complete"
            }
        },
        {
            "instanceAlias":"citSAW",
            "instanceType":"SAW",
            "instance":"msalb003sngx",
            "entityType":"Incident",
            "entityId":"32711",
            "properties":{
                "Status":"Complete"
            }
        }
    ]
}
```

External systems and entities pairing

References from an entity in external system A to an entity in external system B are not stored in HP SX, they are stored in the external systems. HP SX needs **at least two fields** on an entity: "external_system_alias" and "external_entity_id". A better solution is a **table with columns**: "external_system_alias", "external_entity_id" and "external_entity_status". Each row of the table corresponds to one external system.

SX content developer note: If you do not use an external reference "table" for your system/adapter, you can still integrate by case exchange,

but you cannot exchange to a number of other systems at the same time. See Single or Multiple linkedEntity.

Examples:

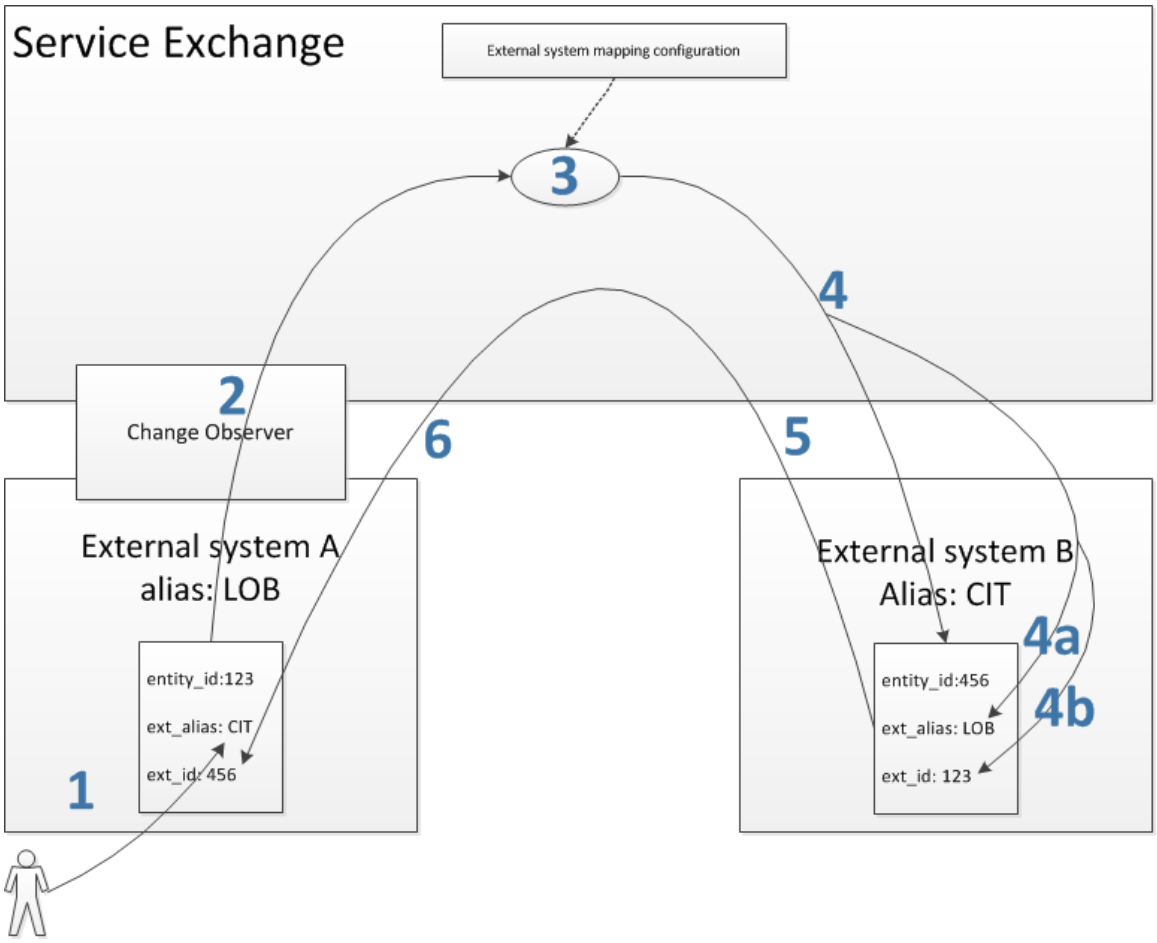
- SM - External references are stored in the SX custom table "SXRegisteredEntitiesV2".
- JIRA - External reference is stored as JSON in field "Environment".
- SAW - External references are stored in an internal table accessed by REST.

External system pairing is configurable in `external_system.json`. There are sections for every external system pair (most often bi-directionally).

external_system.json

```
"externalSystemAliases": [
    {
      "sourceInstanceType": "SM",
      "sourceInstance": "mpavmsm10",
      "targetInstanceType": "SAW",
      "targetInstance": "msalb003sngx",
      "targetAlias": "lobSAW"
    },
    {
      "sourceInstanceType": "SAW",
      "sourceInstance": "msalb003sngx",
      "targetInstanceType": "SM",
      "targetInstance": "mpavmsm10",
      "targetAlias": "supportSM"
    }
  ]
```

Basic linking scenario:



1. User of system A fills the ext_alias field of an entity.
2. HP SX detects the change of the entity.
3. HP SX translates system B alias to the ID of system B by external_system.json mapping.
4. HP SX creates a new entity in System B.
    a. HP SX fills ext_alias of system A.
    b. HP SX fills ext_id of incident in system A
5. System B returns the ID of the new incident.
6. HP SX fills ext_id of the incident in system B.

Change Observation

See Change Observers.

Symmetry

There are three ways to define a set of events for a given entity type:

- Symmetrical (Default)
- Non-symmetrical
- Mixed.

Symmetrical:

- This is the default method.
- Every event can be thrown from both paired systems, for example, an entity is cloned from system A to B and also from B to A.
- There is one **eventGroup** and both paired systems share it.

Example: Incident case exchange for SAW<->SM.

Non-symmetrical:

- There are two sets (groups) of events: incoming and outgoing.
- Incoming and outgoing events can have different implementations.
- First system has incoming events only, the second has outgoing events only.

Example: Problem case exchange for SM<->SM.

Mixed:

- There is one set of events.
- There are two event groups, each for one of any paired systems, for example, an entity is cloned from system A to B only, the entity is resolved in system B and then CX resolves it in system A only.

Example: Incident case exchange for JIRA<->SM.

Configuration

Table of Contents:

Overview

This configuration section describes how the CX framework is configured to communicate with backend systems and to perform entity data exchange from one system to another. It includes:

- Which configuration files are involved in setting up CX operations.
- How various concepts of CX (like Events and Event Groups) are set up, including real-life examples of JSON configuration to illustrate what is being described.

Configuration files

There are two configuration files involved in CX configuration: `external-systems.json` and `case-exchange.json`..The data format of both files is JSON.

external-systems.json

There is one external-systems.json file in the HP SX war. In its first section, externalSystems, it contains the definitions for individual **external systems** one by one in an array. In the second section, **externalSystemAliases**, it contains definitions of external system pairs.
Here is an example from the `external-systems.json` file:

external-systems.json

```
{
     "externalSystems": [ // definitions of external systems, each item of the array
defines one external system instance
     { // the first external system instance
          "instanceType": "SAW", // the type of the defined system
          "instance": "msalb003sngx", // the name of the external system
          "registeredEventGroups": [ // the array of event groups that should be
observed on the external system
               "IncidentCaseExchangeEvents"
          ]
      },
      { // the second external system instance
          "instanceType": "SM",
          "instance": "mpavmsm08",
          "registeredEventGroups": [
              "IncidentCaseExchangeEvents"
          ]
     }
     ],
     "externalSystemAliases": [ // definitions of external system pairs, each item of
the array defines one external system pair
          { // first external system pair
            "sourceInstanceType": "SM", // the type of source external system of the
pair
            "sourceInstance": "mpavmsm08", // the name of the source external system
instance
            "targetInstanceType": "SAW", // the target external system type
            "targetInstance": "msalb003sngx", // the target external system name
            "targetAlias": "saw" // the alias used in source external system to
identify the target external system, eg. the target external system alias in the
source external system
        },
        { // second external system pair
            "sourceInstanceType": "SAW",
            "sourceInstance": "msalb003sngx",
            "targetInstanceType": "SM",
            "targetInstance": "mpavmsm08",
            "targetAlias": "SM08"
        }
     ]
}
```

case-exchange.json

While there is only one `external-systems.json` file, there are typically multiple `case-exchange.json` files. Their content is combined as if there was a single file.

NOTE: If some case-exchange.json files contain incompatible content, the resulting configuration is non-deterministic and may cause problems.
Each `case-exchange.json` file may contain the following sections:

1. **Events**. Events are defined on the level of the external system type, for example, HP SM. The events recognized by the CX framework are defined in this section:

events

```
     "events": { // the "events" section
        "SM": { // the identifier of the external system whose events we are
defining
            "incidentExternalReferenceCreated": { // the name of the defined
event
                "entityType": "probsummary", // the native (eg. external system
specific) entity type
                "entityFilter": "RECORD['vendor']!=null &&
RECORD['reference.no']==null && (ISCREATE || ISUPDATE &&
OLDRECORD['vendor']!=NEWRECORD['vendor'])", // the filter defining event trigger
condition
                "changeType": [ "create", "update" ] // optional Service manager
specific field
            },
            "incidentUpdated": {
                "entityType": "probsummary",
                "entityFilter": "RECORD['vendor']!=null &&
OLDRECORD['vendor']==NEWRECORD['vendor'] &&
(OLDRECORD['brief.description']!=RECORD['brief.description'] ||
OLDRECORD['action'].toString()!=RECORD['action'].toString() ||
OLDRECORD['severity']!=RECORD['severity'] ||
OLDRECORD['initial.impact']!=RECORD['initial.impact'])",
                "changeType": [ "update" ]
            }
          ...
     }
```

2. **eventGroups**. In this section, event groups are defined by specifying a list of contained events for each of them:

eventGroups

```
"eventGroups": { // the "eventGroups" section
    "IncidentCaseExchangeEvents": [ // the name of the event group being defined
        "incidentExternalReferenceCreated", // the name of the first event
belonging to the group
        "incidentUpdated", // the name of the second event belonging to the group
        "incidentResolved", // ...
        "incidentReopened",
        "incidentClosed",
        "incidentOwnershipAssigned",
        "incidentOwnershipAccepted",
        "incidentRejected",
        "incidentCancelled"
     ],
     "TaskCaseExchangeEvents": [ // the name of another event group
          "taskExternalReferenceCreated" // this group only contains one event
      ]
}
```

3. **eventActions**. The action or sequence of actions to be performed once an event is triggered. The order of execution when merging event actions from multiple configuration files is not defined. Each Action represents one of two currently supported action types:
    - **executeOperation** – An HP SX operation is executed. Based on the value of the backendSystemType property, the operation definition is searched for in content packs associated with the respective backend system type.
    - **executeOoFlow** - An OO Flow is executed. Based on the value of the backendSystemType property, the flow is executed on behalf of the corresponding backend system. The flow to be executed is determined by the value of the messageType property. The message type is used to search for flow information in the `flows.json` file.

eventActions

```
"eventActions": { // the "eventActions" section
     "incidentClosed": { // the event we're defining actions for
          { // the first action to be executed when the event is triggered
               "action": "executeOperation", // action = execute operation
               "backendSystemType": "SM" // the backend system to be
searched for the operation (each content pack's metadata.json file contains
the "adapter" property assigning the content pack to the respective backend
system)
               "operationName": "retrieveIncident" // the name of the
operation to be executed
          },
          { // the second action to be executed when the event is triggered
               "action": "executeOperation",
               "backendSystemType": "SX"
               "operationName": "convertAssignmentGroupToInstance"
          },
          { // the third action to be executed when the event is triggered
               "action": "executeOoFlow", // action = execute OO flow
               "backendSystemType": "SX", // the backend system on whose
behalf the OO flow will be executed
               "messageType": "IncidentCaseExchangeFlow" // the type of the
message to be sent to the OO flow; also determines which OO flow should be
used - corresponds to the key in the flows.json configuration file
          }
     }
}
```

4. **eventGroupActions** - The action or sequence of actions to be performed once an event from the given event group is triggered. The order of execution between event actions and event group actions is not deterministic, so it is not recommended to mix event actions and event group actions together when the order of execution is important. Both the syntax and semantics of the eventGroupActions is the same as for the eventActions

*-mappings.json

Each external system type participating in CX has its own set of entities, its own vocabulary, and its own property names and values. To allow CX to communicate between different types of systems, the vocabulary, entities and properties, and their values, have to be unified. The CX implementation uses a common data format for the exchanged data called the Canonical Model. As a helper for data transformation between the canonical model and the external system native data model, each external system can provide a mapping file to aid the translations.

The name of the mapping file is in the form of **<external_system_type>-mappings.json**, for example sm-mappings.json. It may contain translation tables for entity names and property values. The translation tables can be used by content packs to make easy transformations, most importantly in Freemarker templates. Property names are not typically translated via translation table as it is much easier to perform their translation directly in Freemarker templates. In the next paragraphs, we will show an example of each of the mappings.

Entity name mappings

In this section of the mapping file, the native entity names are mapped to canonical model entity names:

Entity Name Mappings

```
"entityType": { // the section start
        "Incident": "probsummary", // pair of Canonical Model/native external system
entity name
        "IncidentTask": "imTask" // another pair for another entity
    }
```

Property value mappings

For each entity, a mapping for some of its property values between the Canonical Model and the native external system values may be provided:

Property Value Mappings

```
"Incident": { // the name of the entity
        "Status": { // the name of the property in Canonical Model whose values will
be translated via this table
            "Open": "Ready", // pair of Canonical Model/native external system
property value
            "WorkInProgress": "InProgress", // another pair
            "PendingChange": "Pending",
            "PendingOther": "Suspended",
            "Complete": "Complete"
        },
        "Urgency": { // another property whose values will be translated
            "U4": "NoDisruption",
            "U3": "SlightDisruption",
            "U2": "SevereDisruption",
            "U1": "TotalLossOfService"
        }
}
```

Freemarker code

Once the mapping is defined in the mapping file, the mapping can be used to translate the value within a Freemarker template:

Freemarker Code

```
<#assign
findKey='com.hp.ccue.serviceExchange.adapter.freemarker.FindKeyForValue'?new()/> //
declare the findKey function defined in Java code of Service Exchange API for Adapters
<#assign sawMapping=loadConfig(context.contentStorage,
"saw-case-exchange/saw-mappings") /> // declare the sawMapping variable containing the
mapping for Service Anywhere (SAW) system

{
 "properties": {
  "Urgency": "${findKey(sawMapping.Incident.Urgency, entityProperties.Urgency)}", //
use the Service Exchange provided findKey() function to perform the translation of
Urgency to Canonical Model specific value
  "Status": "${findKey(sawMapping.Incident.Status, entityProperties.Status)}" // use
the Service Exchange provided findKey() function to perform the translation of Status
to Canonical Model specific value
 }
}
```

Configuration concepts

When configuring a CX framework for HP SX content, the following items need to be configured:

- External Systems
- External System Pairs
- Entity Types to be Case Exchanged
- Events
- Event Filters
- Event Groups
- Event and Event Group Actions

External systems

In order to have an external system participate in CX, it must be present in the external system configuration. The configuration entry must contain:

- the system type (for example HP SM, JIRA), the name of the system instance (corresponding to the name assigned to it in the instances.json configuration file for the respective external system type.)
- the array of event groups CX will handle for this particular external system.

Here is an example of an external system configuration:

External System

```
{
     "instanceType": "SM", // the type of the external system
     "instance": "mpavmsmapp01", // the name of the concrete external system instance
     "registeredEventGroups": [ // the event groups activated for this system instance
          "TaskCaseExchangeEvents",
          "TaskCaseExchangeIncidentEvents"
     ]
}
```

External system pairs

To configure CX to perform entity data exchange between two particular systems, it is necessary to create an external system pair for them. In the pair definition:

- source system must be specified by its type and name
- target system must be specified by its type and name
- an alias to be used by users in the source system to identify the target system.

Here is an example of an external system pair configuration:

External system Pair

```
{
     "sourceInstanceType": "SM", // the source external system type
     "sourceInstance": "mpavmsm08", // the source external system name
     "targetInstanceType": "JIRA", // the target (receiving) external system type
     "targetInstance": "mpavmint01", // the target (receiving) external system name
     "targetAlias": "jira" // the alias used for the target system instance in the
source system
}
```

Entity types to be case exchanged

The entity types to be case exchanged are not specified directly. Instead, for each external system, an array of event groups is specified to be watched for in the system. See the External Systems section for an example of such a configuration. Each event group consists of several individual events, typically all associated with a specific entity type. See the Event Groups section for an example of an Event Group configuration and the Events section for an event configuration example. In this way, this indirect specification determines which entities are processed for the particular external system.

Events

The operation of the CX framework is based on events. Depending on the external system type and the changed entity type, the set of potential events that can occur is defined. The source external system is being watched for changes. Once an entity change occurs, CX is notified by the external system Change Observer. For each applicable event, its filter is checked and if its filter condition is satisfied by the entity change, the corresponding event is triggered. See the Event Filters section for more detail. As a result, each entity change can trigger one or more events. Here is an example event definition:

Event

```
"incidentUpdated": { // the name of the event being defined
     "entityType": "probsummary", // the native type of the entity the event is
defined for; probsummary is Service Manager's type for Incident
     "entityFilter": "RECORD['vendor']!=null &&
OLDRECORD['vendor']==NEWRECORD['vendor'] &&
(OLDRECORD['brief.description']!=RECORD['brief.description'] ||
OLDRECORD['action'].toString()!=RECORD['action'].toString() ||
OLDRECORD['severity']!=RECORD['severity'] ||
OLDRECORD['initial.impact']!=RECORD['initial.impact'])", // the filter containing
boolean expression for the event triggering; Service Manager Change Observer provides
information about entity value before (OLDRECORD) and after (RECORD) the change
     "changeType": [ "update" ] // this field is optional and is used by Service
Manager Change Observer to determine whether the event should be triggered for new,
existing or both type of records
}
```

Event filters

The definition of each event contains one or more filters. The filters are conditional expressions operating over changed entity data, written in Javascript syntax. Once an entity change is being processed by the CX framework, the filters for each potential event are evaluated. If at least one of them is evaluated to true, the respective event is triggered, ready for further processing. The input parameters for the condition vary between external system types because they are heavily depending on the entity change data, which in turn is generated by the system's Change Observer, and their format and content are not standardized.
Here is an example of an event filter definition for HP SM:

Filter Expression

```
"RECORD['assignment']!=null && (ISCREATE || ISUPDATE &&
OLDRECORD['assignment']!=NEWRECORD['assignment'])"
```

Event groups

Events may be grouped together to form an Event Group. All the events in a group need to be applicable to the same entity. Event groups have two purposes:

- To allow assigning a common action to a set of events.
- To configure which events should be observed on a particular system.

Only event groups may be assigned to a target external system. Therefore, the only way to observe an event on a particular external system is to create an event group containing that event and add the event group to the registeredEventGroups property array in the external system configuration. An event may be part of different Event Groups.
Here is an example of an Event Group definition:

Event Group

```
"IncidentCaseExchangeEvents": [ // the name of the Event Group
     "incidentExternalReferenceCreated", // an array of individual Events to be part
of the Event Group, identified by their name
     "incidentUpdated",
     "incidentResolved",
     "incidentReopened",
     "incidentClosed",
     "incidentOwnershipAssigned",
     "incidentOwnershipAccepted",
     "incidentRejected",
     "incidentCancelled"
]
```

Here is an example of how to assign the Event Group to an external system instance:

Event Group assignment

```
{
     "instanceType": "SM", // the External System type
     "instance": "mpavmsm09", // the External System name
     "registeredEventGroups": [ "problem.ReferringEntityEvents" ] // an array of Event
Groups to be observed for this External System instance
}
```

Event and Event Group actions

The last piece of the configuration is to define what the CX framework should perform after an Event is triggered. The execution units in HP SX are called **operations**. For each event, the user can define a set of operations to be executed once the Event is triggered. Another set of operations can be configured for a whole **event group**. If operations are defined for the Event Group and for an Event from such a group, the group operations execute first, and then the event operations execute.
Here is an example of an Event operation definition:

Event Group Actions

```
"IncidentCaseExchangeEvents": [
{
     "action": "executeOperation",
     "operationName": "retrieveIncident"
},
{
     "action": "executeOperation",
     "operationName": "convertIncidentToCanonicalModel"
},
{
     "action": "executeOoFlow",
     "backendSystemType": "SX",
     "messageType": "IncidentCaseExchangeFlow"
}]
```

The same block of configuration can be used to configure operations for an Event Group.

Operations

Introduction

Whenever the CX framework needs to interact with an external system, it uses an HP SX operation. An operation is a set of steps, where each of the steps represents a network interaction with a target system (typically a REST call.) The output of an operation is a JSON message available to other HP SX components.

See Appendix B: Operation executors for information about operation definitions, their format and properties.

The CX actions for Incident events are configured in the following way:

eventGroupActions

```
"eventGroupActions": {

    "IncidentCaseExchangeEvents": [
      {
        "action": "executeOperation",
        "operationName": "retrieveIncident"
      },
      {
        "action": "executeOperation",
        "operationName": "convertIncidentToCanonicalModel"
      },
      {
        "action": "executeOoFlow",
        "backendSystemType": "SX",
        "messageType": "IncidentCaseExchangeFlow"
      }
    ]
}
```

The meaning of this configuration is:

Once a change is detected and an Incident event is triggered, then:

1. Operation 'retrieveIncident' is executed.

2. Operation 'convertIncidentToCanonicalModel' is executed.
3. 'IncidentCaseExchangeFlow' OO flow is executed.

Each adapter willing to participate in Incident CX must provide its own implementation of the two operations, **retrieveIncident** and **convertIncidentToCanonicalModel**. The Incident OO Flow is standardized across various systems and there is no need to customize it.

In addition, there is one more important operation involved in CX: the default implementation of ChangeObserver uses the operation **getChangedIncidents** to periodically poll an external system for any changes on observed Incidents.

Any adapter using a Change Observer extending the **PollingChangeObserverBase** to check for changes in an external system, must also provide the implementation of the **getChangedIncidents** operation.
In the following section, each of the three operations in described in detail.

'getChangedIncidents' operation
This operation only needs to be implemented for adapters having a change observer extending the **PollingChangeObserverBase** class. The result of this operation should contain the array of changes in an external system instance, one array item per changed entity. The concrete format is customizable, as any logic working with changed entity data is left to be implemented by the adapter-specific change observer. See the Javadoc **PollingChangeObserverBase** java class for more detail. This operation is executed from within the PollingChangeObserverBase class common code to check for changes in the external system entities that have happened in the time interval since the last check.
By default, the input message contains a '**messageHeader.targetInstance**' field and a '**lastUpdateTime**' field. The input of the operation can be customized by overriding the **customizeGetChangedEntitiesMessage**() method in the **PollingChangeObserverBase** class.
As already stated, the output of this operation is not strictly defined and must be aligned with the particular **ChangeObserver** implementation for the adapter.

'retrieveIncident' operation
This operation is called by the CX framework whenever an event has been triggered for a particular Incident to retrieve the incident properties. The input for the operation looks like this:

'retrieveIncident' operation input

```
{
     "entityChange": { // the structured information about the changed event
          "instanceType": "SM", // the external system type
          "instance": "mpavmsm04", // the name of the external system
          "entityType": "Incident", // the type of the changed entity
          "entityId": "IM03245", // the ID of the changed entity
          "changeType": "update", // the type of the change (create vs. update)
          "changeReason": "caseExchange", // the reason of the change (the value will
typically be caseExchange)
          "changeArgs": "aaa", // optional property - the arguments of the change
          "data": {} // the data associated with the change
}
```

The operation is responsible for issuing a request to the external system and retrieving all the changed entity properties. In this way it is possible to create an entity representation in the canonical model, which is the next operation in line.

'convertIncidentToCanonicalModel' operation
The CX framework is designed to work with many external system types. Obviously, each external system uses its own data format to represent its entities. The CX framework is introducing a common data format for representing entities from different external systems in a uniform way, see Canonical Model. The '**retrieveIncident**' operation returns the entity data in its native format. The '**convertIncidentToCanonicalModel**' operation is responsible for converting the Incident data from the native format to the canonical format, so that the data can be freely exchanged with other external system types. The '**convertIncidentToCanonicalModel**' operation is executed before the Incident OO flow is invoked, as the OO Flow expects the data to be in canonical format.
The input of the '**convertIncidentToCanonicalModel**' operation is determined by the output of the '**retrieveIncident**' operation and may vary from external system to external system.
The output is a canonical model representing the entity and its properties, together with any linked entity if applicable.

Event/Entity specific operations
Besides the common operations described above, CX defines one or more operations specific to any particular event. For example, if an **incident Updated** event is triggered, the **updateLinkedIncident** operation on the target external system is invoked. Which operation or operations are invoked is determined by the specific OO Flow.

For each event, the OO flow decides which operation on which external system is invoked, and in which order. For the external system to be able to participate in CX for the respective entity, it must implement all the operations defined by the respective OO Flow associated with the respective entity type. Each entity type must have exactly one OO Flow associated with it. A set of operations needed by a specific OO flow is part of its documentation and is not described here.
The arguments for each such operation are determined by the OO flow as well.

OO flows

Overview

OO flow is part of the CX configuration. The main function of the flow is deciding which operations will be called in external systems.

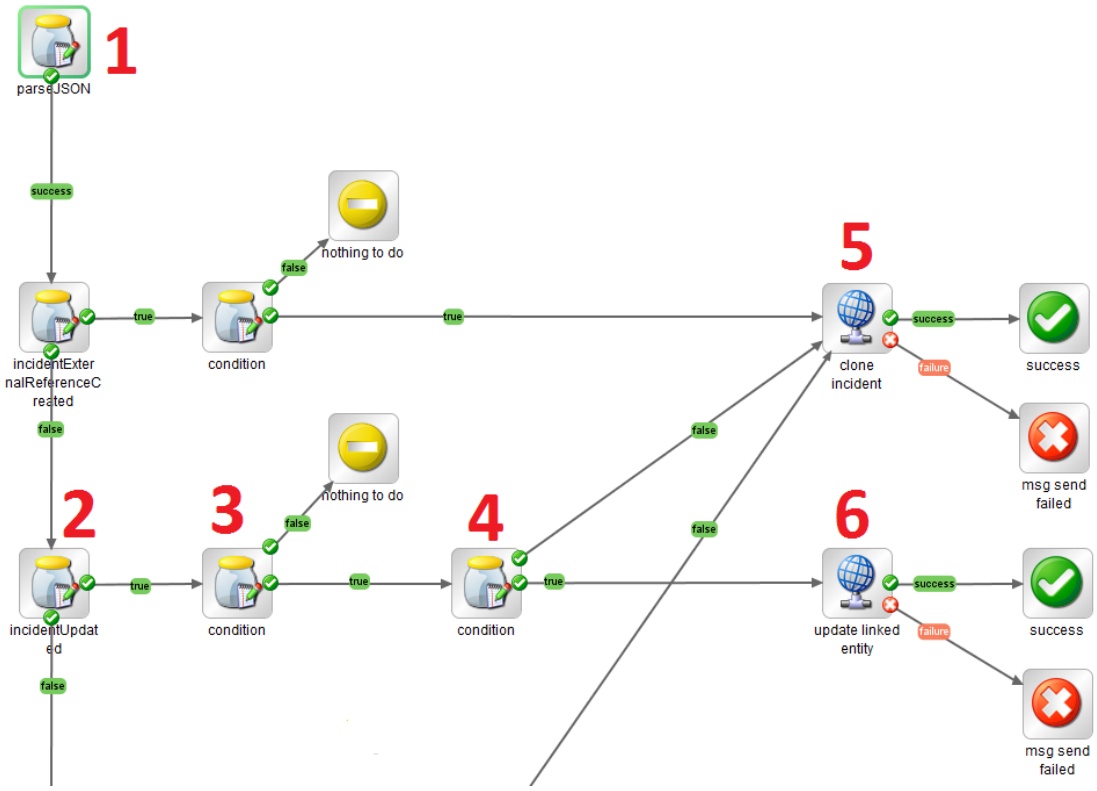
Main facts:

For every entity type (Incident, Task, Problem etc) there is one CX OO flow.

- OO flow runs after an event is triggered in a source external system.
- OO flow works with messages in the canonical model only, see Canonical Model.
- OO flow decides which HP SX operations to call in source and target external systems.
- OO flow decides which adapters process operations.
- OO flow can decide to do nothing: for example, where an entity was rejected but it was not yet cloned to an external system.
- If an entity in a source system is mapped to several external systems, OO flow will run once for every external system, see Single or Multiple linkedEntity.
- OO flow calls HP SX by sending JMS messages.

Case exchange OO flow description

The following section explains how the OO flow is executed and which conditions/operations are called. Here is a screenshot of a real Incident CX flow:



0. Message in canonical model is sent to OO flow.

Input message of the flow

```
{
    "event":"incidentUpdated",
    "entity":{
        "instanceType":"SM",
        "instance":"mpavmsm10",
        "entityType":"Incident",
        "entityId":"IM13087",
        "properties":{
            "Title":"Window was broken",
            "Description":"An angry employee broke the window",
            "Status":"PendingVendor",
            "Urgency":"I4",
            "Impact":"I2",
            "Attachments":[ ]
        }
    },
    "linkedEntity":{
        "instanceAlias":"saw",
        "instanceType":"SAW",
        "instance":"msalb003sngx",
        "entityType":"Incident",
        "entityId":"19822",
        "properties":{
            "Status":"Complete",
            "Attachments":[ ]
        }
    }
}
```

1. The first OO step parses the message and splits it into many OO context variables (for example, entity.instance, entity.properties.Title). The other OO steps use these context variables directly, and do not parse the json message again.

2. This step checks if the event is **incidentUpdate**

condition:

```
$('event') == 'incidentUpdated'
```

3. This step checks if the incident is linked to an external system

condition:

```
nonEmptyString($('linkedEntity.instanceType')) &&
nonEmptyString($('linkedEntity.instance'))
```

4. This step checks if the incident is linked to any other incident in the external system. If it is not, the incident will be cloned, or else the incident will be updated. A similar scenario is used for most events, because a user can work with an incident in the source system and link the incident to an external system in any phase of the lifecycle (for example, during resolving). This means the clone operation has to switch the cloned incident to the right state.

condition:

```
nonEmptyString($('linkedEntity.entityId'))
```

5. This step calls the cloning operation in the external system and the updating operation in the source system. A batch operation is used. The typical implementation in the adapter is: SX creates a new incident in the target system. The operation returns ID + state of the new incident and adds it to the message. The second operation copies this information back to the source system.

In the following example you can see unfilled parts of the message, which will be filled by the first operation. You can also see how the adapter is chosen by target system type.

messageText

```
{ // placeholders ${xxx} will be replaced by OO context variables from input message.
First step of the flow prepared it.
    "operations":[
        { //first operation for target system
            "operationName":"cloneIncident",
            "messageType": "${linkedEntity.instanceType}:PLAIN", //messageType
(adapter) is choosen by information in input message
            "message":{
                "messageHeader":{
                    "backendSystemType":"${linkedEntity.instanceType}",
                    "targetInstance":"${linkedEntity.instance}"
                },
                "args":{
                    "event":"${event}",
                    "entity":${entity},
                    "linkedEntity":{
                        "instanceType":"${linkedEntity.instanceType}",
                        "instance":"${linkedEntity.instance}"
                    }
                }
            }
        },
        { //second operation for source system
            "operationName":"updateLinkedIncidentInfo",
            "messageType": "${entity.instanceType}:PLAIN",
            "message":{
                "messageHeader":{
                    "backendSystemType":"${entity.instanceType}",
                    "targetInstance":"${entity.instance}"
                },
                "args":{
                    "event":"${event}",
                    "entity":{
                        "instanceType":"${entity.instanceType}",
                        "instance":"${entity.instance}",
                        "entityType":"${entity.entityType}",
                        "entityId":"${entity.entityId}"
                    },
                    "linkedEntity":{
                        "instanceAlias":"${linkedEntity.instanceAlias}",
                        "instanceType":"${linkedEntity.instanceType}",
                        "instance":"${linkedEntity.instance}",
                        "TODO:entityType":"TODO: overwrite in cloneIncident", //these
parts will be filled by output of the first operation
                        "TODO:entityId":"TODO: overwrite in cloneIncident",
                        "properties":{
                            "TODO:Status": "TODO: overwrite in cloneIncident"
                        }
                    }
                }
            }
        }
    ]
}
```

6. This step calls the updating operation in the external system and the source system. The content is similar to step 5, but there is only a single operation for the target system.

messageText

```
{
    "messageHeader":{
        "backendSystemType":"${linkedEntity.instanceType}",
        "targetInstance":"${linkedEntity.instance}"
    },
    "args":{
        "event":"${event}",
        "entity":${entity},
        "linkedEntity":${linkedEntity}
    }
}
```

This is a description of one incident flow branch. Other branches are very similar.

About Incident flow

Incident CX flow is shared for all incidents exchanging between every type of system (i.e. it is shared across all backend system adapters). Here is a conversion table between events and SX operations:

	incident was cloned already		incident was not cloned yet	
event name	**operation in target system**	**operation in source system**	**operation in target system**	**operation in source system**
incidentExternalReferenceCreated	--	--	cloneIncident	updateLinkedIncidentInfo
incidentUpdated	updateLinkedIncident	--	cloneIncident	updateLinkedIncidentInfo
incidentClosed	closeIncident	updateLinkedIncidentInfo	cloneIncident	updateLinkedIncidentInfo
incidentResolved	resolveIncident	updateLinkedIncidentInfo	cloneIncident	updateLinkedIncidentInfo
incidentReopened	reopenIncident	updateLinkedIncidentInfo	cloneIncident	updateLinkedIncidentInfo
incidentOwnershipAssigned	assignOwnershipToIncident	updateLinkedIncidentInfo	cloneIncident	updateLinkedIncidentInfo
incidentOwnershipAccepted	acceptOwnershipOfIncident	updateLinkedIncidentInfo	cloneIncident	updateLinkedIncidentInfo
incidentRejected	rejectIncident	updateLinkedIncidentInfo	--	--
incidentCancelled	cancelIncident	updateLinkedIncidentInfo	--	--

All events are symmetrical, see: Symmetry.

message format for updateLinkedIncidentInfo operation:

message

```
"message":{
    "messageHeader":{
        "backendSystemType":"${entity.instanceType}",
        "targetInstance":"${entity.instance}"
    },
    "args":{
        "event":"updateLinkedIncidentInfo",
        "entity":{
            "instanceType":"${entity.instanceType}",
            "instance":"${entity.instance}",
            "entityType":"${entity.entityType}",
            "entityId":"${entity.entityId}"
        },
        "linkedEntity":{
            "instanceAlias":"${linkedEntity.instanceAlias}",
            "instanceType":"${linkedEntity.instanceType}",
            "instance":"${linkedEntity.instance}",
            "entityType":"${previousOperationResult.entityType}",//filled by previous
operation
            "entityId":"${previousOperationResult.entityId}",//filled by previous
operation
            "properties":{
                "Status": "${previousOperationResult.Status}"//filled by previous
operation
            }
        }
    }
}
```

message format for all other operations (cloneIncident, updateLinkedIncident, closeIncident, resolveIncident, reopenIncident, assignOwnershipToIncident, acceptOwnershipOfIncident, rejectIncident, cancelIncident)**:**

message

```
"message":{
    "messageHeader":{
        "backendSystemType":"${linkedEntity.instanceType}",
        "targetInstance":"${linkedEntity.instance}"
    },
    "args":{
        "event":"${event}",
        "entity":${entity},
        "linkedEntity":${linkedEntity}
    }
}
```

Change Observers

Table of Contents:

This section offers:

- An overview of change detection.
- Describes the support for change detection in the existing code.
- Outlines the way to use change detection with new HP SX adapters.
- Gives a specific example of how change detection is implemented in the HP SAW adapter using the Polling Command.
- Describes what should be the outcome of a brand new implementation of change detection not based on existing code.

Overview

Each external system type participating in CX must be able to perform relevant change detection in the external system. The mechanism can be either based on periodical polling of the external system, or on having the changes pushed back to the adapter. In either case, the external system adapter willing to participate in CX must provide its implementation of the ChangeObserver interface provided by the CX API. The implementer of the system-specific Change Observer can chose to reuse an existing polling based system and only override methods for system specific execution, or implement the change observation completely from scratch.

In either case, the Change Observer is responsible for:

- Collecting information about changed entities.
- Checking event filters for each change.
- Sending a CX message to the respective pipeline of the external system adapter for each triggered event.

ChangeObserver implementation

What follows are details for implementing the ChangeObserver:

a. Based on an existing Polling Observer
b. From scratch.

Polling change observer implementation

The existing implementation of ChangeObserver can be found in the CompositeChangeObserver Java class. The class is abstract and anyone wanting to use it as a base for his own implementation must create a subclass. The implementation of the class is fairly simple. In its constructor, the class is expecting a List of Commands (Runnables). The commands are then sequentially executed when checking the external system for changes. The subclass is expected to pass its own list of commands to the CompositeChangeObserver constructor.

There are no restrictions on the nature of the Runnables passed as commands. However, to facilitate the existing code in the CX framework for change polling, the implementer is expected to extend the PollingBaseCommand or one of its subclasses - **CxPollingByAliasCommand** or **CxPollingCommand**. These two classes differ in their mode of operation:

1. **CxPollingByAliasCommand** issues one request per alias, whose source external system type matches the ChangeObserver's external system type.
   An example: The SawChangeObserver is polling HP SAW systems for changes. There are three aliases defined in `external-systems.json`. The first alias connects an HP SAW system to an HP SM system, the second alias connects an HP SM system to an HP SAW system, and the third alias connects an HP SAW system to another HP SAW system. Only the first and third alias is processed by the SawChangeObserver, because the source external system type of these aliases is HP SAW. When polling those systems, one polling request is issued to the source external system of the first alias and the other request is issued to the source external system of the third alias.
2. **CxPollingCommand** processes all the systems present in the `instances.json` configuration file for the corresponding external system type (for example JIRA.) For each instance, it checks whether the system is defined in the external-systems.json file as well. If it is, the CxPollingCommand polls the respective external system for changes.

Once the changes are detected, the mechanism for processing them is the same for both implementations:

- Each event defined for the entity type that has been changed in the external system has an event filter defined in the configuration. The event filter is an expression containing variables whose value is determined by the entity change. If the filter evaluates to true for the particular change, the respective event is triggered.
- The triggered event is placed into a message. The message is sent to HP SX for further processing. The pipeline for processing the message is set to **CaseExchangePipeline**.
- The **CaseExchangePipeline** makes sure the actions defined for the event or its event group are executed, and typically at the end of the processing a message to Operations Orchestration is sent.

Overriding CxPollingCommand

The most important method to override in the Polling Change Observer based on CxPollingCommand is the constructor. Several parameters have to be passed to the CxPollingcommand constructor:

- The String representing the type of the external system to be polled, for example SAW.
- The String representing the (adapter-specific) entity type to be polled, for example probsummary.
- The name of the instances configuration file for the external system type.
- The name of the HP SX operation to be called to poll the remote system for changes.
- The Operation Executor to be used to execute the operation.
- The rule store for the particular external system type. It stores the events and event filters for the respective system and returns them as a List of Listeners
- The Filter Evaluator to be used for event filter evaluations.

Another method to be overridden by the subclass is **extractChangedEntities**(). It is called to extract the list of changed entities from a Map representing the polling operation result. Note that the output of the operation is not standardized and may hugely differ between external system types.

For proper entity ID extraction from a changed entity representation, the **extractEntityId**() method is called. It gets a Map of the entity JSON object as its argument.

The last abstract method to override is **prepareMessageForCustomDataCx**(). This method is called to prepare a message to be sent to the CaseExchangePipeline, representing the event generated by the change in the external system entity.

Overriding SxPollingByAliasCommand

While the mode of operation of SxPollingByAliasCommand is different to CxPollingCommand, the set of methods to be overridden is the same, including the constructor and its parameters.

Provided content packs

OOB content packs

HP SX contains the following out-of-the-box content packs:

- **sx-base** - the base content for HP SX. This content pack is required and cannot be removed.
- **csa-r2f** - the content pack providing files for HP CSA requests-to-fulfillment.
- **sm-r2f** - the content pack providing files for HP SM requests-to-fulfillment.
- **sm-ticketing** - the content pack providing files for HP SM ticketing.
- **sm-case-exchange** - the content pack providing files for HP SX CX customizations.
- **sm-test-ui-support** - the content pack providing files for HP SM related functions of HP SX UI.
- **csa-test-ui-support** - the content pack providing files for HP CSA related functions of HP SX UI.
- **mock-r2f** - an empty content pack.
- **email-r2f** – the content pack providing files for email requests-to-fulfillment.
- **saw-case-exchange** – the content pack providing files for HP SAW CX customizations.

Apart from the system or testing content pack, OOB content packs implement the business functionality outlined in the table below.

Content pack	Description

csa-r2f	Enables request-to-fulfill (R2F) use case for CSA offerings aggregated into the Propel portal catalog. Contains support for portal actions: • approve/deny request • check subscription • cancel subscription
sm-r2f	Enables R2F use case for HP SM catalog items aggregated into Propel portal catalog. It implements notification emails. Notification emails are sent to • approver after the user submits the request • requester notifying the approval/denial • requester notifying fulfillment
email-r2f	Provides support for items that are considered to be fulfilled by confirmation in UI. Scenario: 1. User requests an item. 2. An email is sent to the appropriate person. The email sent contains links to confirm or deny the request. 3. The request is finished/denied by clicking the confirm/deny link.
sm-ticketing	Provides support for the ticketing use case with HP SM as the ticketing backend system. Supports standard ticketing management operations: create, add comment, add attachment, close etc. Create ticket operation results in the creation of an **Interaction** entity in HP SM.
case-exchange	This content pack contains backend system-type independent CX configuration and operations. It supports mainly: • Incident case exchange - where incidents are exchanged between systems • Incident task case exchange - where tasks created under an incident are exchanged. The main advantage with this is that multiple tasks under a single incident can be assigned to different systems simultaneously.
sm-case-exchange	This content pack contains backend system-type specifics: CX configuration and operations specific to HP SM, that support generic configuration in content-case-exchange. This includes mainly case exchange event definitions.
saw-case-exchange	This content pack contains backend system-type specifics: CX configuration and operations specific to **HP SAW**, that support generic configuration in content-case-exchange. This includes mainly CX event definitions.

SDK provided content packs

These are content packs provided within the SDK package in `sx-content` directory:

Content pack	Description

jira	Ticketing --- Provides support for the ticketing use case with JIRA as the ticketing backend system. Supports standard ticketing management operations: create, add comment, add attachment, close etc. Create ticket operation results in the creation of an ISSUE entity in JIRA. The exposed JIRA ISSUE properties that are submitted as ticket properties are: • summary • description • project • issuetype • priority • reporter • asignee R2f --- The user creates an order in Propel. When creating an order, the user chooses a project to create the task in and specifies the properties of the task (title, description, reporter, priority.) As a result, a task in JIRA is created with **Open** status and a mail is sent to the lead of the JIRA project (who is acting as the Approver). Additionally, a notification is sent to the catalog. The lead invokes the approve operation in Propel. As a result, the task status is set to **In Progress** and a notification email is sent to the reporter. Additionally, a notification is sent to the catalog. A developer resolves the task in JIRA. As a result, the reporter receives an email. Additionally, a notification is sent to the catalog. A developer closes the task in JIRA. As a result, SX stops listening to changes of this task. Case-exchange --- Contains support for CX events and operations in the JIRA backend system type. The use case supported: 1. An Incident clone is triggered in a linked external system, an incident is cloned in JIRA as a linked issue. 2. When the JIRA linked issue is resolved, the original incident will be automatically resolved as well.

How to extend HP SX Content (HP SM Problem entity)

This section demonstrates how to extend HP SX content capabilities.

The example shows the the implementation of an HP SX content pack for OOB provided HP SM adapter. To demonstrate HP SX capabalities the HP SM entity called **Problem** was chosen. The example extends HP SX functionality to:

- support the `create` operation for Problem entity
- monitor its state
- send notifications about it back to the Propel catalog
- when the Problem is solved, inform the submitter via email.

To explain the create operation in more detail. After the implementation of this example HP SX will be able to handle an icoming POST request on its /request REST resource that represents a request to create a Problem entity in HP SM. This is basically what the R2F use case of HP SX is called. It is presumed that the icoming request is issued by Propel portal but there is no dependency to it so the request could be issued by any kind of HP SX client.

HP SX content overview

In this overview section a general brief overview is repeated for readers convenience. The aspects of HP SX that are most important to understand the example implementain are highlighted.

HP SX in the HP Propel stack acts as a communication platform with fulfillment systems such as HP SM or CSA. As HP SM is so widely customizable HP SX cannot supply a one-size-fits-all solution. Usually, for a specific HP SM instance specific content matching has to be created.

HP SX content consists of two main parts:

- **OO Flow**
  - This describes high-level interaction state actions. For example, if a request is closed inform the submitter, or if an entity does

not exist yet, create it.

- **Adapter operation definitions**
  - These translate high-level operations, such as `createProblem`, into a sequence of low level system calls using REST or SOAP interfaces on a target system.

HP SX request processing:

1. The processing starts when a message arrives at an HP SX request REST endpoint. The incoming message is in the form of a JSON file containing information such as:

  - the fulfillment system type
  - the fulfillment system instance
  - the requestor
  - the requested items identification
  - selected user options.

  Based on this information HP SX decides which OO flow to invoke.

2. The OO flow usually starts by checking if the request already has an ID in the external system. If not, a `create` operation is invoked back on the SX adapter

3. The SX adapter performs the series of calls needed to create an entity in an external system, and as a last step usually registers to listen out for changes of the just-created entity. As part of the notification registration it determines which adapter operation will be used for the checking of the entity state.

4. When a change occurs in the external system, HP SX invokes a check operation. This check retrieves all the required information about the affected entity in the form of a JSON representing the state of the entity.

5. The OO flow is invoked again, taking this state as an input. The flow decides if the change is worthy of interest, and if any actions are needed. For example, it could invoke another operation or create and send an email notification.

6. The entity lifecycle normally ends in a closed state - if this state is detected, HP SX stops listening out for specific entity changes.

HP SM configuration

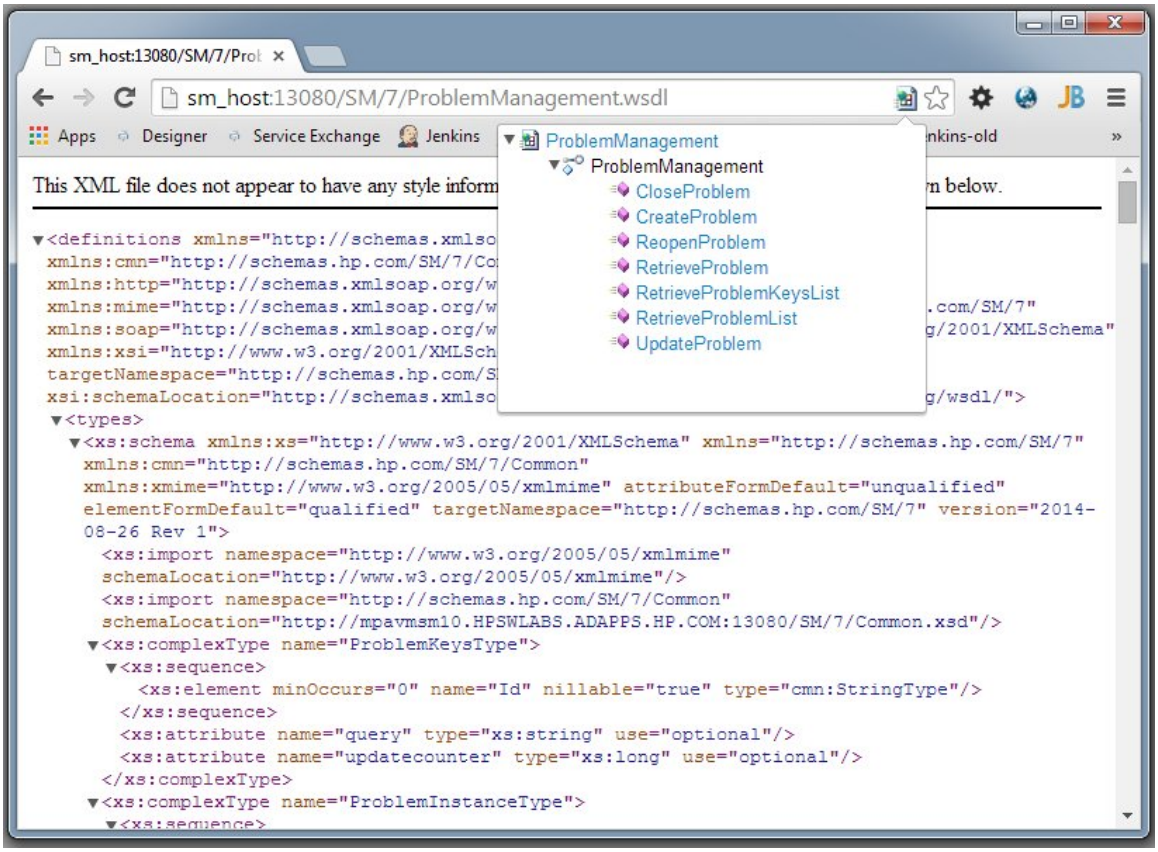
In HP SM Problem entities are kept in the rootcause table. This table is exposed via SOAP and REST endpoints as well. Find this configuration from the HP SM UI, by going to **Tailoring > Web services > Web service configuration**.

NOTE: The preferable method is to use REST endpoints. It may in some cases be a better option to use SOAP, so this is explained here too.

SOAP API configuration and testing

On the Web service configuration page you see that the Problem entity is exposed using a service called **ProblemManagement**. This service name also represents the WSDL name of your service. This WSDL is accessible using the following URL:
http://sm_host:13080/SM/7/ProblemManagement.wsdl

You can test the `create` operation using a SOAP client such as the Wizdler extension for Google Chrome. When it is installed, web service operations can be invoked using the wizdler icon at the end of the address bar, as you see in this screenshot:



When the `CreateProblem` operation is invoked without providing any input, the following errors occur:

- Please provide an Area.
- Please provide a Subarea.
- Please provide a Description.
- Invalid Assignment Group
- Message fc-1501 Could not be found:
- Please provide a Service.
- Please provide a Title.
- Please provide an Impact.
- Please provide an Urgency.

By default these mandatory fields are not exposed on the web service and need to be added manually.

Go to the Problem web service configuration and add fields according to the following table:

Field	Caption	Type
brief.description	Title	
description	Description	
initial.impact	Impact	
severity	Urgency	
subcategory	Area	
category	Category	
affected.item	AffectedService	
assignment	AssignmentGroup	

product.type	Subarea	

Now it is possible to create the problem entity without validation errors. Invoke the `CreateProblem` operation using Wizdler and pass the following message as an input:

Create Problem request

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
    <Body>
        <CreateProblemRequest xmlns="http://schemas.hp.com/SM/7">
            <model>
                <keys/>
                <instance>
                    <Title>Sample Problem</Title>
                    <Description>
                        <Description>It doesn’t work at all.</Description>
                    </Description>
                    <Impact>4</Impact>
                    <Urgency>2</Urgency>
                    <Service>Applications</Service>
                    <AssignmentGroup>Application</AssignmentGroup>
                    <Area>data</Area>
                    <Subarea>data or file incorrect</Subarea>
                </instance>
            </model>
        </CreateProblemRequest>
    </Body>
</Envelope>
```

The element `messages` now informs of the successful problem creation:

Response messages

```
<messages>
    <cmn:message type="String">Problem PM10031 has been opened.</cmn:message>
    <cmn:message type="String">Problem record added.</cmn:message>
</messages>
```

You can also see the ID of the newly created problem entity in the keys element:

Resposne Problem ID

```
<keys>
    <Id type="String">PM10031</Id>
</keys>
```

The newly created problem is also visible in the HP SM UI, for example using **Problem Management > Problem Control > Search Problems.**

REST API configuration and testing

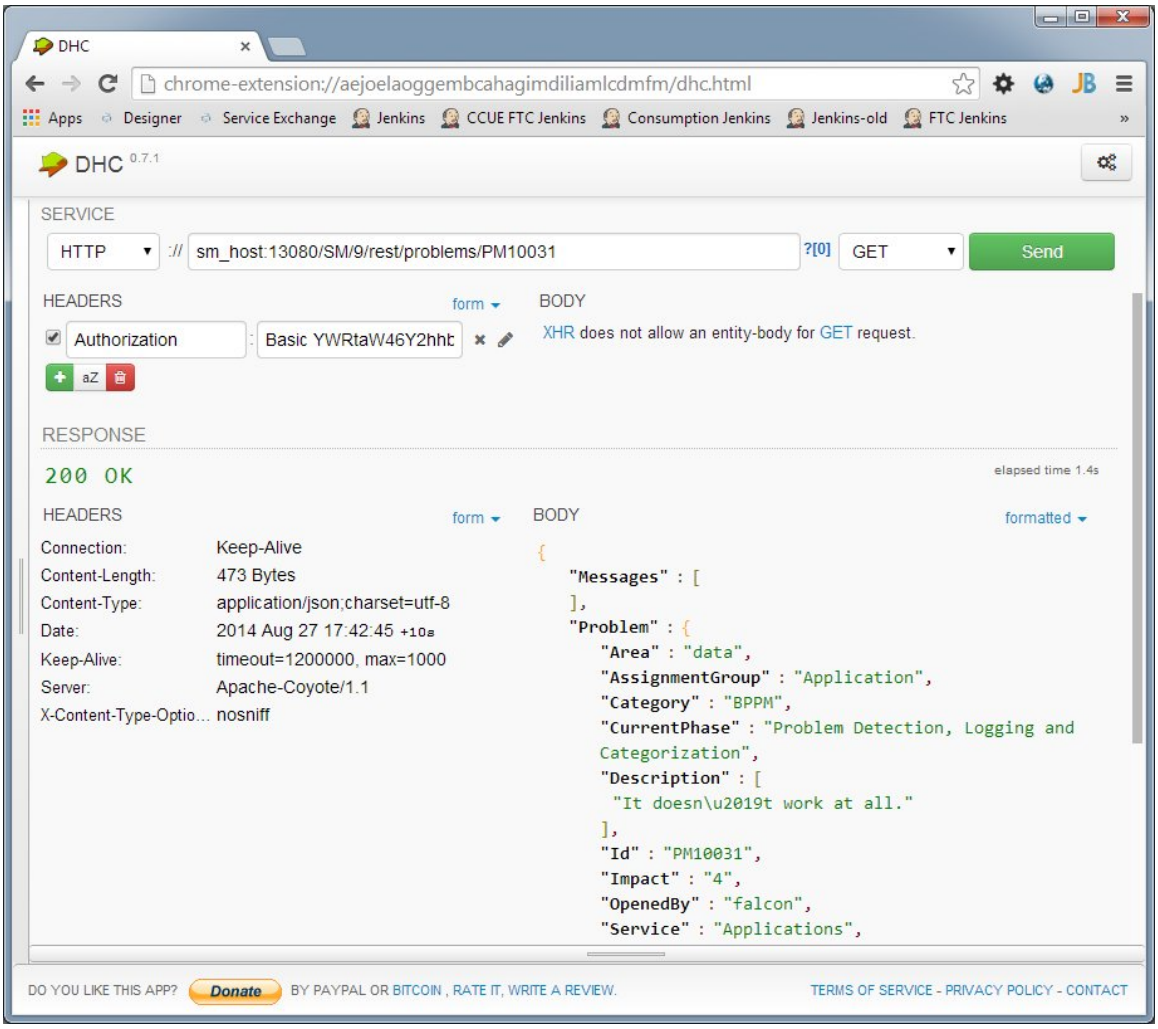
Since version 9.32, HP SM supports REST APIs. Use them for checking the Problem entity state in this example. To do this, expose a few more fields in the response.

The fields are summarized in the following table:

Field	Caption	Type
current.phase	CurrentPhase	
status	Status	
opened.by	OpenedBy	
root.cause	RootCause	
expected.resolution.time	Resolution Time	

In Problem web service you see that the rootcause table is exposed under the problem's collection name. The Rest URL of the detail of the just-created Problem entity looks like this:
http://<sm_host>:13080/SM/9/rest/problems/PM10031

Test the REST endpoint using a REST client. For example, the DHC – REST HTTP API Client available from the Google Chrome app store.



In order to get a proper response you need to provide an Authorization header containing the username and password of an HP SM operator user in your organization. The REST call should return properties of the Problem entity in the form of a JSON structure such as this:

Response Problem detail


```
{
    "Messages": [
    ],
    "Problem": {
        "Area": "data",
        "AssignmentGroup": "Application",
        "Category": "BPPM",
        "CurrentPhase": "Problem Detection, Logging and Categorization",
        "Description": [
            "It doesn\u2019t work at all."
        ],
        "Id": "PM10031",
        "Impact": "4",
        "OpenedBy": "falcon",
        "Service": "Applications",
        "Status": "Open",
        "Subarea": "data or file incorrect",
        "Title": "Sample Problem",
        "Urgency": "2"
    },
    "ReturnCode": 0
}
```


Additional API needed

In order to notify the submitter about the problem's solution the submitter's email address is needed. This can be retrieved using the existing operator SOAP API in HP SM. It is available in the **FSCManagement** web service. The `RetrieveOperator` operation takes the operator name as an input and returns the email address in the Email property.

Setup Problem entity update triggers

HP SX needs to know about any changes occurring with the problem entity in HP SM. This is done through update triggers added to relevant HP SM data tables: the table that stores the problem entity.

`lib.SX_EntityChangeV2.entityAfterUpdate`

The update triggers usually call common HP SX trigger code - `lib.SX_EntityChangeV2.entityAfterUpdate`. This procedure performs the following:

1. Checks whether the change is of interest, which requires listening to the changed entity. This information is stored in the **SxRegisteredEntitiesV2** table.
2. If there is some registration matching, the given entity ID of the new record is stored in the **SxEntityChangesV2** table.

HP SX performs periodic polling for records in **SxEntityChangesV2**. This polling is done through HP SM-defined external access to the table, which makes the polling a REST call. If change polling finds a record, HP SX starts processing it and removes the record when complete.

Defining an update trigger

In order to enable change listening for the Problem entity in this example, a new update trigger needs to be defined.

NOTE: A new trigger can only be defined using an HP SM standalone client, not through the web interface.

1. Go to **System Definitions > Tables >** rootcause.
2. Click **New…** in the **Associated triggers** section.
3. Enter a new trigger name, for example, SX.rootcause.after.update.

4. Choose **4-After** Update Trigger Type and enter the following text into the Script area:

```
lib.SX_EntityChangeV2.entityAfterUpdate('id', oldrecord, record);
```

5. Click **Save**.

This new trigger calls the HP SX common trigger code every time a Problem entity is created or updated.

Creating SM unload files

When HP SM customizations are complete, back them up somewhere and move them across all of your HP SM installations. HP SM supports "**unload**" files for this purpose.

You can define an unload on one system, export it, and then import it to another system.

Defining and creating an HP SM unload:

1. In the HP SM UI go to **System Administration > Ongoing Maintenance > Unload Manager > Create Unload**.

2. Enter values according to the following table:

Name	Value
Defect Id	SXProblem_1.01
Summary	SXProblem
Apps version	SM9.32
Hotfix type	Official

3. Define the content of your unload file. This is all that is necessary to change at this point (problems web service and new problem trigger):

Object Type	Query
triggers	trigger.name="SX.rootcause.after.update"
extaccess	service.name="ProblemManagement" and object.name="Problem"

4. Click **OK** to save the unload definition.

5. To download a new unload file now:

   a. Click **Proceed**.

   b. Provide a file name and click **Proceed** again.

Applying an HP SM unload

1. Go to **System Administration > Ongoing Maintenance > Unload Manager > Apply Unload**.

2. Choose the just-downloaded unload file.

3. Provide a backup file name.

4. *NOTE: There could be a conflict during the application of the unload which has to be solved manually. Perform the conflict resolution using the HP SM Client only, not the web interface.*

For each line in conflict:

   a. Double click the line in conflict and fix the content in the right column.

   b. When complete, check the **Reconciled** checkbox and click **Save**.

5. When all conflicts are reconciled, finish the unload application wizard.

At this point the HP SM instance is prepared, and you are ready to create the sample content.

Defining messages

HP SX is a message driven system. If you want something from HP SX your request has to pass in the form of a JSON message. These messages are consumed by your adapter definitions or OO. HP SX itself only needs to understand a few important message parts. They are **messageType** and **routing properties** (system type, target instance). For a `CreateProblem` operation you can reuse an existing message format, one used for R2F use cases.

An example CreateProblem message:

Create problem message

```
{
   "@type": "urn:x-hp:2014:software:cloud:data_model:sx:request",
   "messageType":"problem",
   "name" : "My problem",
   "description" : "Problem desc",
   "urgency" : "1",
   "items" : [
      {
        "route" : {
           "@type" : "urn:x-hp:2014:software:cloud:data_model:sx:route",
           "system_type" : "SM",
           "target_instance" : "http://<sm_host>:13080/SM"
        }
      }
   ]
}
```

This message will be available as a message source for HP SX's OO flow input bindings, as shown later. It will also be used as an input into an adapter `CreateProblem` operation. The create message is only one message going through HP SX in this example scenario. A representation of the Problem entity state also needs to be produced by the `checkProblem` operation. It is called every time the specific problem entity changes in HP SM, and again if it is used as an input for an OO flow. This message is fully under HP SX's control as it is not exposed outside its system.

This example chooses a simple JSON structure named `problemInfo`, containing all relevant properties. Notice also the message header in this example, which is system generated:

Problem status message

```
{
  "messageHeader" : {
    "backendSystemType" : "SM",
    "externalId" : "b91f09f4-0eb7-48ea-afa9-252f1d8d91ba",
    "messageType" : "problem",
    "targetInstance" : "mpavmsm10"
  },
  "problemInfo" : {
    "contact" : "falcon",
    "contactEmail" : "petr.fiedler@hp.com",
    "contactFullName" : "FALCON",
    "id" : "PM10032",
    "phase" : "Problem Prioritization and Planning",
    "status" : "Open",
    "title" : "My problem"
  }
}
```

HP SX content module

This example uses Apache Maven (http://maven.apache.org/index.html) for building the HP SX content pack.

At least two modules are needed, the first for the OO content pack and the second for the SX content pack. Also a common parent POM module needs to be defined for the two. The directory structure will look like this:

- sx-demo-content
  - content-sm-problem
    - pom.xml
  - oo-sm-problem-cp
    - pom.xml
  - pom.xml

In the parent `pom.xml` the version of dependent libraries is defined, also some further configuration is added:

Parent pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.hp.propel.serviceExchange</groupId>
    <artifactId>service-exchange-content-pom</artifactId>
    <version>1.1.0-SNAPSHOT</version>
    <packaging>pom</packaging>
    <name>Service Exchange Content</name>

    <properties>
        <oosdk.version>10.10.9</oosdk.version>
        <sx.version>1.0.1</sx.version>
        <sx-messaging.version>1.0.1</sx-messaging.version>
    </properties>
    <modules>
        <module>content-sm-problem</module>
        <module>oo-sm-problem-cp</module>
    </modules>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>com.hp.propel.serviceExchange</groupId>
                <artifactId>oo-sm-problem-cp</artifactId>
                <version>1.1.0-SNAPSHOT</version>
            </dependency>
            <dependency>
                <groupId>com.hp.ccue.serviceExchange</groupId>
                <artifactId>oo-sx-plugin</artifactId>
                <version>${sx-messaging.version}</version>
            </dependency>
        </dependencies>
    </dependencyManagement>
</project>
```

The properties element captures versions of HP OO SDK, core HP SX components, and the version of the SX plugin for HP OO. It is also necessary to populate the maven repository with SX and OO artifacts.

To do so, copy the artifacts from the SDK package `m2-repo` directory into the local maven repository in your home folder, usually at: c:\Users\`user_name`\.m2\repository for Windows or `/home/user_name/.m2/repository` for Linux.

OO flow

An OO flow in HP SX is used for high level business process modeling. In this example it is used for:

- problem creation
- submitter notification about problem closure.

NOTE: For OO flow creation, HP OO Studio 10.10 needs to be installed.

Configure OO Studio

Prior to starting, you need to add the SX OO plugin artifact (maven artifactId `oo-sx-plugin`) and all its dependencies into your OO internal maven repository. Copy your full local repository (`home/.m2/repository`) into the directory used by OO (`home/.oo/data/maven`). The following procedure presumes that you have already copied the `m2-repo` directory of the SDK package into your local maven repository.

The Base 1.1.1 OO Content pack needs to be uploaded into your HP OO studio. If it is not listed in the Content packs panel, upload it using the **I mport content pack** button on the panel toolbar.

Prepare Maven OO build

The OO project will be placed in the following directory:

`sx-demo-content/oo-sm-problem-cp/src/main/resources/oo-sm-problem-project`

The Maven `pom.xml` file is also needed for the `oo-sm-problem-cp` module. It is responsible for packaging the OO project into an OO Content pack deployable to an OO Central server. A Maven plugin from the OO SDK is used for this packaging.

OO Content Pack pom.xml

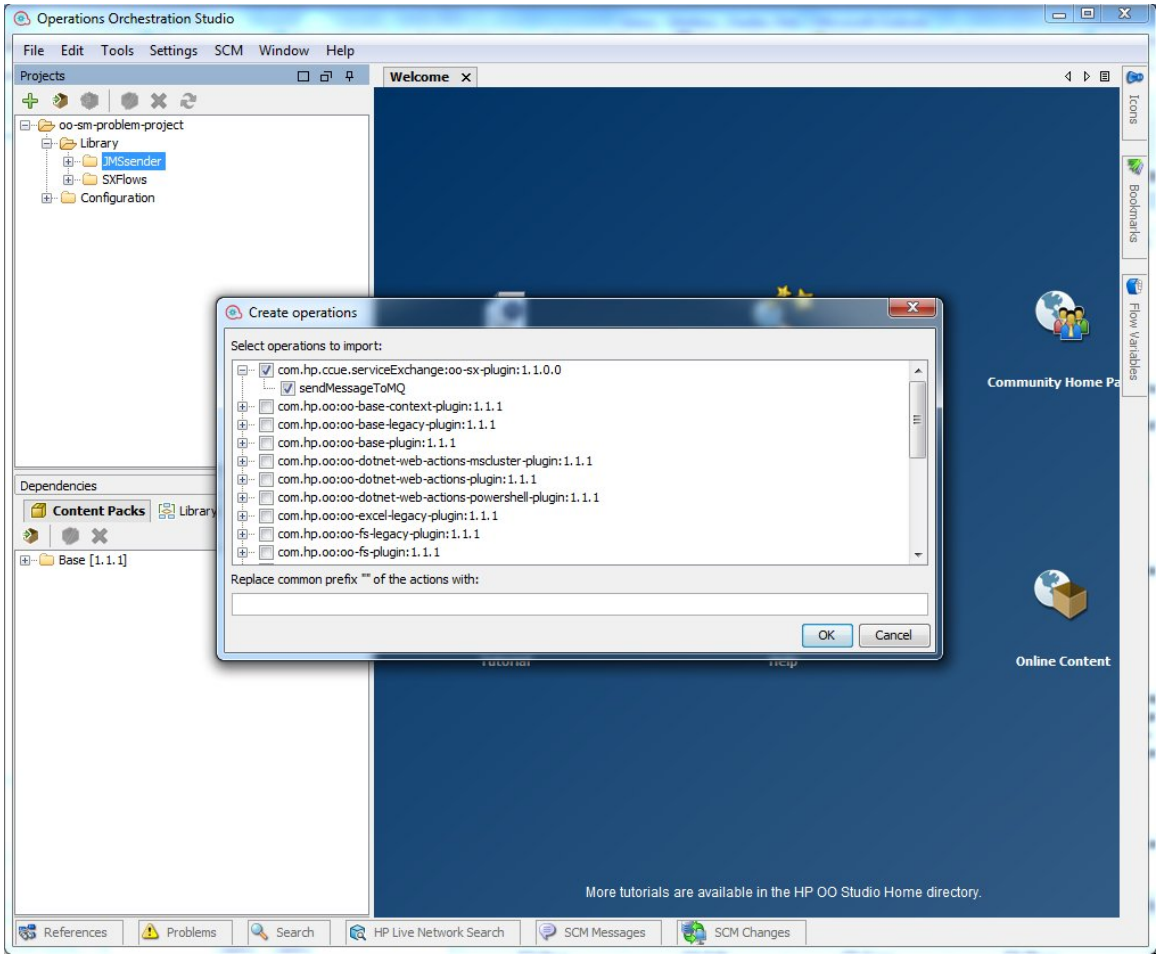
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>com.hp.propel.serviceExchange</groupId>
        <artifactId>service-exchange-content-pom</artifactId>
        <version>1.1.0-SNAPSHOT</version>
    </parent>
    <artifactId>oo-sm-problem-cp</artifactId>
    <packaging>jar</packaging>
    <dependencies>
        <dependency>
            <groupId>com.hp.ccue.serviceExchange</groupId>
            <artifactId>oo-sx-plugin</artifactId>
        </dependency>
    </dependencies>
    <build>
        <resources>
            <resource>

<directory>${project.basedir}/src/main/resources/oo-sm-problem-project</directory>
            </resource>
        </resources>
        <plugins>
            <plugin>
                <artifactId>maven-resources-plugin</artifactId>
                <version>2.6</version>
                <configuration>
                    <includeEmptyDirs>true</includeEmptyDirs>
                </configuration>
            </plugin>
            <plugin>
                <groupId>com.hp.oo.sdk</groupId>
                <artifactId>oo-contentpack-maven-plugin</artifactId>
```

```
                <version>${oosdk.version}</version>
                <executions>
                    <execution>
                        <id>generate-contentpack-plugin</id>
                        <phase>process-sources</phase>
                        <goals>
                            <goal>generate-contentpack</goal>
                        </goals>
                    </execution>
                </executions>
                <configuration>

<destinationFolder>${project.build.outputDirectory}</destinationFolder>
                    <artifactItems>
                        <artifactItem>
                            <groupId>com.hp.ccue.serviceExchange</groupId>
                            <artifactId>oo-sx-plugin</artifactId>
                            <version>${sx-messaging.version}</version>
                        </artifactItem>
                    </artifactItems>
                </configuration>
            </plugin>
        </plugins>
```

```
    </build>
</project>
```

Create an OO project

1. Open OO Studio.
2. Click the New Project icon in the **Projects** section.
3. Enter `oo-sm-problem-project` as a project name. It has to match the project directory name specified in the resources section of `pom.xml`.
4. Specify `sx-demo-content\oo-sm-problem-cp\src\main\resources` as the location.
5. Now OO Studio creates an empty project.
6. Under the **Library** folder of your project, create a new folder using the right mouse menu and choosing **New > Folder…** Name it JMSsender.
7. Under this folder create a new Operation. After a moment the **Create Operation** dialog appears listing all the available operations in the OO internal Maven repository. Choose **sendMessageToMQ** under com.hp.ccue.serviceExchange:oo-sx-plugin:1.1.0.0, as in the example below:



Define OO flow inputs

To design the flow:

1. Create a new folder under **Library**, call it for example SXFlows.

2. Choose **New > Flow** from the context menu for this folder. Enter a flow name such as SMProblemFlow. A new empty flow is now created.

3. Now to define input parameters. Input parameters are not required, however it is a good practice to declare them explicitly. Input parameters can be divided into three categories:

- SX internal

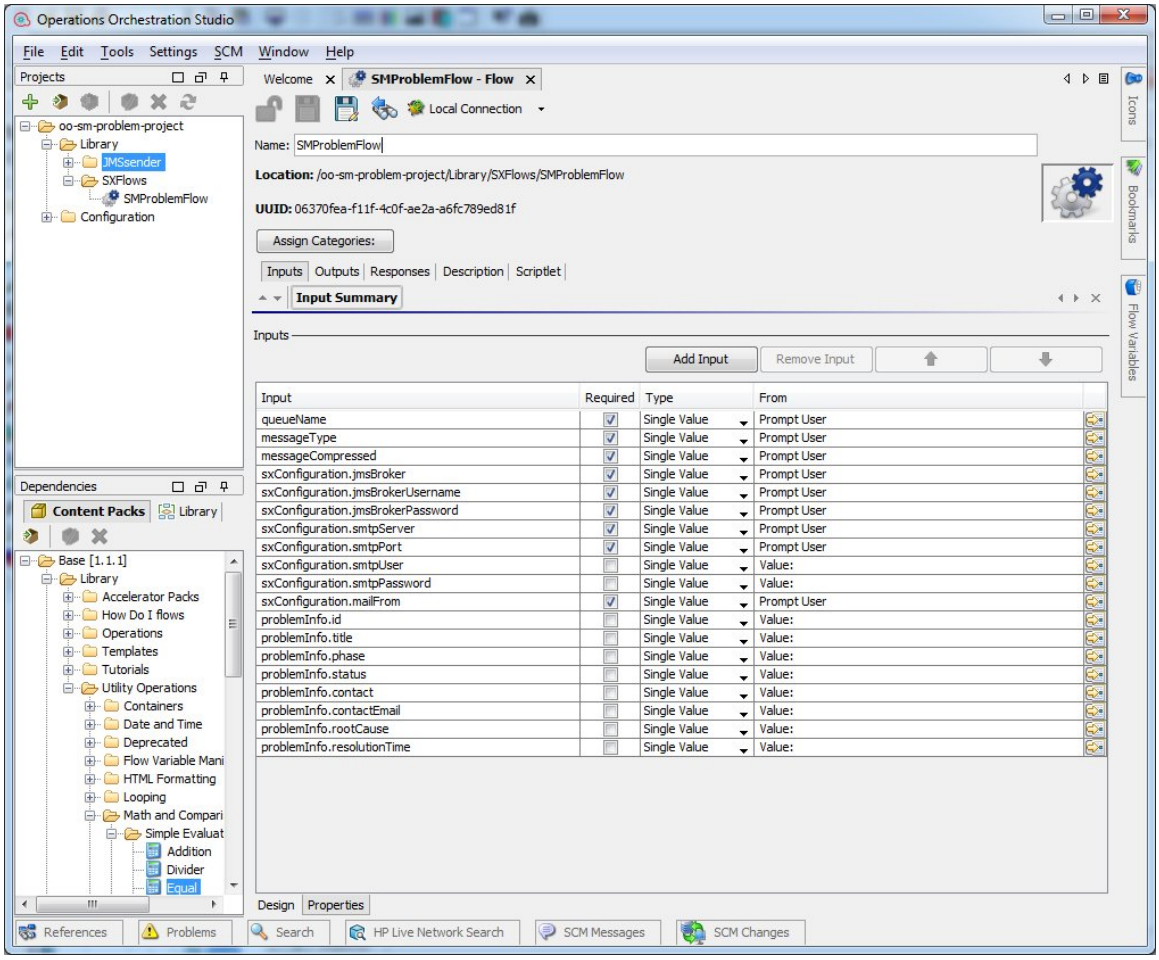
- SX configuration
- Custom ones.

4. Switch to the **Properties** tab and define the input parameters according to the following table:

Input	Required	From	Meaning
queueName	yes	Prompt User	Messaging queue used for communication with adapter.
messageType	yes	Prompt User	Identifies adapter pipeline used to handle message.
messageCompressed	yes	Prompt User	Contains message which will be passed back to SX, for performance reasons message is compressed.
sxConfiguration.jmsBroker	yes	Prompt User	Rabbit MQ hostname.
sxConfiguration.jmsBrokerUsername	yes	Prompt User	User name for authentication in Rabbit MQ.
sxConfiguration.jmsBrokerPassword	yes	Prompt User	Password used for authentication in Rabbit MQ.
sxConfiguration.smtpServer	yes	Prompt User	Mail server hostname.
sxConfiguration.smtpPort	yes	Prompt User	Mail server port.
sxConfiguration.smtpUser	no		Optional mail server username if authentication is switched on.
sxConfiguration.smtpPassword	no		Optional mail server password if authentication is switched on.
sxConfiguration.mailFrom	yes	Prompt User	Sender address used for notification emails.
problemInfo.id	no		ID of problem entity for HP SM or blank for new entity.
problemInfo.title	no		Problem title.
problemInfo.phase	no		Problem workflow current phase.
problemInfo.status	no		Problem status such as Open or Closed.
problemInfo.contact	no		Submitter full name.
problemInfo.contactEmail	no		Submitter email address.
problemInfo.rootCause	no		Problem solution description.
problemInfo.resolutionTime	no		Problem resolution time.

NOTE: These inputs have to match the definitions provided in the `flows.json` file, in the adapter content step created later.

The screenshot below shows fully defined flow inputs:



Design OO flow

Now the flow inputs are defined you can start the OO flow design.

The flow is responsible for two main things:

1. It invokes the create operation on the adapter if the problem entity is new.

2. It notifies the submitter via email if the problem is closed.

The initial node of the flow checks if the `problemInfo.id` parameter is empty or not. Use the Equal operation from the Base content for this. Choose **Base/Library/Utility Operations/Math** and **Comparison/Simple Evaluators/Equal** from the Content Pack's panel and drag them into the flow design area. Fill in inputs according to this table:

Input	Assign from Variable	Otherwise	Constant Value
value1	<not assigned>	Use Constant	${problemInfo.id}
value2	<not assigned>	Use Constant	
operation	<not assigned>	Use Constant	==

When the ID check succeeds the flow will send a message back to HP SX, using the sendMessageToMQ operation created earlier. Drag it from the **Projects** section and move it into the design area. Enter the input bindings according to the following table:

Input	Assign from Variable	Otherwise	Constant Value

brokerUrl	sxConfiguration.jmsBroker	Use Constant	
brokerUsername	sxConfiguration.jmsBrokerUsername	Use Constant	
brokerPassword	sxConfiguration.jmsBrokerPassword	Use Constant	
queueName	queueName	Use Constant	
operationName	<not assigned>	Use Constant	createProblem
messageText	<not assigned>	Use Constant	
messageCompressed	messageCompressed	Use Constant	
messageType	messageType	Use Constant	

When the ID check fails (ID is not empty), another check has to be performed. This time it needs to check if the problem status is **Closed**. Again, use the Equal operation from the Base content pack. Enter the input bindings according to the following table:

Input	Assign from Variable	Otherwise	Constant Value
value1	<not assigned>	Use Constant	${problemInfo.status}
value2	<not assigned>	Use Constant	Closed
operation	<not assigned>	Use Constant	==

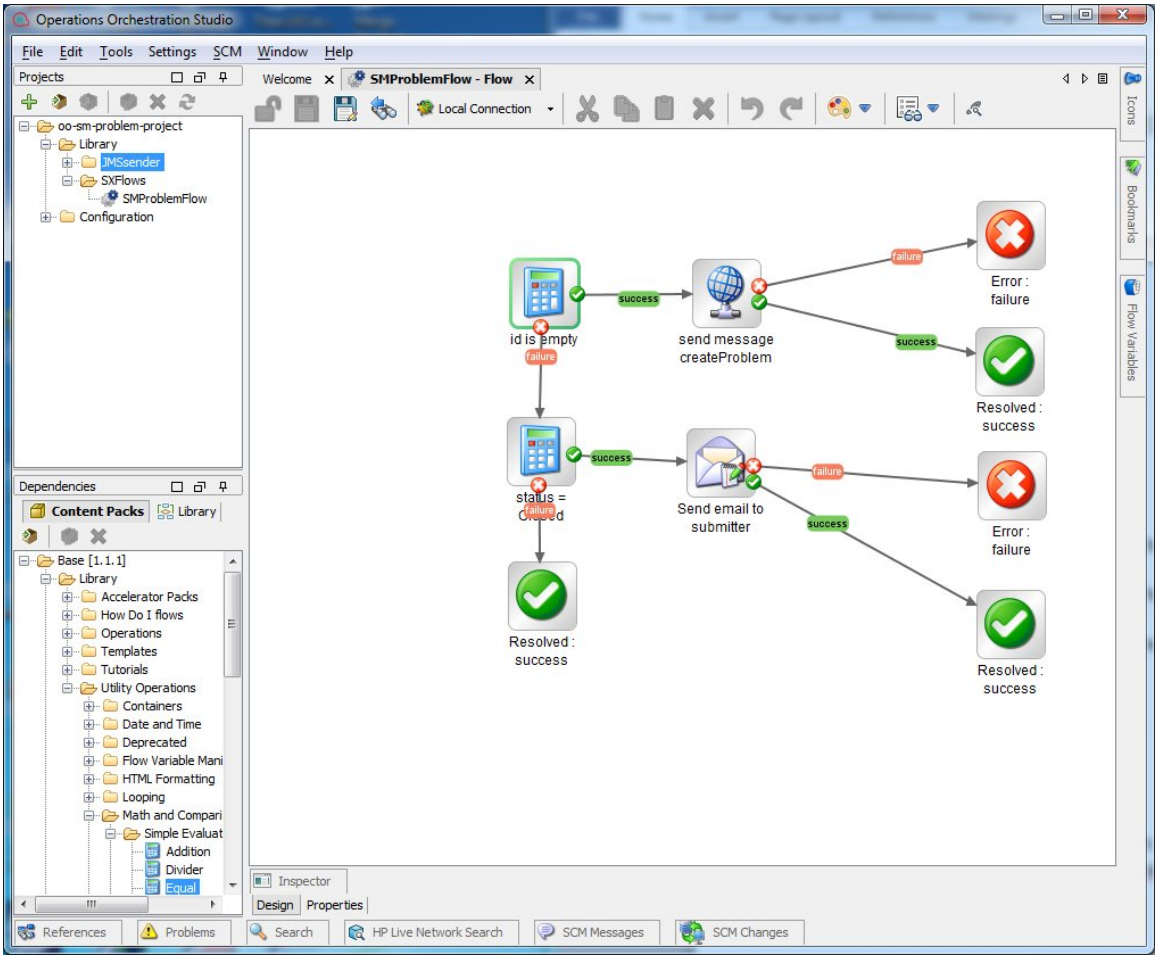
When the status is **closed** HP SX will notify the submitter that this is the case via email. Drag the **Base/Library/Operations/Email/Send Mail** co mponent from the **Content Pack** panel. Input bindings are described in the following table:

Input	Assign from Variable	Otherwise	Constant Value
hostname	<not assigned>	Use Constant	${sxConfiguration.smtpSever}
port	<not assigned>	Use Constant	${sxConfiguration.smtpPort}
from	<not assigned>	Use Constant	${sxConfiguration.mailFrom}
to	<not assigned>	Use Constant	${problemInfo.contactEmail}
subject	<not assigned>	Use Constant	Problem was solved: ${problemInfo.title}
body	<not assigned>	Use Constant	See bellow
htmlEmail	<not assigned>	Use Constant	true
username	<not assigned>	Use Constant	${sxConfiguration.smtpUser}
password	<not assigned>	Use Constant	${sxConfiguration.smtpPassword}

Email body

```
<html>
    <body>
        <h3>Your problem was solved: ${problemInfo.title}</h3>
        <table border="1">
            <tr>
                <td>Root Cause</td>
                <td>${problemInfo.rootCause}</td>
            </tr>
            <tr>
                <td>Resolution Time</td>
                <td>${problemInfo.resolutionTime}</td>
            </tr>
        </table>
    </body>
</html>
```

With the key flow steps configured, the last step is to connect them all together and add the appropriate flow results. The screenshot below shows the fully designed flow:



The flow is now complete. In order to test it you need to design an adapter operation, as described in next section.

HP SX Adapter configuration

This section describes how to implement an HP SX adapter configuration. I.e. it describes the configuration that needs to be present in the content pack so that the HP SM adapter uses the content.

Prepare Maven content build

The content project will be placed in the directory:

`sx-demo-content/content-sm-problem/src/main/resources`.

The module's `pom.xml` is responsible for packaging our content together with the OO content pack into one content archive, uploadable into a running HP SX instance.

SX Content Pack pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <artifactId>service-exchange-content-pom</artifactId>
        <groupId>com.hp.propel.serviceExchange</groupId>
        <version>1.1.0-SNAPSHOT</version>
    </parent>
    <artifactId>content-sm-problem</artifactId>
    <dependencies>
        <dependency>
            <groupId>com.hp.propel.serviceExchange</groupId>
            <artifactId>oo-sm-problem-cp</artifactId>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-dependency-plugin</artifactId>
                <executions>
                    <execution>
                        <id>copy-dependencies</id>
                        <phase>compile</phase>
                        <goals>
                            <goal>copy-dependencies</goal>
                        </goals>
                        <configuration>

<outputDirectory>${project.build.directory}/classes/oo</outputDirectory>
                            <overWriteReleases>false</overWriteReleases>
                            <overWriteSnapshots>false</overWriteSnapshots>
                            <overWriteIfNewer>true</overWriteIfNewer>
                            <excludeTransitive>true</excludeTransitive>

<includeGroupIds>com.hp.propel.serviceExchange</includeGroupIds>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

Content pack structure

The content pack structure consists of three JSON configuration files:

1. A JSON configuration file for the content pack metadata definition.

2. A JSON configuration file for the flow invocation.

3. A JSON configuration file for the operation step definitions.

There are also Freemarker template files for the transformation of requests and responses from external system APIs. They can also contain HP SM unload files and other proprietary content not directly interpreted by HP SX.

The file structure under the `sx-demo-content/content-sm-problem/src/main/resources` directory should look like this:

- sm
    - SXProblem.unl (exported from SM earlier)
- oo (OO content pack is placed here during the build, it does not have to exist in the source directories)
- sx
    - flows.json
    - operations.json
    - templates
        - Freemarker templates
- metadata.json

`Metadata.json` contains basic content pack metadata such as id, name, description and version. It also contains the version of the nested HP SM unload file.

Content pack metadata

```
{
    "id": "sm-problem",
    "name": "SM problem demo content",
    "description": "Demo Service Exchange content dealing with problem entity
lifecycle in Service Manager",
    "version": "1.0.0",
    "adapter": "SM",
    "features":[
    ],
    "files": [
        {
            "path": "sm/SXProblem.unl",
            "version": "1.01",
            "type": "sm_unload"
        }
    ]
}
```

Flow configuration

Flow configuration is expressed in a `flows.json` file. It specifies which flow will be invoked for problem entity operations, and bindings of flow inputs to different information sources in HP SX. These could be input messages or various configuration files. The Flow ID could be obtained from the **Properties** tab of the SMProblemFlow detail page in **OO Studio**. For each flow input parameter one structure in the parameters list must be provided. The structure contains names which have to match the flow input names. It also contains source property, possible values are described in the following table:

Source	Purpose
message	Used for values coming from an SX input message. This has a default value when no source is provided.
infrastructure	Provides infrastructure configuration values such as messaging server connection settings, or various SX REST endpoint URLs. The configuration is in the `sx.war/WEB-INF/classes/config/infrastructure.json` file.

oo-properties	Contains OO configuration properties such as SMTP server settings. The configuration is in the `sx.war/WEB-INF/classes/config/oo/properties.json` file.

The **ValueSelector** property holds a JSONPath expression selecting a specific value from a selected source. It has a similar syntax to XPath. It usually starts with **$** and contains dot delimited property names. For example, if you want to select a contact email value from a problem status message, JSONPath will look lthis way: `$.problemInfo.contactEmail`. See http://code.google.com/p/json-path/ for more examples. It is necessary to bind several properties for JMS settings (used for passing messages back to HP SX), SMTP server settings (used for sending email notifications), and Problem entity status properties from the status message defined earlier.

```
{
    "SM": {
        "problem": {
            "flowId": "06370fea-f11f-4c0f-ae2a-a6fc789ed81f",
            "compressMessage": true,
            "parameters": [
                {
                    "name": "sxConfiguration.jmsBroker",
                    "valueSelector": "$.JMS_BROKER.endpoint",
                    "source": "infrastructure"
                },
                {
                    "name": "sxConfiguration.jmsBrokerUsername",
                    "valueSelector": "$.JMS_BROKER.loginName",
                    "source": "infrastructure"
                },
                {
                    "name": "sxConfiguration.jmsBrokerPassword",
                    "valueSelector": "$.JMS_BROKER.password",
                    "source": "infrastructure"
                },
                {
                    "name": "sxConfiguration.smtpServer",
                    "valueSelector": "$.smtpServer",
                    "source": "oo-properties"
                },
                {
                    "name": "sxConfiguration.smtpPort",
                    "valueSelector": "$.smtpPort",
                    "source": "oo-properties"
                },
                {
                    "name": "sxConfiguration.smtpUser",
                    "valueSelector": "$.smtpUser",
                    "source": "oo-properties"
                },
                {
                    "name": "sxConfiguration.smtpPassword",
                    "valueSelector": "$.smtpPassword",
                    "source": "oo-properties"
                },
                {
                    "name": "sxConfiguration.mailFrom",
                    "valueSelector": "$.mailFrom",
                    "source": "oo-properties"
                },
                {
                    "name": "sxConfiguration.emailBcc",
                    "valueSelector": "$.emailBcc",
```

```
            "source": "oo-properties"
        },
        {
            "name": "problemInfo.id",
            "valueSelector": "$.problemInfo.id",
            "source": "message"
        },
        {
            "name": "problemInfo.title",
            "valueSelector": "$.problemInfo.title",
            "source": "message"
        },
        {
            "name": "problemInfo.phase",
            "valueSelector": "$.problemInfo.phase",
            "source": "message"
        },
        {
            "name": "problemInfo.status",
            "valueSelector": "$.problemInfo.status",
            "source": "message"
        },
        {
            "name": "problemInfo.contactFullName",
            "valueSelector": "$.problemInfo.contactFullName",
            "source": "message"
        },
        {
            "name": "problemInfo.contactEmail",
            "valueSelector": "$.problemInfo.contactEmail",
            "source": "message"
        },
        {
            "name": "problemInfo.rootCause",
            "valueSelector": "$.problemInfo.rootCause",
            "source": "message"
        },
        {
            "name": "problemInfo.resolutionTime",
            "valueSelector": "$.problemInfo.resolutionTime",
            "source": "message"
        }
    ]
}
```

```
    }
}
```

Operations configuration and templates

Two operations are needed for the example content:

1. `createProblem` will be responsible for the creation of the **Problem** entity in HP SM.

2. `checkProblem` will retrieve the current status of the entity from HP SM.

For each of these operations it is necessary to define a sequence of steps that correspond to the individual SOAP or REST calls invoked on the target HP SM instance.

CreateProblem operation

For the create operation, one SOAP request (the **CreateProblemRequest** that was tested earlier) is needed. Typically for SOAP calls it is necessary to provide the `requestUrlTemplate` to get the URL, the `requestTemplate` for request transformation, the response template for the server response transformation, and a few http headers as well.

Create problem operation

```
{
"createProblem": [
        {
            "label": "Create problem",
            "requestUrlTemplate": "smSoapUrl.ftl",
            "requestTemplate": "createProblem.ftl",
            "responseTemplate": "createProblemResponse.ftl",
            "header-SOAPAction": "Create",
            "header-Accept": "text/xml"
        },
        …
    ]
}
```

The result of the request transformation should be a SOAP request containing all the relevant information, which is then sent to the HP SM server. In this example problem title, description and urgency values are entered.

Templates are written using Freemarker, a powerful template language good for generating text output. A Freemarker manual is available here: http://freemarker.org/docs/index.html.

Each transformation takes input in the form of structured hash (or **Map** in Java terminology.) It contains key value pairs. **Key** is always a string, **value** could be a primitive type (String, Boolean, Number or another Map or List.)

In your template include values from the input using the following syntax: `${expression}`. An **expression** could be a simple dot delimited key name or something more complicated. See the Freemarker Manual expressions section of http://freemarker.org/docs/dgui_template_exp.html#exp_cheatsheet for a full reference.

For more complicated template operations such as iteration over lists or conditions there are Freemarker directives. They have xml-like element syntax, and the element name starts with the # character. In the example below an escape directive is used, which ensures that all expression results included into the template are properly encoded for XML output. For example, **<** is replaced by the XML entity **>**. For a full directive reference see http://freemarker.org/docs/ref_directive_alphaidx.html

NOTE: The message is available in the template input under the message key.

createProblem template

```
<#-- @ftlvariable name="message" type="java.util.Map" -->
<#escape x as x?xml>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
    <Body>
        <CreateProblemRequest xmlns="http://schemas.hp.com/SM/7">
            <model>
                <keys/>
                <instance>
                    <Title>${message.name}</Title>
                    <Description>
                        <Description>${message.description}</Description>
                    </Description>
                    <Impact>4</Impact>
                    <Urgency>${message.urgency}</Urgency>
                    <Service>Applications</Service>
                    <AssignmentGroup>Application</AssignmentGroup>
                    <Area>data</Area>
                    <Subarea>data or file incorrect</Subarea>
                </instance>
            </model>
        </CreateProblemRequest>
    </Body>
</Envelope>
</#escape>
```

After the SOAP request is processed by an HP SM server, HP SX will receive back a SOAP response. Usually it is necessary to be able to use information from the response in the later steps of the operation. It is a task for response template transformation. It is passed the original message and server response as an input. Freemarker has native support for working with XML inputs, for example XPath queries can be used for selecting values from the response. The result of response transformation has to be a JSON structure. HP SX will merge this structure into the original message and use the result as an input for the next operation step. See the `ftl` directive defining the XML namespaces in the example below. They are then used in an XPath query selecting the ID of the just-created problem entity. ID is placed under the id property of the problemInfo structure. Also note that the server response is available in the template input under the doc.result key

createProblemResponse template

```
<#ftl ns_prefixes={
    "soap":"http://schemas.xmlsoap.org/soap/envelope/",
    "sm":"http://schemas.hp.com/SM/7"}
>
<#escape x as x?json_string>
{
    "problemInfo":{
        "id":
"${doc.result["soap:Envelope/soap:Body/sm:CreateProblemResponse/sm:model/sm:keys/sm:Id"]
}
}
</#escape>
```

To summarize: after the first step of the `createProblem` operation, a problem entity is created in HP SM and its ID is made available in the input message.

The goal of the next step is to register HP SX as a listener for any problem entity changes. To make this happen some information is needed.

Property	Description
notifyTemplate	This template is used to generate the input message that is later passed as the input for a checkProblem operation every time the problem entity is changed in HP SM.
callbackTemplate	This template creates a notification for the Propel catalog when a change occurs in HP SM.
operationName	Name of the operation invoked when the problem entity changes.
idSelector	JSONPath expression selecting the ID of the problem entity being watched.
firstRunImmediately	If set to true, the first check operation executes immediately.
entityTypeSelector	Name of the HP SM data table containing the problem entity.

Create problem operation cont.

```
{
"createProblem": [
        …
        {
            "label": "Watch interaction changes",
            "notifyTemplate": "checkProblem.ftl",
            "callbackTemplate": "callbackNotify.ftl",
            "operationName": "checkProblem",
            "idSelector": "$.problemInfo.id",
            "firstRunImmediately": true,
            "entityTypeSelector": "rootcause"
        }
    ]
}
```

The `createProblem` operation is now complete..

CheckProblem operation

The goal of the `checkProblem` operation is to retrieve various problem entity properties from HP SM. This time the SM REST interface will be used. For REST calls you first have to obtain the URL in `requestUrlTemplate`. The http method also needs to be provided, which is GET in this case. In addition `responseTemplate` is needed.

Check problem operation

```
{
"checkProblem": [
        {
            "label": "Retrieve problem",
            "requestUrlTemplate": "retrieveProblemUrl.ftl",
            "responseTemplate": "retrieveProblemResponse.ftl",
            "method": "GET",
            "header-Accept": "application/json"
        },
        …
```

The Request URL template takes input consisting of message and instance configuration (in this example an HP SM instance.) The instance configuration is used as the base URL and appended by the HP SM REST problem collection context. At the end there is the ID of a specific problem entity.

```
<#escape x as x?url>
<#noescape>${instanceConfig.endpoint}</#noescape>/9/rest/problems/${message.problemInfo.id}
</#escape>
```

For the http GET operation there is no request body, so the request template is missing here. The Response transformation is easier also, as HP SM produces a JSON response which easily maps into Freemarker hash input. Notice that Freemarker is strict about `${expression}` results. If the expression produces null, it causes a processing error. See the usage of the `#if` directive bellow which deals with this problem.

retrieveProblem response template

```
<#escape x as x?json_string>
{
    "problemInfo": {
        "title": "${doc.result.Problem.Title}",
        "phase": "${doc.result.Problem.CurrentPhase}",
        "status": "${doc.result.Problem.Status}",
        "contact": "${doc.result.Problem.OpenedBy}"
        <#if doc.result.Problem.RootCause?? && doc.result.Problem.RootCause?size &gt;
0>
            ,"rootCause": "${doc.result.Problem.RootCause?join("\n")}"
        </#if>
        <#if doc.result.Problem.ResolutionTime??>
            ,"resolutionTime": "${doc.result.Problem.ResolutionTime}"
        </#if>
    }
}
</#escape>
```

The next step in the `checkProblem` operation is to fetch the email address and the full name of the given operator (`Contact` property.) The existing SOAP API is used here.

Check problem operation cont.

```
{
"checkProblem": [
       …
        {
            "label": "Retrieve submitter email",
            "requestUrlTemplate": "smSoapUrl.ftl",
            "requestTemplate": "retrieveOperator.ftl",
            "responseTemplate": "retrieveOperatorResponse.ftl",
            "header-SOAPAction": "Retrieve",
            "header-Accept": "text/xml"
        }
    ]
```

The request template fills the operator name into the request.

retrieveOperator template

```
<#escape x as x?xml>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
    <Body>
        <RetrieveOperatorRequest xmlns="http://schemas.hp.com/SM/7">
            <model>
                <keys>
                    <Name>${message.problemInfo.contact}</Name>
                </keys>
                <instance/>
            </model>
        </RetrieveOperatorRequest>
    </Body>
</Envelope>
</#escape>
```

The response template extends the `problemInfo` structure by the full name and email.

retrieveOperator template

```
<#ftl ns_prefixes={
    "soap":"http://schemas.xmlsoap.org/soap/envelope/",
    "sm":"http://schemas.hp.com/SM/7"}
>
<#escape x as x?json_string>
{
    "problemInfo":{
        "contactFullName":
"${doc.result["soap:Envelope/soap:Body/sm:RetrieveOperatorResponse/sm:model/sm:instance/s
"contactEmail":
"${doc.result["soap:Envelope/soap:Body/sm:RetrieveOperatorResponse/sm:model/sm:instance/s
}
}
</#escape>
```

Callback notification

The goal of the notification callback is to inform the catalog about changes in HP SM. For this example it is not needed as the HP Propel Catalog is not aware of the Problem entity and therefore will ignore such notifications.

However, this functionality is explained here in case it is required.

The following statuses are reported in notifications:

Status	Description
submitted	Problem in HP SM is in phase 'Problem Detection, Logging and Categorization' or 'Problem Prioritization and Planning'
completed	Problem in HP SM is closed
in_progress	For all other Problem phases

Notifications have to contain an HP SX request ID, and can in addition contain other optional attributes, for example displayName or externalId.

```
<#escape x as x?json_string>
{
   "@type": "urn:x-hp:2014:software:cloud:data_model:sx:notification",
   "id": "${message.messageHeader.externalId}",
   "remoteId": "${message.problemInfo.id}",
   "displayName": "${message.problemInfo.title}",
   "submitter": "${message.problemInfo.contact}",
    <#if message.problemInfo.status == 'Closed'>
       "state" : "completed"
    <#elseif message.problemInfo.phase == 'Problem Detection, Logging and
Categorization' || message.problemInfo.phase == 'Problem Prioritization and Planning'>
       "state" : "submitted"
    <#else>
       "state" : "in_progress"
    </#if>
}
</#escape>
```

Testing and Troubleshooting

At this point the HP SX example content is complete, it now needs to be tested.

To build the content, call:

```
mvn clean install
```

The newly built content pack will be in:

`content-sm-problem\target\content-sm-problem-1.1.0-SNAPSHOT.jar`.

Deploy it either using the Content management UI, or the HP SX content upload maven plugin.

Content management UI

The Content Management UI is available using the following URL:

`https://sx_host:8444/sx/contentManagement.jsp`

Procedure to upload a content pack:

a) Upload the content pack using the **Upload** button.

b) Select the `content-sm-problem\target\content-sm-problem-1.1.0-SNAPSHOT.jar` file.

c) Click **OK**.

Content upload maven plugin

Another way of uploading content is by using the HP SX content upload maven plugin from the command line. In order to do this you need a properly configured `sx-maven-plugin` in your root `pom.xml` file. The configuration contains the following properties:

Property	Default value	Description
idmUrl	https://catalog_host:8444/idm-service	URL of the IdM service on the Propel catalog machine

idmTransportUser	idmTransportUser	User used for authorization of API calls on the IdM service
idmTransportPassword	idmTransportUser	User password used for authorization of API calls on the IdM service
sxUrl	https://sx_host:8444/sx	URL of SX service on HP Propel SX machine
username	admin	HP SX Administrator User with permission to upload SX content packs - it has to have the ADMINISTRATOR role within the selected tenant
password	cloud	Password of the HP SX Administrator User
tenant	Provider	Organization name of the HP SX Administrator User

When configured properly it is possible to upload an HP SX content pack using the following command, executed in the `content-sm-problem` directory:

```
mvn install com.hp.ccue.serviceExchange:sx-maven-plugin:uploadContent
```

When the operation completes content metadata is returned by the HP SX server. You can find it in `sx.log`.

Content upload maven plugin output

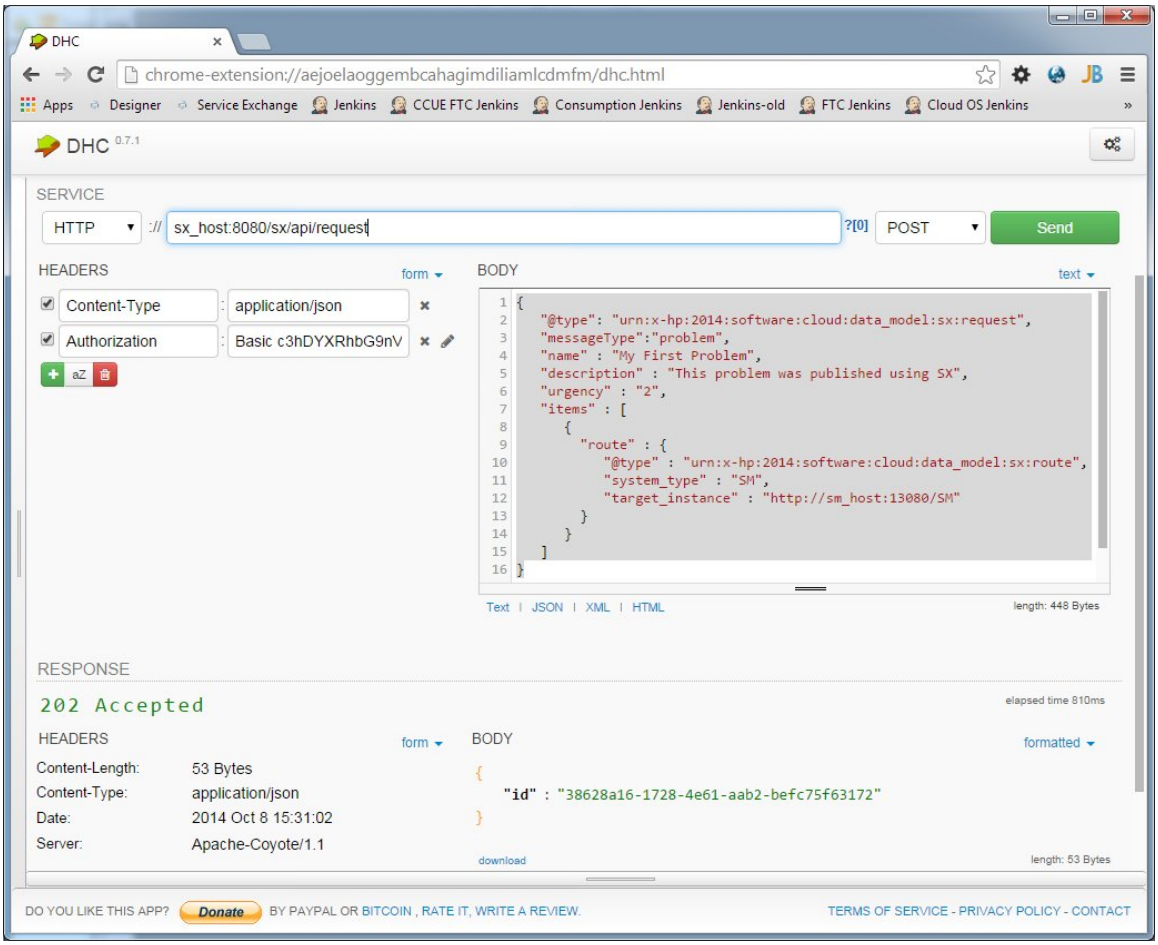

```
{
  "id" : "sm-problem",
  "files" : [ {
    "path" : "sm/SXProblem.unl",
    "version" : "1.01",
    "type" : "sm_unload"
  } ],
  "description" : "Demo Service Exchange content dealing with problem entity lifecycle
in Service Manager",
  "name" : "SM problem demo content",
  "features" : [ ],
  "ooContent" : {
    "name" : "oo-sm-problem-project",
    "version" : "1.0.0"
  },
  "uploadTime" : "2014-09-12T16:59:30+0200",
  "adapter" : "SM",
  "version" : "1.0.0",
  "_links" : {
    "self" : {
      "href" : "/sx/api/content/sm-problem"
    }
  }
}
```


Sometimes it is useful (no change in OO flow) and faster, to upload HP SX content and not update the OO flow in the OO Central server. To do so use the `-DskipOOUpload=true` parameter from the command line:

```
mvn install com.hp.ccue.serviceExchange:sx-maven-plugin:uploadContent
-DskipOOUpload=true
```

Testing using the SX REST interface

When the new content pack is uploaded you can use for example the DHC client again, to test it.



A POST request has to be submitted to the http://sx_host:8080/sx/api/request endpoint. The request content has to contain the JSON message described earlier.

Create problem request

```
{
   "@type": "urn:x-hp:2014:software:cloud:data_model:sx:request",
   "messageType":"problem",
   "name" : "My First Problem",
   "description" : "This problem was published using SX",
   "urgency" : "2",
   "items" : [
      {
        "route" : {
           "@type" : "urn:x-hp:2014:software:cloud:data_model:sx:route",
           "system_type" : "SM",
           "target_instance" : "http://sm_host:13080/SM"
        }
      }
   ]
}
```

If the request is successful, HP SX will return a request ID.

Create problem response

```
{
    "id":"38628a16-1728-4e61-aab2-befc75f63172"
}
```

In case of an error, troubleshoot the issue by inspecting the log files, see the HP SX log files section.

HP SX log files

Log files for the HP SX instance are placed in `JBOSS_HOME/standalone/logs` directory.

There are several files containing log messages from different components and using different log levels:

File name	Purpose
sx.log	General SX log containing info and error messages from all components
sx-messages.log	Contains incoming messages as they entered SX
notification.log	Outgoing catalog notification log
adapter type-messages.log	Detailed log for given adapter type, containing full communication with external system (request, responses, operation inputs/outputs,...) . For example, an HP SM file is named `sm-messages.log`
sx-trace.log	Detailed log aggregating trace messages from all adapters

When an issue occurs, checking the `sx.log` is a good starting point as overall information about what HP SX is doing, together with all errors (if they occurred), is contained there.

For a more detailed analysis of issues communicating with external systems, use the adapter-specific log file. For HP SM it is the file `sm-messages.log`. In the log shown below you can see the create problem message dispatches successfully. At first HP SX sent a message saying 'execute problem OO flow into Rabbit MQ'. Then the OO message listener picked up this message and executed the flow on the OO Central server. Based on the message the OO flow decided that CreateProblem needs to be invoked back on SX, and so it is executed. In addition, the registration states that the entity status should be checked immediately, so a checkProblem operation is executed too. As a result of this operation a notification is issued saying that the problem is in the **submitted** state.

Create problem in sx.log

```
2014-10-08 17:28:03.450 INFO  [com.hp.ccue.serviceExchange.oo.OoUtils] - Compressing
oo message original:680 compressed:472 ratio:69.411766
2014-10-08 17:28:03.463 INFO  [com.hp.ccue.serviceExchange.jms.RabbitMqSenderImpl] -
connecting to rabbit_sx@mpavmoo02.hpswlabs.adapps.hp.com
2014-10-08 17:28:03.780 INFO  [com.hp.ccue.serviceExchange.rest.RequestResource] - SX
response: {
  "id" : "942aec34-80eb-47e1-918e-ac21832e485a"
}
2014-10-08 17:28:04.097 INFO  [com.hp.ccue.serviceExchange.oo.OoFlowMessageListener] -
OO flow successfully executed, uri =
http://mpavmoo02.hpswlabs.adapps.hp.com:8080/oo/rest/executions/221752429/summary
2014-10-08 17:28:04.261 INFO  [com.hp.ccue.serviceExchange.adapter.sm.SmAdapter] -
executing pipeline PLAIN
2014-10-08 17:28:04.264 INFO
[com.hp.ccue.serviceExchange.adapter.sm.SmOperationExecutor] - Executing operation
'createProblem'
2014-10-08 17:28:05.657 INFO
[com.hp.ccue.serviceExchange.adapter.sm.SmOperationExecutor] - register entity:
mpavmsm10:PM10160
2014-10-08 17:28:05.977 INFO  [com.hp.ccue.serviceExchange.adapter.sm.SmAdapter] -
executing pipeline SX_MANAGED_CHANGE
2014-10-08 17:28:05.989 INFO
[com.hp.ccue.serviceExchange.adapter.sm.SmOperationExecutor] - Executing operation
'checkProblem'
2014-10-08 17:28:06.731 INFO  [com.hp.ccue.serviceExchange.oo.OoUtils] - Compressing
oo message original:555 compressed:432 ratio:77.83784
2014-10-08 17:28:06.808 INFO  [com.hp.ccue.serviceExchange.oo.OoFlowMessageListener] -
OO flow successfully executed, uri =
http://mpavmoo02.hpswlabs.adapps.hp.com:8080/oo/rest/executions/221752445/summary
2014-10-08 17:28:07.241 INFO  [com.hp.ccue.serviceExchange.rest.CatalogResource] -
Received notification ({
  "@type" : "urn:x-hp:2014:software:cloud:data_model:sx:notification",
  "displayName" : "My test ticket",
  "id" : "942aec34-80eb-47e1-918e-ac21832e485a",
  "remoteId" : "PM10160",
  "state" : "submitted",
  "submitter" : "falcon"
})
2014-10-08 17:28:07.250 INFO
[com.hp.ccue.serviceExchange.catalog.CatalogNotificationMessageListener] -
Notification to http://localhost:8080/sx/api/catalog was successful.
```

When a change occurs in HP SM, for example the problem entity is moved to the next phase, HP SX detects this change and the `checkProblem` operation is invoked again. In this case it notifies the catalog about an **in_progress** problem state.

SM change notification in sx.log

```
2014-10-08 17:30:00.217 INFO
[com.hp.ccue.serviceExchange.adapter.sm.db.SmChangeObserver] - SM change
rootcause/PM10160/sxManaged/null/update from mpavmsm10 is processing
2014-10-08 17:30:00.356 INFO  [com.hp.ccue.serviceExchange.adapter.sm.SmAdapter] -
executing pipeline SX_MANAGED_CHANGE
2014-10-08 17:30:00.363 INFO
[com.hp.ccue.serviceExchange.adapter.sm.SmOperationExecutor] - Executing operation
'checkProblem'
2014-10-08 17:30:01.304 INFO  [com.hp.ccue.serviceExchange.oo.OoUtils] - Compressing
oo message original:598 compressed:464 ratio:77.59197
2014-10-08 17:30:01.723 INFO  [com.hp.ccue.serviceExchange.oo.OoFlowMessageListener] -
OO flow successfully executed, uri =
http://mpavmoo02.hpswlabs.adapps.hp.com:8080/oo/rest/executions/221752473/summary
2014-10-08 17:30:02.285 INFO  [com.hp.ccue.serviceExchange.rest.CatalogResource] -
Received notification ({
  "@type" : "urn:x-hp:2014:software:cloud:data_model:sx:notification",
  "displayName" : "My test ticket",
  "id" : "942aec34-80eb-47e1-918e-ac21832e485a",
  "remoteId" : "PM10160",
  "state" : "in_progress",
  "submitter" : "falcon"
})
2014-10-08 17:30:02.293 INFO
[com.hp.ccue.serviceExchange.catalog.CatalogNotificationMessageListener] -
Notification to http://localhost:8080/sx/api/catalog was successful.
```

An example of a detailed communication with HP SM in an `sm-messages.log`:

Create problem in sm-messages.log

```
2014-10-08 17:28:04.261 TRACE [com.hp.ccue.serviceExchange.adapter.sm.SmAdapter] -
initializing context for message:
{@type=urn:x-hp:2014:software:cloud:data_model:sx:request, messageType=problem,
name=My test ticket, description=My test desc, urgency=4,
items=[{route={@type=urn:x-hp:2014:software:cloud:data_model:sx:route, system_type=SM,
target_instance=http://mpavmsm10.hpswlabs.adapps.hp.com:13080/SM}}],
messageHeader={messageType=problem, backendSystemType=SM, targetInstance=mpavmsm10,
externalId=942aec34-80eb-47e1-918e-ac21832e485a}, startDate=2014-10-08T15:28:03Z,
endDate=2015-10-08T15:28:03Z}
2014-10-08 17:28:04.261 INFO  [com.hp.ccue.serviceExchange.adapter.sm.SmAdapter] -
executing pipeline PLAIN
2014-10-08 17:28:04.264 INFO
[com.hp.ccue.serviceExchange.adapter.sm.SmOperationExecutor] - Executing operation
'createProblem'
2014-10-08 17:28:04.270 DEBUG
[com.hp.ccue.serviceExchange.adapter.sm.SmOperationExecutor] - executing step 'Create
problem'
2014-10-08 17:28:04.273 TRACE
[com.hp.ccue.serviceExchange.adapter.sm.SmOperationExecutor] - using integration
account: 'true'
```

```
2014-10-08 17:28:04.273 TRACE [com.hp.ccue.serviceExchange.adapter.sm.HttpClientImpl]
- Sending 'POST' request to http://mpavmsm10.hpswlabs.adapps.hp.com:13080/SM/7/ws,
payload = content-type: text/xml; charset=UTF-8, content: <Envelope
xmlns="http://schemas.xmlsoap.org/soap/envelope/">
    <Body>
        <CreateProblemRequest xmlns="http://schemas.hp.com/SM/7">
            <model>
                <keys/>
                <instance>
                    <Title>My test ticket</Title>
                    <Description>
                        <Description>My test desc</Description>
                    </Description>
                    <Impact>4</Impact>
                    <Urgency>4</Urgency>
                    <Service>Applications</Service>
                    <AssignmentGroup>Application</AssignmentGroup>
                    <Area>data</Area>
                    <Subarea>data or file incorrect</Subarea>
                </instance>
            </model>
        </CreateProblemRequest>
    </Body>
</Envelope>
.
2014-10-08 17:28:05.623 TRACE
[com.hp.ccue.serviceExchange.adapter.sm.HttpClientImpl.HttpResponse] - Response XML
<?xml version="1.0" encoding="UTF-8"?><SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
    <CreateProblemResponse xmlns="http://schemas.hp.com/SM/7"
xmlns:cmn="http://schemas.hp.com/SM/7/Common"
xmlns:xmime="http://www.w3.org/2005/05/xmlmime"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" message="Success" returnCode="0"
schemaRevisionDate="2014-10-01" schemaRevisionLevel="0" status="SUCCESS"
xsi:schemaLocation="http://schemas.hp.com/SM/7
http://mpavmsm10.HPSWLABS.ADAPPS.HP.COM:13080/SM/7/Problem.xsd">
      <model>
        <keys>
          <Id type="String">PM10160</Id>
        </keys>
        <instance recordid="PM10160 - BPPM - Open - My test ticket"
uniquequery="id=&amp;quot;PM10160&amp;quot;">
          <Id type="String">PM10160</Id>
          <Category type="String">BPPM</Category>
          <AssignmentGroup type="String">Application</AssignmentGroup>
          <Status type="String">Open</Status>
          <Title type="String">My test ticket</Title>
          <Description type="Array">
            <Description type="String">My test desc</Description>
          </Description>
          <OpenedBy type="String">falcon</OpenedBy>
          <Urgency type="String">4</Urgency>
          <Area type="String">data</Area>
          <Subarea type="String">data or file incorrect</Subarea>
          <CurrentPhase type="String">Problem Detection, Logging and
Categorization</CurrentPhase>
          <Impact type="String">4</Impact>
```

```
          <Service type="String">Applications</Service>
          <rcStatus type="String">Open</rcStatus>
        </instance>
      </model>
      <messages>
        <cmn:message type="String">Problem PM10160 has been opened.</cmn:message>
        <cmn:message type="String">Problem record added.</cmn:message>
      </messages>
    </CreateProblemResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

2014-10-08 17:28:05.640 DEBUG
[com.hp.ccue.serviceExchange.adapter.sm.SmOperationExecutor] - executing step 'Watch
interaction changes'
2014-10-08 17:28:05.657 INFO
[com.hp.ccue.serviceExchange.adapter.sm.SmOperationExecutor] - register entity:
mpavmsm10:PM10160
2014-10-08 17:28:05.666 TRACE [com.hp.ccue.serviceExchange.adapter.sm.db.SmDaoImpl] -
registerEntity rootcause/PM10160/'null'/sxManaged/null in mpavmsm10
2014-10-08 17:28:05.759 TRACE [com.hp.ccue.serviceExchange.adapter.sm.soap.SmClient] -
got REST response
http://mpavmsm10.hpswlabs.adapps.hp.com:13080/SM/9/rest/sxregisteredentities[POST]:
code = 200, body = {
  "Messages": ["SxRegisteredEntitiesV2 record added."],
  "ReturnCode": 0,
  "SXRegisteredEntities": {
    "entityId": "PM10160",
    "entityType": "rootcause",
    "id": "fa2835c9-f101-44dc-8e1b-c38096ca1d48",
    "reason": "sxManaged",
    "sxId": "FidoSM"
```

```
  }
}
```

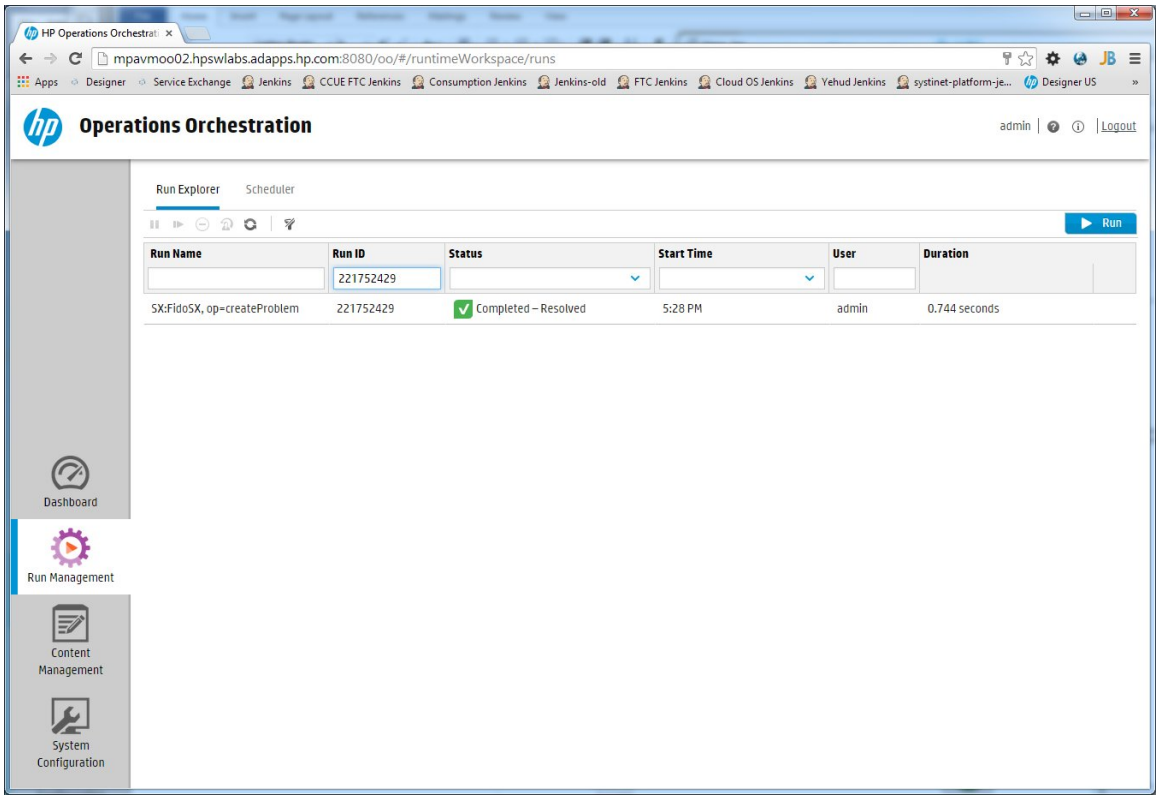
HP OO UI

Sometimes you can see that OO is executed but no further operation is called back on HP SX. In the case of a `create` operation this is an error.

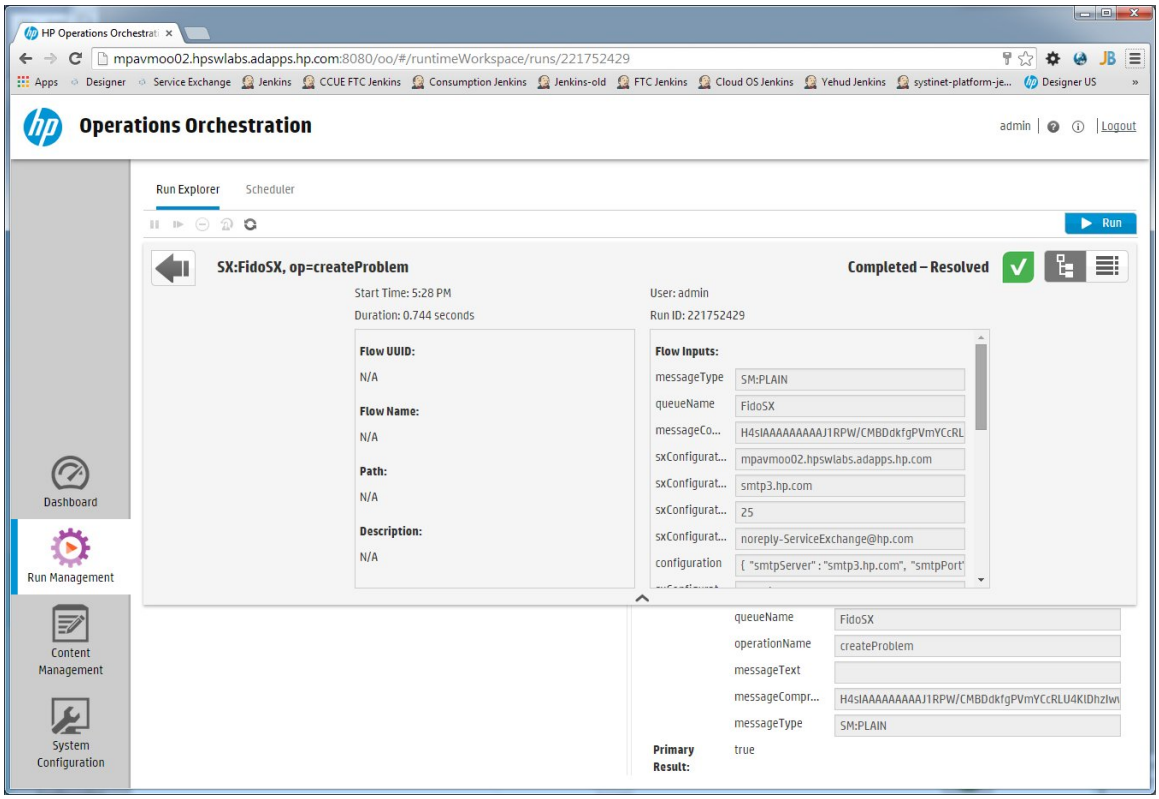
To identify what went wrong go to the OO Central UI and look at the specific flow executed there. OO is typically running on https://sx_host:8443/oo. The default username is *admin* and the password *changeit*. In the left column click **Run Management** and choose the row corresponding to your execution. It could be identified by name, or the Run ID displayed in the HP SX logs. The earlier log example contains following:

```
2014-10-08 17:28:04.097 INFO  [com.hp.ccue.serviceExchange.oo.OoFlowMessageListener] -
OO flow successfully executed, uri =
http://mpavmoo02.hpswlabs.adapps.hp.com:8080/oo/rest/executions/221752429/summary
```

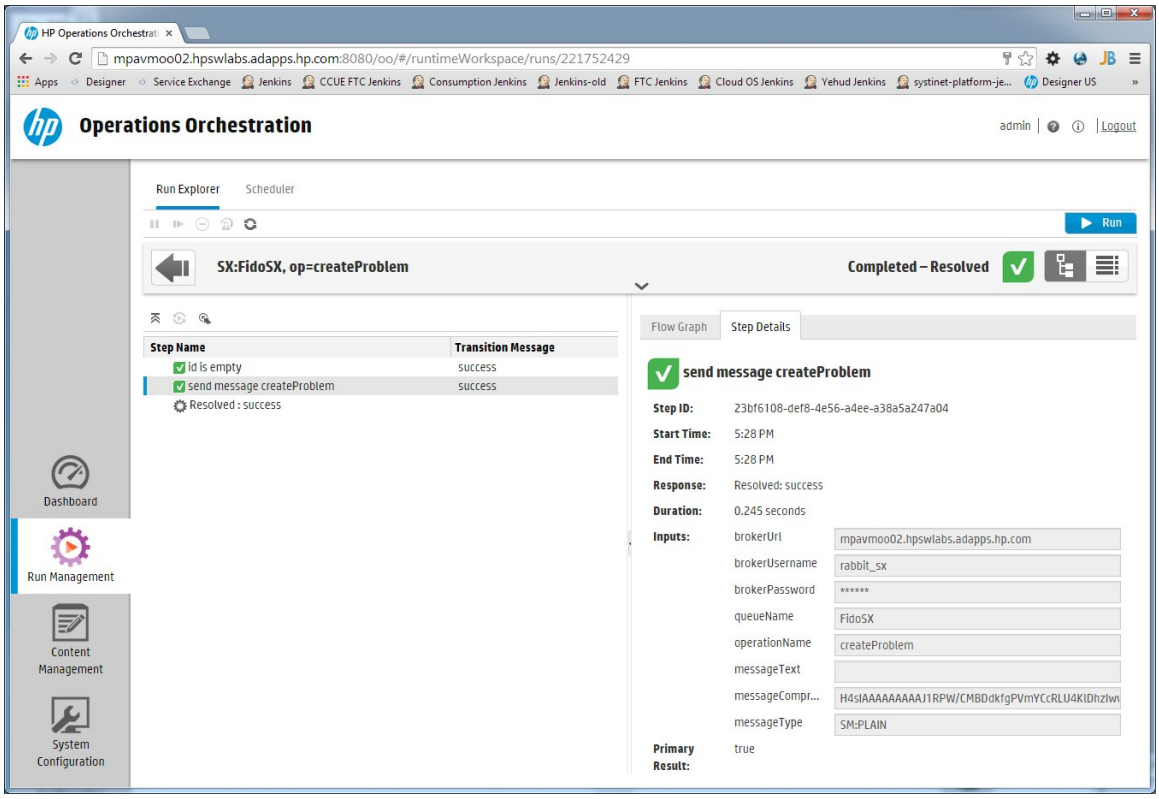
The Run ID is the number close to the end of the URL. For example, in the example above it is 221752429. Use it for filtering on the run list, or simply paste the ID into the **Run ID** column filter field.



To check the input flow parameters, click on a run row. Display it by clicking on the down pointing arrow in the middle of the top bar (containing the execution name and status.)



Inspect individual step inputs and outputs within the flow execution by selecting a step in the left column and switching to **Step Details** in the right column, as in the example screenshot below:



How to develop an adapter (JIRA)

Developing an adapter explained with a JIRA adapter example

When to use this guide

This chapter explains how to integrate HP SX with a new backend system that is not yet supported by HP SX. If you wish to customize the behavior of an already supported system, for example HP SM or HP CSA, it may be enough to only create a new content pack for the existing adapter. See How to extend HP SX Content (HP SM Problem entity) for creating a custom content pack.

This section describes the implementation of three HP SX functions: ticketing, request-to-fulfilment (R2F), and Case Exchange. The procedures start from the simplest to the more complex, explained with the example of JIRA.

Throughout this example it is presumed that a maven project is being built, and that a root maven module is set up as described in How to extend HP SX Content (HP SM Problem entity). The modules are created under this root project. The project is included in the SDK pack under `sx-content/jira`.

For the java part of the adapter, a maven module is created according to this example:

Adapter maven module

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>your.parent.group.id</groupId>
        <artifactId>your-parent-artifact-id</artifactId>
        <version>your_version</version>
    </parent>
 <artifactId>sx-adapter-jira</artifactId>
    <packaging>jar</packaging>
    <name>JIRA Adapter</name>
    <dependencies>
        <dependency>
         <groupId>com.hp.ccue.serviceExchange</groupId>
   <artifactId>sx-api</artifactId>
        </dependency>
        <dependency>
          <groupId>com.hp.ccue.serviceExchange</groupId>
       <artifactId>sx-adapter-api</artifactId>
        </dependency>
   <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
        </dependency>
 </dependencies>
</project>
```

The dependencies that you need to have acces to are:

- sx-api
- sx-adapter-api
- spring-context

Ticketing usecase

JIRA adapter example implementation - Ticketing usecase

Case exchange usecase

Case exchange use case

Request-to-fulfill usecase

Request to fulfill use case

Java adapter deployment

The methods to deploy the content part are described in How to extend HP SX Content (HP SM Problem entity). For development, the usage of `sx-maven-plugin` is advisable.

The java part of the adapter is a jar file that must be deployed into the HP SX application server. Do this before you upload your content pack as there is a validation that will not let you upload a content pack for a non existing adapter.

To deploy the adapter jar:

1. Stop your SX instance jboss e.g. `service jboss-7 stop`
2. Copy the adapter jar file into `<YOUR_JBOSS_HOME>/standalone/deployments/sx.war/WEB-INF/lib`
3. Create an `instances.json` configuration file in a location that reflects adapter implementation.
4. Restart your SX instance jboss e. g. `service jboss-7 start`

Ticketing use case

Ticketing is the simplest HP SX use case. In order to adapt a backend system to the ticketing use case it is necessary to:

- Setup an adapter class.
- Setup an operation executor class.
- Define a set of operations that are necessary to perform ticket management through the Propel Portal.

At the end of the implementation of this use case you can manage JIRA issues as tickets in the Propel Portal.

It is necessary that you have access to the example implementation sources contained in the HP SX SDK distribution. You will need to

refer to the source code to be able to fully understand the contents of this topic. Especially refer to the ftl templates.

It is also advisable to refer to HP SX API javadoc, also contained in the SDK package under the javadoc folder.

Finally, you may also want to consult the How to extend HP SX Content (HP SM Problem entity) topic which has details about many development techniques, for example:

- uploading SX content packs using the SX content management UI and the SX upload maven plugin
- analysis of SX log files.

Ticket management message flow in HP SX

1. The incoming ticket management request is submitted through SX RESTful interface (/ticket REST resource)
2. The incoming message is decorated and updated according to the specific call
3. The incoming rest call is mapped to the appropriate operation from `operations.json`
4. The correct adapter is chosen from among the registered adapters
5. The operation is executed using the adapter's operation executor.

The ticket management requests are documented in the SX API doc. Use the Appendix C: Ticket management operations messages as a reference on the format of messages to be passed to and returned from an operation execution.

The set of operations that the API calls are mapped to and that need to be defined in `operations.json` are as follows:

Operation	Note
createTicket	Creates a new ticket
retrieveTicket	Retrieves a ticket
listTickets	Lists tickets matching the given criteria
listTicketProperties	Returns enumeration of properties that are wanted visible to portal users
ticketProperty-${propertyName}	This operation must be implemented for each property returned by listTicketProperties
listTicketAttachments	Lists ticket attachments
createTicketAttachment	Creates a ticket attachment
retrieveTicketAttachment	Retrieves ticket attachment metadata
createTicketComment	Creates a comment
closeTicket	Closes a ticket

NOTE: This set of operations is configured in the `sx-base` content pack in the `operationMappings.json` file. It is possible to change this mapping by customizing the `sx-base` content pack. See Content packs, in particular the content pack update details. Be aware that customizations made will affect all adapters in your HP SX instance.

Adapter class

The provided abstract class `com.hp.ccue.serviceExchange.adapter.provided.AdapterAbstract` is used.

JIRA adapter class

```
@Component
public class JiraAdapter extends AdapterAbstract {
    @Autowired
    public JiraAdapter(JiraOperationExecutor operationExecutor, JiraPipelineBuilder
pipelineBuilder) {
        super(JiraConstants.JIRA_ADAPTER_NAME, operationExecutor, pipelineBuilder);

    }
}
```

The `@org.springframework.stereotype.Component` annotation is used in order to have it spring-enabled. `JiraConstants.JIRA_ADAPTER_NAME` name constant is introduced in a separate class. Make sure it is unique among adapter name constants.

Define the Spring framework *component scanning* in the context definition with a name following the `*Context.xml` pattern, for example:

Spring context definition file

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:context="http://www.springframework.org/schema/context"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="com.example.adapter.package.jira"/>

</beans>
```

OperationExecutor and Pipelinebuilder

The adapter constructor accepts operation executor and pipeline builder. Minimal implementations are sufficient. In both cases you can extend the provided implementation, see the example code below:

Operation Executor

Operation executor

```
@Component
public class JiraOperationExecutor extends BaseOperationExecutor {

 public JiraOperationExecutor() {
        super(JiraConstants.JIRA_ADAPTER_NAME, JiraInstancesCfg.CFG_NAME);
        setDefaultHttpRequestContentType(MediaType.APPLICATION_JSON);
    }
}
```

Here a default content type `application/json` is specified. This means that all requests defined in `operations.json` will be issued with this

content type unless another content type is specified..

It is useful to override the getDetailErrorMessage()method, see the source code for the actual implementation.

PipelineBuilder

PipelineBuilder

```
 @Component
public class JiraPipelineBuilder implements AdapterPipelineBuilder {

    @Override
     public Pipeline buildPipeline(Adapter adapter, PipelineBuilderFactory factory,
String name) {
      return null;
 }
}
```

This is the minimal adapter implementation sufficient for an HP SX ticketing use case - the Java implementation of the adapter itself is now complete.

Ticketing content pack

The above implementation of the adapter has only defined a new backend system of type JIRA, and an adapter that will process messages for this type of system with a default HP SX implementation. Now the set of ticketing operations must be defined. The operation definitions are deployed in an HP SX content pack.

For the content pack, define a new maven module in your project, for example:

Content maven module

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <groupId>your.parent.group.id</groupId>
        <artifactId>your-parent-artifact-id</artifactId>
        <version>your_version</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>content-jira-ticketing</artifactId>
</project>
```

In this module create a structure as described in Content packs. This content pack will only contain operation definitions and ftl templates, so omit the oo directory.

```
`-- src
     `-- main
        `-- resources
           |-- sx
           |   |-- templates
           |   `-- operations.json
           `-- metadata.json
```

Example operation implementation - createTicket

The operations definition is explained here using the example of the `createTicket` operation.

NOTE: You can test the operations by performing corresponding actions in the Propel UI. Alternatively, you could test the REST calls in a browser client like Postman or DHC, but the APIs require a valid IdM Token to be passed in the X-Auth-Token header. You need to perform a REST call to the IdmServer to get such a token (using basic HTTP authentication with user idmTransportUser and password idmTransportUser), like this one:

```
POST /idm-service/v2.0/tokens
Authorization: Basic aWRtVHJhbnNwb3J0VXNlcjppZG1UcmFuc3BvcnRVc2Vy
Content-Type: application/json
{
  "passwordCredentials" : {
    "username" : "consumer",
    "password" : "cloud"
  },
  "tenantName" : "CONSUMER"
}
```

According to JIRA rest API documentation you need to issue the following request to create an issue in JIRA:

Method	URI	Request media type	Response media type
POST	`(/[context]?)/rest/api/latest/issue`	application/json	application/json

```
{    "fields": {
       "project":
       {
          "key": "TEST"
       },
       "summary": "REST ye merry gentlemen.",
       "description": "Creating of an issue using project keys and issue type names
using the REST API",
       "issuetype": {
          "name": "Bug"
       }
   }
}
```

Example response:

JIRA create issue response

```
{   "id":"39000",
    "key":"TEST-101",
    "self":"http://localhost:8090/rest/api/2/issue/39000"
}
```

The `createTicket` operation is defined in the `operations.json` file of the content pack.

operations.json

```
{
    "createTicket": [
        {
            "label": "Create Ticket",
            "requestUrlTemplate": "createTicketUrl.ftl",
            "requestTemplate": "createTicketRequest.ftl",
            "responseTemplate": "createTicketResponse.ftl",
            "method": "POST"
        }
    ]
}
```

The above is an example of an operation that has a single step labeled "Create Ticket". In this step it issues a POST http request to a URL returned by requestUrlTemplate with a body returned by requestTemplate, and the response of the POST http request will be transformed by the responseTemplate which will be the result of the operation.

The implementations of the ftl templates that are used in this operation configuration are presented below.

The operation definition is interpreted by the operation executor. The operation executor's input is a generic HP SX message to create a ticket. This message together with some context information are the input to the requestUrlTemplate and requestTemplate ftl, whereas the response template will receive the http request response. For a detailed introduction to the HP SX operation executor framework see Appendix B: Operation executors.

A general description of the messages passed to the HP SX operation executor is documented in Ticket management operations messages doc. The example message that will be passed to the operation executor when creating a ticket is as follows:

Example create ticket message

```
{
  "description" : "JIRA Test ticket 2 desc",
  "name" : "JIRA Test ticket 2",
  "properties" : [ {
    "@type" : "urn:x-hp:2014:software:cloud:data_model:property:text",
    "name" : "name",
    "value" : "JIRA Test ticket 2"
  }, {
    "@type" : "urn:x-hp:2014:software:cloud:data_model:property:text",
    "name" : "description",
    "value" : "JIRA Test ticket 2 desc"
  }, {
    "@type" : "urn:x-hp:2014:software:cloud:data_model:property:select",
    "name" : "project",
    "value" : "DES"
  }, {
    "@type" : "urn:x-hp:2014:software:cloud:data_model:property:select",
    "name" : "issuetype",
    "value" : "1"
  }, {
    "@type" : "urn:x-hp:2014:software:cloud:data_model:property:select",
    "name" : "priority",
    "value" : "1"
  }, {
    "@type" : "urn:x-hp:2014:software:cloud:data_model:property:select_from_many",
    "name" : "reporter",
    "value" : "consumer"
  }, {
    "@type" : "urn:x-hp:2014:software:cloud:data_model:property:select_from_many",
    "name" : "assignee",
    "value" : ""
  } ],
  "messageHeader" : {
    "backendSystemType" : "JIRA",
    "userId" : "consumer",
    "targetInstance" : "mpavmint01"
  },
  "recipient" : {
    "name" : "consumer"
  }
}
```

requestUrl

The template composes the rest call URL. It does not need any data from an incoming message. Note: The ftl transformation data model contains the `instanceConfig` object. For a full context reference see Appendix B: Operation executors.

createTicketUrl.ftl

```
<#escape x as x?url>
<#noescape>${instanceConfig.endpoint}</#noescape>
/rest/api/latest/issue
</#escape>
```

requestTemplate

This transforms the message into a valid POST request body of the JIRA REST endpoint.

NOTE: The message content is accessed through the `message` ftl data model node. See line 2 for an example.

createTicketRequest.ftl

```
<#assign valueMap = {}/>
<#list message.properties as property>
    <#assign valueMap = valueMap + {property.name : property.value}/>
</#list>
<#escape x as x?json_string>
{
    "fields": {
        "summary": "${message.name}",
        <#if message.description?has_content>
        "description": "${message.description}",
        </#if>
        "project": {
            "key": "${valueMap.project}"
        },
        "issuetype": {
            "id": "${valueMap.issuetype}"
        },
        "priority": {
            "id": "${valueMap.priority}"
        },
        "reporter": {
            "name": "${valueMap.reporter}"
        }
        <#if valueMap.assigned_to?has_content>
        ,
        "assignee": {
            "real_name": "${valueMap.assigned_to}"
        }
        </#if>
    }
}
</#escape>
```

responseTemplate

This adapts the JIRA response format to the HP SX format. Use Appendix C: Ticket management operations messages as a reference.

The example response of the HP SX `createTicket` operation:

createTicket response

```
{
"result":
 {
   "_links" : {
     "self" : {
       "href" : "/sx/api/ticket/DES-2313"
     }
   },
   "id" : "DES-2313",
   "name" : "JIRA Test ticket 2",
   "description" : "JIRA Test ticket 2 desc",
   "openTime" : "2014-10-14T13:36:17+02:00",
   "updateTime" : "2014-10-14T13:36:17+02:00",
   "status" : "submitted",
   "properties" : [ {
     "@type" : "urn:x-hp:2014:software:cloud:data_model:property:select",
     "name" : "project",
     "value" : "DES"
    }, {
     "@type" : "urn:x-hp:2014:software:cloud:data_model:property:select",
     "name" : "issuetype",
     "value" : "1"
    }, {
     "@type" : "urn:x-hp:2014:software:cloud:data_model:property:select",
     "name" : "priority",
     "value" : "1"
     }, {
     "@type" : "urn:x-hp:2014:software:cloud:data_model:property:select_from_many",
     "name" : "reporter",
     "value" : "consumer"
  } ],
  "comments" : [ ]
 }
}
```

It is not possible to construct this response as the JIRA response only contains ID, key and self. It is necessary to add a second step that retrieves the created JIRA issue.

In the `operations.json` file:

operations.json

```
{
    "createTicket": [
        {
            "label": "Create Ticket",
            "requestUrlTemplate": "createTicketUrl.ftl",
            "requestTemplate": "createTicketRequest.ftl",
            "responseTemplate": "createTicketResponse.ftl",
            "method": "POST"
        },
  {
            "label": "Retrieve Ticket",
            "requestUrlTemplate": "getTicketUrl.ftl",
            "responseTemplate": "getTicketResponse.ftl"
        }
 ]
}
```

Method specification is not needed as the GET method is the default when the `requestTemplate` is not specified.

The first step's `responseTemplate` writes down the ID of the created issue into the message body.

NOTE: The responseTemplate gets exactly the same input as requestTemplate except that it is nested under the "doc" key in the incoming data model. The response body is under "result".

createTicketResponse.ftl

```
<#escape x as x?json_string>
{
    "id": "${doc.result.key}"
}
</#escape>
```

The next step is using the ID to retrieve the issue:

getTicketUrl.ftl

```
<#escape x as x?url>
<#noescape>${instanceConfig.endpoint}</#noescape>
/rest/api/latest/issue/${message.id}?fields=summary,description,created,updated,status,pr
```

Finally, transform the JIRA response into the HP SX format:

getTicketResponse.ftl

```
<#include "jiraTicketingUtils.ftl"/>
```

```
<#escape x as x?json_string>

<#assign result = doc.result/>
<#assign fields = result.fields/>
{
    "result": {
        "_links": {
            "self": {
                "href": "/sx/api/ticket/${result.key?url}"
            }
        },
        "id": "${result.key}",
        "name": "${fields.summary}",
        "description": "${fields.description!""}",
        "openTime": "${toSxDate(fields.created)}",
        "updateTime": "${toSxDate(fields.updated)}",
        "status": "${toSxStatus(fields.status.statusCategory.key)}",
        "properties": [
            {
                "@type": "urn:x-hp:2014:software:cloud:data_model:property:select",
                "name": "project",
                "value": "${fields.project.key}"
            },
            {
                "@type": "urn:x-hp:2014:software:cloud:data_model:property:select",
                "name": "issuetype",
                "value": "${fields.issuetype.id}"
            },
            {
                "@type": "urn:x-hp:2014:software:cloud:data_model:property:select",
                "name": "priority",
                "value": "${fields.priority.id}"
            },
            {
                "@type":
"urn:x-hp:2014:software:cloud:data_model:property:select_from_many",
                "name": "reporter",
                "value": "${fields.reporter.name}"
            }
            <#if fields.assignee??>
            ,
            {
                "@type":
"urn:x-hp:2014:software:cloud:data_model:property:select_from_many",
                "name": "assignee",
                "value": "${fields.assignee.name}"
            }
            </#if>
        ],
        "comments": [
            <#list fields.comment.comments as comment>
            {
                "id": "${comment.id}",
                "author": "${comment.author.name}",
                "time": "${toSxDate(comment.created)}",
                "description": "${comment.body}"
            }<#if comment_has_next>,</#if>
            </#list>
```

```
        ]
    }
}

</#escape>
```

`jiraTicketingUtils.ftl` is included. It contains the utility functions `toSxDate and toSxStatus`.

NOTE: The `createTicket` operation in the example project is more complicated than what is shown here, as it also contains permission checks for the current user. The permission checks are necessary as all REST calls are performed under the integration user, not under the current user.

Ticket properties

The JIRA issue attributes: those properties that need to be visible in the Propel portal, are defined by the `listTicketProperty` operation. This operation simply enumerates the properties. See the following example implementation:

operations.json

```
...
,
    "listTicketProperties": [
        {
            "label": "List Ticket Properties",
            "resultTemplate": "listTicketPropertiesResponse.ftl"
        }
    ],
...
```

listTicketPropertiesResponse.ftl

```
<#escape x as x?json_string>

{
    "result": {
        "@self": "/sx/api/ticket/property",
        "properties": [
            {
                "@type": "urn:x-hp:2014:software:cloud:data_model:property:text",
                "name": "name",
                "displayName": "Title",
                "length": 255,
                "required": false,
                "requiredOutsidePropertiesList": true
            },
            {
                "@type": "urn:x-hp:2014:software:cloud:data_model:property:text",
                "name": "description",
                "displayName": "Description",
                "length": 200000,
                "required": false,
                "requiredOutsidePropertiesList": false
            },
```

```
            {
                "@type": "urn:x-hp:2014:software:cloud:data_model:property:select",
                "name": "project",
                "displayName": "Project",
                "valuesUrl": "/sx/api/ticket/property/project",
                "required": true
            },
            {
                "@type": "urn:x-hp:2014:software:cloud:data_model:property:select",
                "name": "issuetype",
                "displayName": "Issue Type",
                "valuesUrl": "/sx/api/ticket/property/issuetype",
                "required": true
            },
            {
                "@type": "urn:x-hp:2014:software:cloud:data_model:property:select",
                "name": "priority",
                "displayName": "Priority",
                "valuesUrl": "/sx/api/ticket/property/priority",
                "required": true
            },
            {
                "@type":
"urn:x-hp:2014:software:cloud:data_model:property:select_from_many",
                "name": "reporter",
                "displayName": "Reporter",
                "searchUrl": "/sx/api/ticket/property/user",
                "required": true,
                "default": "${message.messageHeader.userId}"
            },
            {
                "@type":
"urn:x-hp:2014:software:cloud:data_model:property:select_from_many",
                "name": "assignee",
                "displayName": "Assignee",
                "searchUrl": "/sx/api/ticket/property/user",
                "required": false,
                "default": ""
            }
        ]
    }
```

```
}

</#escape>
```

All possible property types can be listed using an SX API. See the Ticket property descriptors in the SX API doc.

Each property returned by `listTicketProperties` that is of the type `select` or `select_from_many` needs a corresponding operation to list the possible values, for example, project property.

opertations.json

```
...
,
    "ticketProperty-project": [
        {
            "label": "Get Project Property Values",
            "requestUrlTemplate": "listProjectsUrl.ftl",
            "responseTemplate": "listProjectsResponse.ftl"
        }
    ],
...
```

listProjectsUrl.ftl

```
<#escape x as x?url>
<#noescape>${instanceConfig.endpoint}</#noescape>
/rest/api/latest/project
</#escape>
```

listProjectsResponse.ftl

```
<#escape x as x?json_string>
{
    "result": {
        "_links": {
            "self": {
                "href": "/sx/api/ticket/property/project"
            }
        },
        "values": [
            <#list doc.result?sort_by("name") as item>
            {
                "value": "${item.key}",
                "label": "${item.name}"
            }<#if item_has_next>,</#if>
            </#list>
        ]
    }
}
</#escape>
```

Summary

The rest of the operations are implemented analogically. Use the Appendix C: Ticket management operations messages as a reference on the input and output messages of individual operations.

Case exchange use case

This section describes how to implement a CX-capable adapter, using the example of JIRA as the ticketing system. This procedure extends the implementation described in Ticketing use case.

It is necessary that you have access to the example implementation sources contained in the SX distribution. You will need to refer to the source code to be able to fully understand the contents of this topic. Especially refer to the ftl templates.

It is also advisable to refer to the SX API javadoc.

Finally, you may also want to consult the How to extend HP SX Content (HP SM Problem entity) topic which has details about many development techniques, for example:

- uploading SX content packs using the SX content management UI and the SX upload maven plugin
- analysis of SX log files.

Use case definition

The goal of this example is to enable the following functionality.

1. HP SM incident is delegated to JIRA by setting incident's properties as
    - Status = Pending vendor
    - Vendor = <*your_jira_instance_alias*>

    In this way, a new linked issue is created in your JIRA instance.
2. When the JIRA linked issue is resolved, the original HP SM incident is also automatically resolved.

Overview

The core actors that initiate actions in a CX use case are the change observers. The change observers periodically poll an external system for changes and filter these changes to trigger configured events. Once an event is detected a message is generated and sent to HP SX for processing. Appropriate actions are performed based on the configuration.

The implementation of our use case consists of two tasks:

1. configuration
2. implement the JIRA adapter CX support

Implementation

Configuration

To configure incident CX between JIRA and an HP SM instance it is necessary to register `IncidentCaseExchangeEvents` on the particular instances and to pair these instances in both directions.

So in our example we need to have support for this example configuration:

external-systems.json

```
"externalSystems": [
        {
            "instanceType": "SM",
            "instance": <your_HP_SM_instance>,
            "registeredEventGroups": [
                "IncidentCaseExchangeEvents"
            ]
        },
        {
            "instanceType": "JIRA",
            "instance": <your_JIRA_instance>,
            "registeredEventGroups": [
                "IncidentCaseExchangeEvents"
            ]
        }
    ]
"externalSystemAliases": [
{
            "sourceInstanceType": "SM",
            "sourceInstance": <your_HP_SM_instance>,
            "targetInstanceType": "JIRA",
            "targetInstance": <your_JIRA_instance>,
            "targetAlias": <your_JIRA_instance_alias>
        },
{
            "sourceInstanceType": "JIRA",
            "sourceInstance": <your_JIRA_instance>,
            "targetInstanceType": "SM",
            "targetInstance": <your_HP_SM_instance>,
            "targetAlias": <your_HP_SM_instance_alias>
        }
]
```

See Configuration for a complete reference of CX configuration.

The HP SM adapter is available OOB and supports Incident CX. The OOB content contains `IncidentCaseExchangeEvents` and `eventGroup Actions` definition.

content-case-exchange/case-exchange.json

```
{
  "eventGroups": {
    "IncidentCaseExchangeEvents": [
        "incidentExternalReferenceCreated",
        "incidentUpdated",
        "incidentResolved",
        "incidentReopened",
        "incidentClosed",
        "incidentOwnershipAssigned",
        "incidentOwnershipAccepted",
        "incidentRejected",
        "incidentCancelled"
    ]
...
  },
"eventGroupActions": {
    "IncidentCaseExchangeEvents": [
      {
        "action": "executeOperation",
        "operationName": "retrieveIncident"
      },
      {
        "action": "executeOperation",
        "operationName": "convertIncidentToCanonicalModel"
      },
      {
        "action": "executeOoFlow",
        "backendSystemType": "SX",
        "messageType": "IncidentCaseExchangeFlow"
      }
    ],
...
```

This configuration contains all the operations we need to support both HP SM and JIRA.

For HP SM the operations are available OOB.

For JIRA it is clear that it is necessary to implement the following

- retrieveIncident
- convertIncidentToCanonicalModel

The rest of the operations are derived from the `IncidentCaseExchangeFlow` OO flow. With deeper inspection of the flow you may note that for this use case JIRA will only need:

- cloneIncident

See OO flows, for the `IncidentCaseExchangeFlow` description.

In addition to the operations, the events need to be defined. The events definition for HP SM is available OOB. See `content-sm-case-exchange/case-exchage.json`. The event `incidentExternalReferenceCreated` is the crucial starting point for this use case.

For JIRA we must define the event that will trigger the action in point 2 of the use case. Create a `resources/case-exchange.json` file in the JIRA content module:

content-jira/case-exchange.json

```
{
    "events": {
        "JIRA": {
            "incidentResolved": {
                "entityType": "Incident",
                "entityFilter": "Boolean(RECORD.status) && RECORD.status.to == '5'" //
a javascript condition matching updates where the issue status was set to Resolved
            }
        }
    }
}
```

`incidentResolved` is an event belonging to `IncidentCaseExchangeEvents` that is registered on the JIRA instance. As noted, these events are mapped to the specific set of actions in `content-case-exchange/case-exchange.json` OOB. The `entityType` and `entityFilter` fi elds are adapter-specific and are dependent on the implementation of change observing. See below for further information.

JIRA adapter support

To implement adapter support for CX it is necessary to add the following capabilities to the adapter:

- the adapter must be able to handle a message that will execute the creation of a linked incident in JIRA (i.e. plain pipeline)
- implement change observing
- the adapter must handle the message generated by the change observer (i.e. case_exchange_pipeline.)

Plain Pipeline

In summary, the process that enables the use case: the creation of new linked issues in a JIRA instance, is:

1. HP SM adapter's change observer (implemented OOB) detects an `incidentExternalReferenceCreated` event in HP SM.
2. The event results in a CX type message being sent.
3. The message is handled by an adapter (the HP SM adapter in this case) in a CX pipeline.
4. The CX pipeline contains a CX handler block that is responsible for the execution of eventGroupActions defined in `content-case-exc hange/case-exchange.json`
5. The last action in the defined `eventGroupAction` is an `IncidentCaseExchangeFlow` execution which, based on the message, decides that it should call a clone incident in JIRA and so sends a message to execute the `cloneIncident` operation.
6. This message is sent to the PLAIN pipeline.
7. At this point the JIRA adapter comes into play. Support for a PLAIN pipeline needs to be added for the JIRA adapter as it does not know how to build a PLAIN pipeline yet.

To extend the empty `PipelineBuilder` implementation introduced as part of the ticketing use case:

```
@Component
public class JiraPipelineBuilder implements AdapterPipelineBuilder {


  @Override
    public Pipeline buildPipeline(Adapter adapter, PipelineBuilderFactory factory,
String name) {
        switch (name) {
            case Names.PIPELINE_PLAIN:
                return buildPlainPipeline(factory);
            default:
                return null;
        }
    }

   public Pipeline buildPlainPipeline(PipelineBuilderFactory factory) {
        PipelineBuilder builder = factory.newBuilder(Names.PIPELINE_PLAIN);
        builder.addBlock(new OperationExecutionBlock());
        return builder.build();
    }
}
```

`OperationExecutionBlock` is now added into the PLAIN pipeline, making the pipeline capable of executing operations received in messages.

Now it is enough to implement the JIRA `cloneIncident` operation and the first part of the use case (HP SM to JIRA incident delegation) is implemented:

operations.json

```
...
    "cloneIncident": [
        {
            "label": "Clone incident",
            "requestUrlTemplate": "createTicketUrl.ftl",
            "requestTemplate": "cloneIncidentRequest.ftl",
            "responseTemplate": "cloneIncidentResponse.ftl",
            "method": "POST"
        },
        {
            "label": "Get ticket after clone incident",
            "requestUrlTemplate": "getTicketAfterCloneIncidentUrl.ftl",
            "responseTemplate": "getTicketAfterCloneIncidentResponse.ftl",
            "method": "GET"
        }
    ],
...
```

See the ftl templates in the example sources for details. For your convenience, here are the `cloneIncidentRequest.ftl` and `getTicketAfterCloneIncidentResponse.ftl` templates:

cloneIncidentRequest.ftl

```
<#include "jiraCaseExchangeUtils.ftl"/>
<#assign loadConfig =
'com.hp.ccue.serviceExchange.adapter.freemarker.LoadConfig'?new()/>
<#assign findAliasForExternalSystem =
'com.hp.ccue.serviceExchange.adapter.freemarker.caseex.FindAliasForExternalSystem'?new()
/>
<#assign jiraMapping = loadConfig(context.contentStorage, "jira/jira-cx-mappings")/>
<#assign entity = message.args.entity/>
<#assign linkedEntity = message.args.linkedEntity/>
<#assign properties = entity.properties/>
<#escape x as x?json_string>
<#assign environmentValue>
[{
    "externalEntityType": "${entity.entityType}",
    "externalEntityId": "${entity.entityId}",
    "externalInstanceAlias": "${findAliasForExternalSystem(context.appContext,
        linkedEntity.instanceType, linkedEntity.instance, entity.instanceType,
entity.instance)}"
}]
</#assign>
{
    "fields": {
        "summary": "${properties.Title}",
        <#if properties.Description?has_content>
        "description": "${properties.Description}",
        </#if>
        "project": {
            "key": "SE" <#-- hard-coded for now; use a project key in your JIRA
instead -->
        },
        "issuetype": {
            "id": "1" <#-- Bug -->
        },
        "priority": {
            "id": "${getMappingValue(jiraMapping.Incident.Urgency,
properties.Urgency)}"
        },
        "reporter": {
            "name": "System.admin" <#-- hard-coded for now; use a username in your
JIRA instead -->
        },
        "environment": "${environmentValue}" <#-- better solution: use custom fields;
we store the externalEntity in environment field for demo purposes only -->
    }
}
</#escape>
```

getTicketAfterCloneIncidentResponse.ftl

```
<#include "jiraCaseExchangeUtils.ftl"/>
<#assign
loadConfig='com.hp.ccue.serviceExchange.adapter.freemarker.LoadConfig'?new()/>
<#assign jiraMapping = loadConfig(context.contentStorage, "jira/jira-cx-mappings")/>
<#escape x as x?json_string>
{
    "args": {
        "linkedEntity": {
            "entityType": "Incident",
            "entityId": "${doc.result.key}",
            "properties": {
                "Status": "${getMappingKey(jiraMapping.Incident.Status,
doc.result.fields.status.id)}"
            }
        }
    },
    "tmp": null
}
</#escape>
```

*NOTE: HP SX is storing a reference to the HP SM entity in the **environment** property (as a JSON string). Storing the reference is actually not needed in the first use case, but it will be important for the second.*

The first use case is now implemented and can be tested like this:

- Choose an existing SM incident or create a new one.
- Set the status field to "Pending Vendor" and the Vendor field to "jira" (or whatever targetAlias for JIRA is in the `external-systems.js` on file).
- Click **Save** and **Exit**.

  You should now observe that:
  - A JIRA issue has been created with the same title (i.e. summary) and description as the HP SM incident. Also, the issue should have the corresponding priority.
  - If you open the HP SM incident again, it should have the Vendor Ticket property updated to the key of the new issue.

JiraAdapter class

The adapter needs to be capable of change observation. To modify the adapter class:

JiraAdapter.java

```
@Component
public class JiraAdapter extends AdapterAbstract {
    @Autowired
    public JiraAdapter(JiraOperationExecutor operationExecutor, JiraPipelineBuilder
pipelineBuilder, JiraChangeObserver changeObserver) {
        super(JiraConstants.JIRA_ADAPTER_NAME, operationExecutor, pipelineBuilder);

  // setting change observer
        setChangeObserver(changeObserver);
    }
...
```

Autowire the `JiraChangeObserver` in the constructor and set it. The `JiraChangeObserver` implementation is described below.

JiraChangeObserver class

For `JiraChangeObserver` we use SDK's base class `CompositeChangeObserver`. This base class is used as it provides the possibility to use multiple polling commands which will be useful in an R2F use case.

JiraChangeObserver.java

```
@Component
public class JiraChangeObserver extends CompositeChangeObserver {
    @Value("${adapter.jira.change.listener.delayBeforeNextRun}")
    private int pollInterval;
    @Autowired
    public JiraChangeObserver(JiraCxPollingCommand cxPollingCommand) {
        super(ImmutableList.<Runnable>of(cxPollingCommand));
    }
    @Override
    public int getPollIntervalSec() {
        return pollInterval;
    }
}
```

JiraCxPollingCommand

See the following example implementation that already contains the implementation of abstract ancestor methods. `CxPollingCommand` is an implementation that uses an operation of the supplied name to list changes in an external system. It then uses the `entityFilter` property of event definitions to assign the changes to configured events, and sends an internal CX type message. The abstract methods provide customization points to change retrieval and message generation processes.

NOTE: `JiraCaseExchangeRuleStore` and `JiraEventFilterEvaluator` are autowired in the constructor.

JiraCxPollingCommand.java

```
@Component
public class JiraCxPollingCommand extends CxPollingCommand {
    private static final String KEY_ENTITY_ID = "entityId";
    private static final String KEY_EXTERNAL_ENTITIES = "externalEntities";
    private static final String KEY_HISTORIES = "histories";
    public static final String OPERATION_GET_CHANGES = "getChangedIncidentsForCx";
    @Autowired
    public JiraCxPollingCommand(JiraOperationExecutor operationExecutor,
JiraCaseExchangeRuleStore caseExchangeRuleStore,
                                JiraEventFilterEvaluator filterEvaluator) {
        super(JiraConstants.JIRA_TYPE, JiraConstants.ENTITY_INCIDENT,
JiraConstants.JiraInstancesCfg.CFG_NAME,
                OPERATION_GET_CHANGES,
                operationExecutor, caseExchangeRuleStore, filterEvaluator);
    }
    @Override
    @Nonnull
    protected List<Map<String, Object>> extractChangedEntities(Map<String, Object>
changedEntities) {
        List<Map<String, Object>> result = getField(changedEntities, KEY_HISTORIES);
        return Objects.firstNonNull(result, Collections.<Map<String,
Object>>emptyList());
    }
    @Nonnull
    @Override
    protected String extractEntityId(Map<String, Object> entity) {
        return getStrField(entity, KEY_ENTITY_ID);
    }
    @Override
    protected Map<String, Object> prepareMessageCustomDataForCx(Map<String, Object>
entity, final Set<String> externalInstanceAliases) {
        Map<String, Map<String, Object>> externalEntities = getField(entity,
KEY_EXTERNAL_ENTITIES);
        Map<String, Map<String, Object>> relevantExternalEntities =
Maps.filterKeys(externalEntities, new Predicate<String>() {
            @Override
            public boolean apply(@Nullable String alias) {
                return externalInstanceAliases.contains(alias);
            }
        });
        return ImmutableMap.<String, Object>of(KEY_EXTERNAL_ENTITIES,
relevantExternalEntities);
    }
}
```

The dependencies here are `JiraCaseExchangeRuleStore and JiraEventFilterEvaluator`.

JiraCaseExchangeRuleStore

For `JiraCaseExchangeRuleStore` we simply extend `MemoryCaseExchangeRuleStore` and make it a bean.

JiraCaseExchangeRuleStore.java

```
@Component
public class JiraCaseExchangeRuleStore extends MemoryCaseExchangeRuleStore {
}
```

This class is necessary for correct `conten-jira/case-exchange.json` rules for loading.

JiraEventFilterEvaluator

JiraEventFilterEvaluator.java

```
@Component
public class JiraEventFilterEvaluator extends EventFilterEvaluator {
    private static final String KEY_ITEMS = "items";
    @Override
    protected Map<String, Object> extractProperties(Map<String, Object> entityJson) {
        return getField(entityJson, KEY_ITEMS);
    }
    @Override
    protected String extractOperation(Map<String, Object> entityJson) {
        return null;
    }
}
```

`JiraEventFilterEvaluator` is a class that is responsible for the evaluation of a filter that defines events in `case-exchange.json`. See the JIRA `incidentResolved` event definition for an example. This class will perform the filter on changes detected by the change observer. See javadoc.

getChangedIncidentsForCx operation

`JiraCxPollingCommand` supplies `getChangedIncidentsForCx` for the operation that is used to list changes. The result of this operation will be passed to `JiraEventFilterEvaluator.`

For JIRA this example implementation is recommended:

operations.json

```
...
"getChangedIncidentsForCx": [
        {
            "label": "Get time zone",
            "requestUrlTemplate": "getTimeZoneUrl.ftl",
            "responseTemplate": "getTimeZoneResponse.ftl"
        },
        {
            "label": "Get changed incidents for CX",
            "requestUrlTemplate": "listTicketsUrl.ftl",
            "requestTemplate": "getChangedIncidentsForCxRequest.ftl",
            "responseTemplate": "getChangedIncidentsForCxResponse.ftl",
            "method": "POST"
        }
    ],
...
```

See the ftl template in the example source for details. For your convenience here are `getChangedIncidentsForCxRequest.ftl` and `getChangedIncidentsForCxResponse.ftl`:

getChangedIncidentsForCxRequest.ftl

```
<#assign formatDate =
'com.hp.ccue.serviceExchange.adapter.freemarker.FormatDate'?new()/>
<#escape x as x?json_string>
{
    "maxResults": 1000, <#-- overriding as it defaults to 50; note that the count is
also limited by jira.search.views.default.max -->
    "validateQuery": true,
    "jql": "updated >= \"${formatDate(message.lastUpdateTime, "yyyy-MM-dd HH:mm",
message.tmp.timeZone)}\"",
    "fields": ["environment"],
    "expand": ["changelog"]
}
</#escape>
```

getChangedIncidentsForCxResponse.ftl

```
<#include "jiraConstants.ftl"/>
<#assign writeJson =
'com.hp.ccue.serviceExchange.adapter.freemarker.WriteJson'?new()/>
<#function isHistoryRecordRelevant history lastUpdatedTime>
    <#local timestamp = history.created?date(JIRA_TIME_FORMAT)/>
    <#local referenceTime = lastUpdatedTime?number_to_date/>
    <#return timestamp gte referenceTime/>
</#function>
<#function getExternalEntities issue>
    <#local environment = issue.fields.environment!"[]"/>
    <#attempt>
```

```
        <#local externalEntities = environment?eval/>
        <#recover>
            <#local externalEntities = []/>
    </#attempt>
    <#if externalEntities?is_sequence>
        <#return externalEntities/>
    <#else>
        <#return []/>
    </#if>
</#function>
<#escape x as x?json_string>
{
    "histories": [
        <#assign firstItem = true>
        <#list doc.result.issues as issue>
            <#assign externalEntities = getExternalEntities(issue)/>
            <#if externalEntities?has_content> <#-- exclude records without external
entities -->
                <#list issue.changelog.histories as history>
                    <#if isHistoryRecordRelevant(history, message.lastUpdateTime)>
<#-- exclude records older than lastUpdateTime-->
                        <#if !firstItem>,<#else><#assign firstItem = false></#if>
        {
            "entityId": "${issue.key}",
            "externalEntities": {
            <#list externalEntities as externalEntity>
                "${externalEntity.externalInstanceAlias}": {
                    "entityId": "${externalEntity.externalEntityId}",
                    "entityType": "${externalEntity.externalEntityType}"
                }<#if externalEntity_has_next>,</#if>
            </#list>
            },
            "items": {
                <#list history.items as item>
                "${item.field}": {
                    "from": <#noescape>${writeJson(item.from)}</#noescape>,
                    "to": <#noescape>${writeJson(item.to)}</#noescape>
                }<#if item_has_next>,</#if>
                </#list>
            }
        }
                    </#if>
                </#list>
            </#if>
        </#list>
    ],
    "Date": "${doc.resultHeaders.Date}",
```

```
    "tmp": null
}
</#escape>
```

Case exchange pipeline

So far change listening and event filtering have been implemented. We have enabled that an event is triggered after a defined change, and a message is sent to HP SX. Now it is necessary to process the message correctly.

The CX message processing needs to be assigned into a case exchange pipeline.

JiraAdapter.java

```
...
    @Override
       protected String getPipelineNameForMessage(MessageProperties properties,
Map<String, Object> amqpMessage) {
           final String subType = extractMessageSubtype(properties.getType());
           if (MessageSubType.CHANGE.equals(subType)) {
            final String reason = getStrField(amqpMessage, EntityChangeMsg.REASON);
            switch (reason) {
                ...
                case EntityChangeMsg.REASON_CASE_EXCHANGE:
                    return JiraConstants.PIPELINE_CASE_EXCHANGE_CHANGE;
                default:
                    throw new IllegalArgumentException();
            }
        }
           return super.getPipelineNameForMessage(properties, amqpMessage);
       }
...
```

The pipeline is built in `JiraPipelineBuilder`:

```
@Component
public class JiraPipelineBuilder implements AdapterPipelineBuilder {
    @Autowired
       private JiraCaseExchangeAdapter caseExchangeAdapter;
    @Override
    public Pipeline buildPipeline(Adapter adapter, PipelineBuilderFactory factory,
String name) {
        switch (name) {
   ...
            case JiraConstants.PIPELINE_CASE_EXCHANGE_CHANGE:
                return buildCaseExchangeChangePipeline(factory);
           ...
        }
    }

   ...

    private Pipeline buildCaseExchangeChangePipeline(PipelineBuilderFactory factory) {
           final PipelineBuilder builder =
factory.newBuilder(JiraConstants.PIPELINE_CASE_EXCHANGE_CHANGE);
           builder.addBlock(new CaseExchangeChangeHandlerBlock(caseExchangeAdapter));
           return builder.build();
       }
}
```

The `CaseExchangeHandlerBlock` that is supplied with `JiraCaseExchangeAdapter`. `CaseExchangeChangeHandlerBlock` is used. It is the crucial block that decides, based on the `content-case-exchange/case-exchange.json` configuration, which actions will be performed. Basically it ensures that the correct event group action is taken. `JiraCaseExchangeAdapter` supplies the correct rulestore. See the following example:

JiraCaseExchangeAdapter

```
@Component
public class JiraCaseExchangeAdapter extends AbstractCaseExchangeAdapter {
    private static final Logger log =
LoggerFactory.getLogger(JiraCaseExchangeAdapter.class);
    private CaseExchangeRuleStore ruleStore;
    @Autowired
    public JiraCaseExchangeAdapter(JiraCaseExchangeRuleStore ruleStore) {
        this.ruleStore = ruleStore;
    }
    @Override
    public void boot() {
        if (log.isTraceEnabled()) {
            log.trace("booting");
        }
        if (log.isTraceEnabled()) {
            log.trace("boot completed");
        }
    }
    @Override
    public void shutdown() {
        if (log.isTraceEnabled()) {
            log.trace("shutdown");
        }
    }
    @Override
    public String getAdapterType() {
        return JiraConstants.JIRA_TYPE;
    }
    @Override
    public void registerEntityChangeListener(CaseExchangeEntityChangeListener
listener) {
        ruleStore.registerEntityChangeListener(listener);
    }
    @Override
    public void
updateCaseExchangeEntityChangeListeners(Set<CaseExchangeEntityChangeListener>
listeners) {
        ruleStore.updateCaseExchangeEntityChangeListeners(listeners);
    }
    @Override
    public void unregisterEntityChangeListener(String instance, String entityType,
String entityId, String changeReason) {
        ruleStore.unregisterEntityChangeListener(instance, entityType, entityId,
changeReason);
    }
    @Override
    public void onConfigurationReloaded() {
        // not supported
    }
}
```

To finish the implementation of the second use case the `retrieveIncident` and `convertIncidentToCanonicalModel` must be defined:

operations.json

```
...
    "retrieveIncident": [
        {
            "label": "Retrieve incident",
            "requestUrlTemplate": "retrieveIndicidentUrl.ftl",
            "responseTemplate": "retrieveIndicidentResponse.ftl"
        }
    ],
    "convertIncidentToCanonicalModel": [
        {
            "label": "Convert incident to canonical model",
            "resultTemplate": "convertIncidentToCanonicalModelResult.ftl"
        }
    ],
...
```

The `retrieveIncident` operation basically retrieves the issue and stores its data in the `entityChange.entity` field in the message. Here is the `convertIncidentToCanonicalModelResult.ftl` template:

convertIncidentToCanonicalModelResult.ftl

```
<#include "jiraCaseExchangeUtils.ftl"/>
<#assign
loadConfig='com.hp.ccue.serviceExchange.adapter.freemarker.LoadConfig'?new()/>
<#assign
findExtSystemForAlias='com.hp.ccue.serviceExchange.adapter.freemarker.caseex.FindExterna
/>
<#assign jiraMapping = loadConfig(context.contentStorage, "jira/jira-cx-mappings")/>
<#assign entityChange = message.entityChange />
<#assign externalEntities = entityChange.data.externalEntities![]/>
<#assign entity = entityChange.entity />
<#escape x as x?json_string>
{
    "event": "${entityChange.changeReason}",
    "entity": {
        "instanceType": "${entityChange.instanceType}",
        "instance": "${entityChange.instance}",
        "entityType": "${entityChange.entityType}",
        "entityId": "${entityChange.entityId}",
        "properties": {
            "Status": "${getMappingKey(jiraMapping.Incident.Status,
entity.fields.status.id)}"
        }
    },
    "linkedEntities": [
    <#list externalEntities?keys as alias>
        <#assign aliasHash = findExtSystemForAlias(context.appContext,
entityChange.instanceType, entityChange.instance, alias)!{} />
        <#assign externalEntity = externalEntities[alias] />
        {
            "instanceAlias": "${alias}",
            "instanceType": "${aliasHash.targetInstanceType}",
            "instance": "${aliasHash.targetInstance}",
            "entityType": "${externalEntity.entityType}",
            "entityId": "${externalEntity.entityId}"
        }<#if alias_has_next>,</#if>
    </#list>
    ],
    "entityChange": {}
}
</#escape>
```

The implementation of the CX process has been described up to the point where the content-case-exchange/case-exchange.json configuration actions will be taken. That means `IncindentCaseExchangeOOFlow` is invoked. This flow will send messages to execute operations in the linked systems. In our use case the HP SM incident will be resolved.

The second use case can now be tested like this:

- Choose an existing HP SM incident or create a new one. Set the status field to "Pending Vendor" and the Vendor field to "jira" (or whatever is the targetAlias for JIRA in the `external-systems.json` file.) Click **Save** and **Exit**.
  You should now observe that:
  - A corresponding JIRA issue has been created.
- Resolve the JIRA issue.

- If you now examine the HP SM incident, it should have the Resolved status as well.

Request to fulfill use case

In this chapter how to implement a request-to-fulfill (R2F) capable adapter with JIRA as an example fulfillment system is explained. This information is extending the example implementation described in Ticketing use case and Case exchange use case.

It is necessary that you have access to the example implementation sources contained in the HP SX distribution. You will need to refer to the source code to be able to fully understand the contents of this topic. Especially refer to the ftl templates and the OO flow.

It is also advisable to refer to the SX API javadoc.

Finally, you may also want to consult the How to extend HP SX Content (HP SM Problem entity) topic which has details about many development techniques, including:

- testing REST APIs with browser clients
- uploading SX content packs using SX content management UI and SX upload maven plugin
- analysis of SX log files
- OO - development of new OO content packs, OO UI.

Use case definition

For the purpose of this example, fulfillment in JIRA will be interpreted as performing a JIRA task (i.e. to an issue of type Task) in a chosen project in JIRA. Projects will play the role of catalog items. Also included is a slightly artificial approval process in order to demonstrate how to implement support for the approve/deny operations. The use case can be summarized in the following flow:

- The user creates an order in Propel. When creating an order, the user chooses a project in which to create a task and specifies the properties of this task (title, description, reporter, priority). As a result, a task in JIRA is created with **Open** status and an email is sent to the lead of the JIRA project who is acting as the Approver. Additionally, a notification is sent to the catalog.
- The lead invokes the approve operation in Propel. As a result, the task status is set to **In Progress** and a notification email is sent to the reporter. Additionally, a notification is sent to the catalog.
- A developer resolves the task in JIRA. As a result, the reporter receives an email. Additionally, a notification is sent to the catalog.
- A developer **closes** the task in JIRA. As a result, HP SX stops listening out for changes of this task.

Implementation

Initial changes

First, add "r2f" to the feature list in `content-jira/src/main/resources/metadata.json`. Before the actual implementation, try to call the REST endpoint for creating orders in a browser REST client such as Postman - http://www.getpostman.com/ or DHC, and see what happens.

```
POST /sx/api/request
Content-Type: application/json
Authorization: Basic c3hDYXRhbG9nVHJhbnNwb3J0VXNlcjpzeENhdGFsb2dUcmFuc3BvcnRVc2Vy
{
    "@type": "urn:x-hp:2014:software:cloud:data_model:sx:request",
    "messageType": "order",
    "name": "I need 'Service Exchange Task'",
    "description": "I need to perform a 'Service Exchange Task'",
    "items": [
        {
            "id": "SE",
            "name": "Service Exchange Task",
            "quantity": "1",
            "recipient": {
                "@self": "",
                "@type": "urn:x-hp:2014:software:cloud:data_model:sx:person",
                "name": "consumer"
            },
            "properties": [
                {
                    "@type":
"urn:x-hp:2014:software:cloud:data_model:property:select",
                    "name": "priority",
                    "value": "3",
                    "displayName": "Priority"
                }
            ],
            "route": {
                "@type": "urn:x-hp:2014:software:cloud:data_model:sx:route",
                "system_type": "JIRA",
                "target_instance": "http://mpavmint01.hpswlabs.adapps.hp.com:8080"
            }
        }
    ]
}
```

This example uses basic HTTP authentication with the notification user. See `WEB-INF/sx.properties`, properties `catalog.notificationUser` and `catalog.notificationUserPassword`, Defaults are `sxCatalogTransportUser/sxCatalogTransportUser.` Note that the Postman client (or other) can create the basic authentication header for you.

Normal authentication - as documented in Appendix A: Service Exchange - API - is done with an IdM token passed via the X-Auth-Token header; however, we also support basic authentication for debugging purposes.

NOTE: The key of the JIRA project (where to create the task) ("SE") is used as the id of the catalog item, and HP SX passes priority as an example of a catalog item property.

If you now perform the REST call, you will find the following exception from the RequestResource class in the logs:

```
java.lang.NullPointerException: messageHeader == null
```

This indicates that it is necessary to set a request message header template in the JiraAdapter. For this, add the following setRequestMessageHeaderTemplate() call to the JiraAdapter constructor:

JiraAdapter.java

```
...
 public JiraAdapter(JiraOperationExecutor operationExecutor, JiraPipelineBuilder
pipelineBuilder) {
        super(JiraConstants.JIRA_ADAPTER_NAME, operationExecutor, pipelineBuilder);

setRequestMessageHeaderTemplate("jira/sx/templates/generateMessageHeader.ftl");
    }
...
```

Then add the following `generateMessageHeader.ftl` template (whose structure is suggested in the javadoc of the setRequestMessageHeaderTemplate() method):

generateMessageHeader.ftl

```
<#assign route = items[0].route/>
<#escape x as x?json_string>
{
    "messageHeader": {
        "messageType": "${messageType}",
        "backendSystemType": "${route.system_type}",
        "targetInstance": "${route.target_instance}"
    }
}
</#escape>
```

Although it is not strictly necessary, override also the decorateRequestMessage() method in the JiraAdapter. This proactively converts a URL passed in the POST body to an instance ID from the `jira/instances.json` file:

JiraAdapter.java

```
...
    @Override
    public void decorateRequestMessage(Map<String, Object> message) {
        super.decorateRequestMessage(message);
        Map<String, Object> messageHeader = getField(message,
MessageConstants.MESSAGE_HEADER);
        Map<String, Object> instances =
configuration.getConfiguration(JiraInstancesCfg.CFG_NAME);
        MessageUtils.fixTargetInstanceInMessageHeader(messageHeader, instances);
    }
...
```

With these changes in place, if you now perform the REST call you will get the following exception from the RequestResource class:

```
java.lang.RuntimeException: Failed to find flow config for JIRA adapter,
messageType=order
```

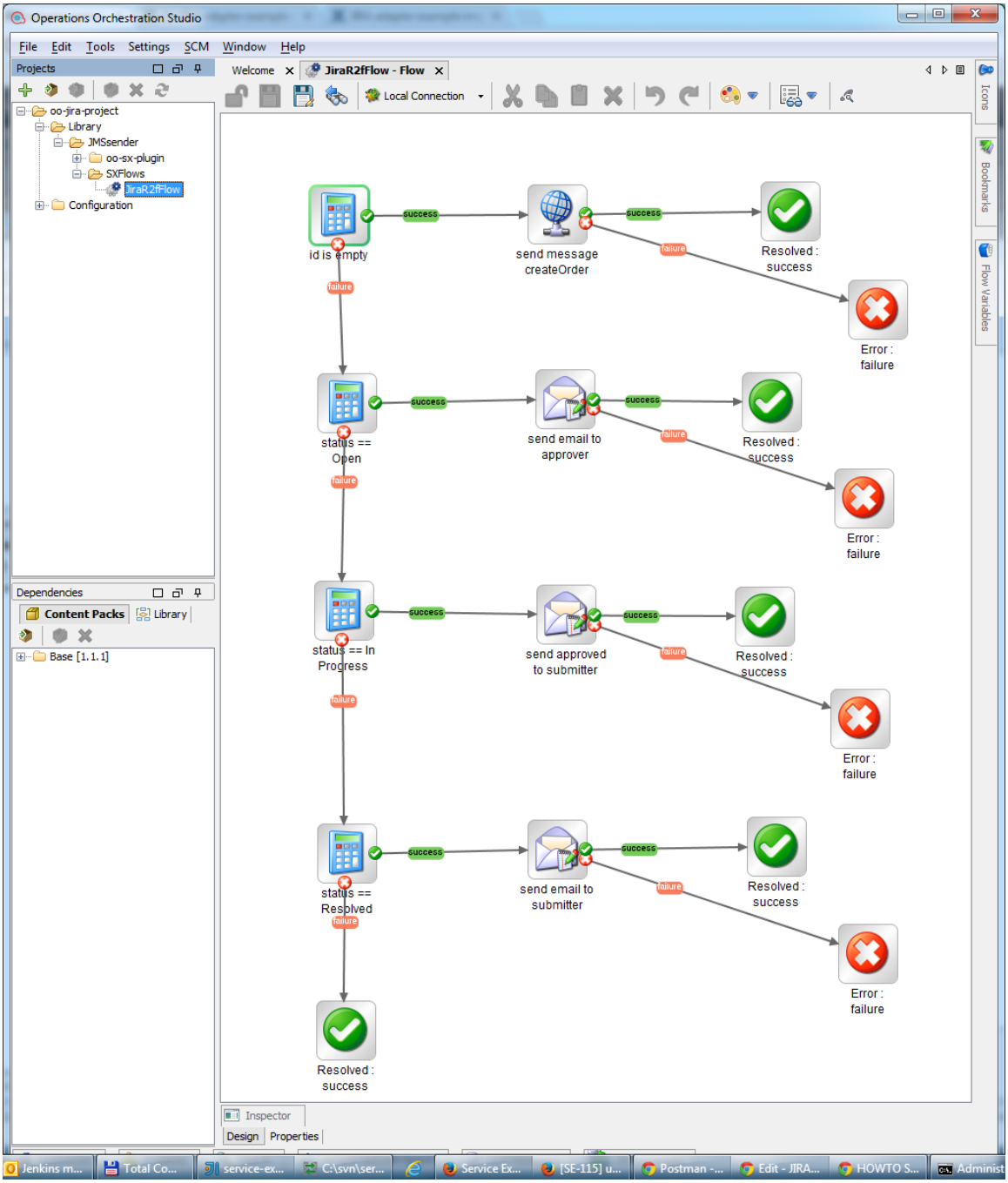
This is because HP SX is trying to invoke an OO flow to initiate the creation.

OO Flow and the createOrder operation

Now you must create the necessary OO flow and the related infrastructure, for details see the example sources.

1. Create a maven module `oo-jira-cp` similar to the `oo-sm-problem-cp` module in How to extend HP SX Content (HP SM Problem entity).
2. Modify the `pom.xml` in `content-jira-cp` to refer to `oo-jira-cp`, as it was done for `content-sm-problem`.
3. Create an OO project `oo-jira-project` in `oo-jira-cp/src/main/resource`.

The OO flow will look like this:



If the incoming message does not contain the task ID, the OO flow sends a message to HP SX, which (through the PLAIN pipeline) executes an operation named `createOrder`. Otherwise, if the message contains an ID, the OO flow sends a notifying email message based on the task status. For details, see the example sources.

Next, to register the flow in HP SX and specify its input parameter bindings, add a `flows.json` file to `content-jira/src/main/resources/sx`:

flows.json

```
{
    "JIRA": {
        "order": {
            "flowId": <your_flow_id>,
            "compressMessage": true,
            "parameters": [
                {
                    "name": "sxConfiguration.jmsBroker",
                    "valueSelector": "$.JMS_BROKER.endpoint",
                    "source": "infrastructure"
                },
    // ... and other properties coming from configuration files
                {
                    "name": "orderInfo.id",
                    "valueSelector": "$.orderInfo.id",
                    "source": "message"
                },
                {
                    "name": "orderInfo.title",
                    "valueSelector": "$.orderInfo.title",
                    "source": "message"
                },
                // ... and other properties coming from the input message
            ]
        }
    }
}
```

If you now call the REST request endpoint, HP SX will invoke the OO flow and return a success response similar to this:

```
{
    "id":"38628a16-1728-4e61-aab2-befc75f63172"
}
```

This is a reference ID for the request and is generated by HP SX. This document calls this id **externalId**. However, the actual creation of the task will fail as the createOrder operation is not yet defined, which can be done in this way:

operations.json

```
"createOrder": [
        {
            "label": "Create order",
            "requestUrlTemplate": "createTicketUrl.ftl",
            "requestTemplate": "createOrderRequest.ftl",
            "responseTemplate": "createOrderResponse.ftl",
            "method": "POST"
        }
    ],
```

The step is straight-forward as you can see from the following templates:

createOrderRequest.ftl

```
<#assign item = message.items[0]/>
<#assign valueMap = {}/>
<#list item.properties as property>
    <#assign valueMap = valueMap + {property.name : property.value}/>
</#list>
<#escape x as x?json_string>
{
    "fields": {
        "summary": "${message.name}",
        <#if message.description?has_content>
        "description": "${message.description}",
        </#if>
        "project": {
            "key": "${item.id}"
        },
        "issuetype": {
            "id": "3"
        },
        "priority": {
            "id": "${valueMap.priority}"
        },
        "reporter": {
            "name": "${item.recipient.name}"
        }
    }
}
</#escape>
```

createOrderResponse.ftl

```
<#escape x as x?json_string>
{
    "orderInfo": {
        "id": "${doc.result.key}"
    }
}
</#escape>
```

If you now call the REST request endpoint, the new task will finally be created in JIRA. Check that it is created with the chosen title (i.e. summary), description, project, priority, and reporter, and that it has the initial Open status.

Change listening

Support for R2F changes in JiraChangeObserver

Now JiraChangeObserver needs to be changed so that it notifies about R2F changes:

JiraChangeObserver.java

```
@Component
public class JiraChangeObserver extends CompositeChangeObserver {
    @Value("${adapter.jira.change.listener.delayBeforeNextRun}")
    private int pollInterval;
    @Autowired
    public JiraChangeObserver(JiraR2fPollingCommand r2fPollingCommand,
JiraCxPollingCommand cxPollingCommand) {
        super(ImmutableList.<Runnable>of(r2fPollingCommand, cxPollingCommand));
    }
    @Override
    public int getPollIntervalSec() {
        return pollInterval;
    }
}
```

Here is the JiraR2fPollingCommand that performs the actual polling functionality:

JiraR2fPollingCommand.java

```
@Component
public class JiraR2fPollingCommand extends R2fPollingCommand {
    private static final String KEY_ENTITY_ID = "entityId";
    private static final String KEY_ENTITIES = "entities";
    /**
     * Operation for fetching changed incidents.
     */
    public static final String OPERATION_GET_CHANGES = "getChangedIncidentsForR2f";
    @Autowired
    public JiraR2fPollingCommand(JiraOperationExecutor operationExecutor) {
        super(JiraConstants.JIRA_TYPE, JiraConstants.ENTITY_INCIDENT,
JiraInstancesCfg.CFG_NAME, OPERATION_GET_CHANGES, operationExecutor);
    }
    @Nonnull
    @Override
    protected List<Map<String, Object>> extractChangedEntities(Map<String, Object>
changedEntities) {
        return getField(changedEntities, KEY_ENTITIES);
    }
    @Nonnull
    @Override
    protected String extractEntityId(Map<String, Object> entity) {
        return getStrField(entity, KEY_ENTITY_ID);
    }
}
```

For details about this class, see the javadoc R2fPollingCommand. In short, the command performs the following

- It executes the getChangedIncidentsForR2f operation which is passed the lastUpdateTime as the timestamp for which changes are to be retrieved.
- For each changed entity whose ID is registered in HP SX for R2F polling (as if stored with DefaultNotificationSetupExecutor, see the next section for details about registration), an entity changed message is sent to HP SX.
- It extracts a new timestamp from the operation result and saves it to the database.

Here is the getChangedIncidentsForR2f operation:

operations.json

```
...
 "getChangedIncidentsForR2f": [
        {
            "label": "Get time zone",
            "requestUrlTemplate": "getTimeZoneUrl.ftl",
            "responseTemplate": "getTimeZoneResponse.ftl"
        },
        {
            "label": "Get changed incidents for R2F",
            "requestUrlTemplate": "listTicketsUrl.ftl",
            "requestTemplate": "getChangedIncidentsForR2fRequest.ftl",
            "responseTemplate": "getChangedIncidentsForR2fResponse.ftl",
            "method": "POST"
        }
    ]
...
```

This operation will return JIRA tasks updated or created since the last check for changes:

getChangedIncidentsForR2fRequest.ftl

```
<#assign formatDate =
'com.hp.ccue.serviceExchange.adapter.freemarker.FormatDate'?new()/>
<#escape x as x?json_string>
{
    "maxResults": 1000, <#-- overriding as it defaults to 50; note that the count is
also limited by jira.search.views.default.max -->
    "validateQuery": true,
    "jql": "updated >= \"${formatDate(message.lastUpdateTime, "yyyy-MM-dd HH:mm",
message.tmp.timeZone)}\""
}
</#escape>
```

getChangedIncidentsForR2fResponse.ftl

```
<#include "jiraConstants.ftl"/>
<#function isIssueRelevant issue lastUpdatedTime>
    <#local timestamp = issue.fields.updated?date(JIRA_TIME_FORMAT)/>
    <#local referenceTime = lastUpdatedTime?number_to_date/>
    <#return timestamp gte referenceTime/>
</#function>
<#escape x as x?json_string>
{
    "entities": [
        <#assign firstItem = true>
        <#list doc.result.issues as issue>
            <#if isIssueRelevant(issue, message.lastUpdateTime)>
                <#if !firstItem>,<#else><#assign firstItem = false></#if>
        {
            "entityId": "${issue.key}"
        }
            </#if>
        </#list>
    ],
    "Date": "${doc.resultHeaders.Date}",
    "tmp": null
}
</#escape>
```

Registering our task for change observing

It is now necessary to make sure that this task is registered for change observing, otherwise the changes will be ignored by the JiraR2fPollingCommand.

To do this add a second step to the createOrder operation:

operations.json

```
"createOrder": [
...
        {
            "label": "Setup notifications",
            "notifyTemplate": "notifyTemplate.ftl",
            "callbackTemplate": "callbackTemplate.ftl",
            "operationName": "checkOrder",
            "idSelector": "$.orderInfo.id"
        }
    ],
```

This step is a SetupNotifications step, see Appendix B: Operation executors topic for details. In order that the JiraOperationExecutor properly supports this type of step, its executeNotificationSetup() method must be overridden. This is done in the following way:

JiraOperationExecutor.java

```
...
    @Autowired
    public JiraOperationExecutor(JiraNotificationSetupExecutor
notificationSetupExecutor) {
        super(JiraConstants.JIRA_ADAPTER_NAME, JiraInstancesCfg.CFG_NAME);
        this.notificationSetupExecutor = notificationSetupExecutor;
        ...
    }
...
 @Override
    protected void executeNotificationSetup(
            String entityId, String checkOperation, Map<String, Object>
checkOperationInputMessage, String catalogCallbackTemplate,
            EntityRegistrationMode mode, Map<String, Object> context, Map<String,
Object> stepConfig) throws Exception {
        notificationSetupExecutor.executeNotificationSetup(
                entityId, checkOperation, checkOperationInputMessage,
catalogCallbackTemplate, mode, context, stepConfig);
    }
...
```

That is, the actual step execution is delegated to a subclass of DefaultNotificationSetupExecutor called JiraNotificationSetupExecutor:

JiraNotificationSetupExecutor.java

```
@Component
public class JiraNotificationSetupExecutor extends DefaultNotificationSetupExecutor {
    public JiraNotificationSetupExecutor() {
        super(JiraConstants.JIRA_TYPE, JiraConstants.ENTITY_INCIDENT);
    }
}
```

For details of the implementation, see DefaultNotificationSetupExecutor javadocs.

In short, the step does the following:

- It extracts the JIRA task ID from the input message using the JSONPath $.orderInfo.id.
- It performs a FreeMarker transformation with the notifyTemplate.ftl template.
- It registers the task in the SX database, namely it stores
    - the externalId
    - the JIRA task id
    - the operation name to be executed when changes are detected ("checkOrder" here)
    - the result of the notifyTemplate.ftl transformation (which will serve as input for the checkOrder operation)
    - the callbackTemplate property value which is the path of a FreeMarker template from which catalog notifications will be generated.

Here is the `notifyTemplate.ftl` file:

notifyTemplate.ftl

```
<#assign route = message.items[0].route/>
<#assign messageHeader = message.messageHeader/>
<#escape x as x?json_string>
{
    "messageHeader": {
    <#list messageHeader?keys as key>
        "${key}": "${messageHeader[key]}"<#if key_has_next>,</#if>
    </#list>
    },
    "@type": "${message.@type}",
    "name": "${message.name}",
    "description": "${message.description}",
    "items": [
        {
            "route": {
                "@type": "${route.@type}",
                "system_type": "${route.system_type}",
                "target_instance": "${route.target_instance}"
            }
        }
    ],
    "orderInfo": {
        "id": "${message.orderInfo.id}"
    }
}
</#escape>
```

The checkOrder operation and the callbackTemplate.ftl template are outlined in the next section - they can be left empty for now.

Pipeline for handling R2F changes

If you now test the observer by making changes to a registered entity, you will see that the JIRA adapter will try to handle an incoming message of subtype MessageSubType.CHANGE and a reason EntityChangeMsg.REASON_SX_MANAGED.

To add support for such a message:

JiraAdapter.java

```
...
      @Override
       protected String getPipelineNameForMessage(MessageProperties properties,
Map<String, Object> amqpMessage) {
           final String subType = extractMessageSubtype(properties.getType());
           if (MessageSubType.CHANGE.equals(subType)) {
            final String reason = getStrField(amqpMessage, EntityChangeMsg.REASON);
            switch (reason) {
                case EntityChangeMsg.REASON_SX_MANAGED: // Adding pipeline name
                    return JiraConstants.PIPELINE_SX_MANAGED_CHANGE;
                case EntityChangeMsg.REASON_CASE_EXCHANGE:
                    return JiraConstants.PIPELINE_CASE_EXCHANGE_CHANGE;
                default:
                    throw new IllegalArgumentException();
            }
        }
           return super.getPipelineNameForMessage(properties, amqpMessage);
       }
...
```

PipelineBuilder.java

```
...
   @Autowired
    private StorageFactory storageFactory;
    @Autowired
    private CatalogNotificationMessagePublisher cnPublisher;
    @Autowired
    private OoFlowMessagePublisher ooFlowMessagePublisher;
    @Autowired
    private MessageTransformer messageTransformer;
...


   @Override
  public Pipeline buildPipeline(Adapter adapter, PipelineBuilderFactory factory,
String name) {
        switch (name) {
   ...
    case JiraConstants.PIPELINE_SX_MANAGED_CHANGE:
                return buildSxManagedChangePipeline(factory);
           ...
        }
  }
...
   public Pipeline buildSxManagedChangePipeline(PipelineBuilderFactory factory) {
        PipelineBuilder builder =
factory.newBuilder(JiraConstants.PIPELINE_SX_MANAGED_CHANGE);
        // prepare entityInfo variable
        ContextVariable<EntityInfo> entityInfo =
ContextVariable.newDataValue(EntityInfo.class, ENTITY_INFO_PROPERTY_PATH);
        // retrieve entity info from database
        // keep reference to RetrieveEntityInfoBlock - we want its VAR_TARGET_INSTANCE
```

```
        RetrieveEntityInfoBlock retrieveEntityInfoBlock;
        builder.addBlock(retrieveEntityInfoBlock = new
RetrieveEntityInfoBlock(JIRA_TYPE, storageFactory, entityInfo));
        // operation execution from entity info
        builder.addBlock(new EntityInfoOperationExecutorBlock(
                storageFactory,
                entityInfo));
        // extract target instance (it is not in the message anymore)
        ContextVariable<String> targetInstance =
retrieveEntityInfoBlock.descVariable(RetrieveEntityInfoBlock.VAR_TARGET_INSTANCE,
String.class).binding;
        // catalog notification
        builder.addBlock(new EntityInfoAwareCatalogNotificationBlock(
                JIRA_TYPE,
                cnPublisher,
                storageFactory,
                messageTransformer,
                targetInstance,
                null,
                entityInfo
        ));
        // finally if not explicitly suppressed, notify OO
        builder.addBlock(new OoInvocationBlock(CONDITIONAL_OO_FLOW_BLOCK_NAME_SUFFIX,
ooFlowMessagePublisher) {
            protected boolean isInterested(ExecutionContext context) {
                return !context.message.isEmpty() &&
!context.message.containsKey(MessageDirectives.SKIP_FLOW_RUN);
            }
        });
        // entity change cleanup
        builder.addBlock(new EntityChangeCleanupBlock(
                JIRA_TYPE,
                storageFactory,
                targetInstance));
```

```
        return builder.build();
    }
...
```

To summarize the purpose of each block added into the pipeline.

- RetrieveEntityInfoBlock
  - Using the ID of the entity, it retrieves the information stored at the notification setup time, see the previous section.
- EntityInfoOperationExecutorBlock
  - Executes the check operation, called "checkOrder" in this example.
- EntityInfoAwareCatalogNotificationBlock
  - Applies the catalog notification template to the message and notifies the catalog with the resulting message.
- OoInvocationBlock
  - Calls the OO flow.
- EntityChangeCleanupBlock
  - Deregisters listening out for entity changes based on whether the output message of the checkOrder operation contains the stopListenting directive.

Now, to implement the checkOrder specified in the notification setup, follow this example:

operations.json

```
...
 "checkOrder": [
        {
            "label": "Get order",
            "requestUrlTemplate": "getOrderUrl.ftl",
            "responseTemplate": "getOrderResponse.ftl"
        },
        {
            "label": "Get order approver",
            "requestUrlTemplate": "getOrderApproverUrl.ftl",
            "responseTemplate": "getOrderApproverResponse.ftl"
        },
        {
            "label": "Get order approver info",
            "requestUrlTemplate": "getOrderApproverInfoUrl.ftl",
            "responseTemplate": "getOrderApproverInfoResponse.ftl"
        }
    ],
...
```

The first step gets most of the task data:

getOrderResponse.ftl

```
<#escape x as x?json_string>
<#assign fields = doc.result.fields/>
{
    "orderInfo": {
        "id": "${doc.result.key}",
        "title": "${fields.summary}",
        "description": "${fields.description!""}",
        "status": {
            "id": "${fields.status.id}",
            "name": "${fields.status.name}"
        },
        <#if fields.resolution??>
        "resolution": {
            "id": "${fields.resolution.id}",
            "name": "${fields.resolution.name}"
        },
        </#if>
        "resolutiondate": "${fields.resolutiondate!""}",
        "reporter": {
            "name": "${fields.reporter.name}",
            "emailAddress": "${fields.reporter.emailAddress}"
        },
        "project": {
            "id": "${fields.project.key}",
            "name": "${fields.project.name}"
        },
        "itemName": "${fields.project.name} Task"
    }
    <#if fields.status.id == "6"> <#-- Closed -->
    ,
    "stopListening": "${doc.result.key}"
    </#if>
}

</#escape>
```

NOTE: The stopListening directive is used to unregister a task from HP SX for a task with a Closed status.

The other two steps in the checkOrder operation retrieve information about the approver and store it in under orderInfo.approver.

Here is the callbackTemplate.ftl file which is used by EntityInfoAwareCatalogNotificationBlock to create the notification message:

callbackTemplate.json

```
<#assign SUBMITTED = "submitted">
<#assign PENDING_APPROVAL = "pending_approval">
<#assign APPROVED = "approved">
<#assign IN_PROGRESS = "in_progress">
<#assign COMPLETED = "completed">
<#assign REJECTED = "rejected">
<#function toSxStatus jiraStatus jiraResolution>
    <#switch jiraStatus>
        <#case "1"> <#-- Open -->
            <#return PENDING_APPROVAL>
        <#case "3"> <#-- In Progress -->
            <#return APPROVED/>
        <#case "4"> <#-- Reopened -->
            <#return IN_PROGRESS/>
        <#case "5"> <#-- Resolved - fall through -->
        <#case "6"> <#-- Closed -->
            <#switch jiraResolution>
                <#case "1"> <#-- Fixed -->
                    <#return COMPLETED/>
                <#case "2"> <#-- Won't Fix -->
                    <#return REJECTED/>
                <#case "3"> <#-- Duplicate -->
                    <#return COMPLETED/>
                <#case "4"> <#-- Incomplete -->
                    <#return REJECTED/>
                <#case "5"> <#-- Cannot Reproduce -->
                    <#return REJECTED/>
                <#case "10000"> <#-- Done -->
                    <#return COMPLETED>
            </#switch>
    </#switch>
</#function>
<#escape x as x?json_string>
<#assign orderInfo = message.orderInfo/>
<#assign status = toSxStatus(orderInfo.status.id, (orderInfo.resolution.id)!"")/>
[
    {
        "@type": "urn:x-hp:2014:software:cloud:data_model:sx:notification",
        "id": "${message.messageHeader.externalId}",
        "remoteId": "${orderInfo.id}",
        "displayName": "${orderInfo.title}",
        "requestor": "${orderInfo.reporter.name}",
        "state": "${status}",
        <#if status == PENDING_APPROVAL>
        "approvers" : [
            {
                "name": "${orderInfo.approver.name}"
            }
        ],
        </#if>
        "subscription": {
            "id": "N/A"
        }
    }
]
</#escape>
```

Test of the change listening implementation

You can now test the implementation like this:

- Call the create-order REST endpoint. You should now observe that:
  - A new task is created in JIRA.
  - A catalog notification has been sent reporting the status "pending_approval" and showing the project lead as Approver, which you can check in the notification.log.

```
{
    "@type": "urn:x-hp:2014:software:cloud:data_model:sx:notification",
    "id": "d4cac721-15bc-43f7-87ec-6ea7d3748537",
    "remoteId": "SE-6469",
    "displayName": "I need 'Service Exchange Task'",
    "requestor": "consumer",
    "state": "pending_approval",
    "approvers": [
        {
            "name": "joe.manager"
        }
    ],
    "subscription": {
        "id": "N/A"
    }
}
```

  - A notification mail has been sent to the Approver asking them to approve or deny the request.
- Now resolve the task with the resolution Fixed in JIRA. You should now observe that:
  - A catalog notification has been sent reporting the status "completed":

```
{
    "@type": "urn:x-hp:2014:software:cloud:data_model:sx:notification",
    "id": "d4cac721-15bc-43f7-87ec-6ea7d3748537",
    "remoteId": "SE-6469",
    "displayName": "I need 'Service Exchange Task'",
    "requestor": "consumer",
    "state": "completed",
    "subscription": {
        "id": "N/A"
    }
}
```

  - A notification mail has been sent to the reporter.
- If you now close the task in JIRA, HP SX will stop observing changes of the task. You can check this by reopening the task where you will see that no catalog notifications have been sent.

Approve/deny operation

Approval still needs to be implemented. The approval operation is invoked through the SX RESTful API (/operation resource), as documented in Appendix A: Service Exchange - API. Try to call it now with an Open task:

```
GET /sx/api/operation?messageText=<encodedMessage>
Content-Type: application/json
Authorization: Basic am9lLm1hbmFnZXI6Y2hhbmdlaXQ=
```

Here the encodedMessage is base64 encoding of:

```
{
    "@type": "urn:x-hp:2014:software:cloud:data_model:sx:invoke",
    "entityId": "d4cac721-15bc-43f7-87ec-6ea7d3748537",
    "entityType": "request",
    "operationName": "approve",
    "recipient": {
        "name": "joe.manager"
    },
    "parameters": [
        {
            "name": "message",
            "value": "Approved."
        }
    ]
}
```

You can use an online encode e.g. on https://www.base64encode.org/.

NOTE: The documented POST endpoint is not used as it requires an IdM token passed via the X-Auth-Token header. Instead a GET version with basic authentication is used, which exists for debugging purposes. If you perform the call, it will succeed, but the actual approval will fail with the following exception in the log:

```
java.lang.IllegalArgumentException: pipeline OPERATION not recognized
```

To add support for the OPERATION pipeline to JiraPipelineBuilder:

JiraPipelineBuilder.java

```
...
  @Override
  public Pipeline buildPipeline(Adapter adapter, PipelineBuilderFactory factory,
String name) {
        switch (name) {
   ...
    case Names.PIPELINE_OPERATION:
                return buildOperationPipeline(factory);
           ...
        }
  }
...
  private Pipeline buildOperationPipeline(PipelineBuilderFactory factory) {
        final PipelineBuilder builder = factory.newBuilder(Names.PIPELINE_OPERATION);
        builder.addBlock(new OperationExecutionBlock());
        ContextVariable<Map> catalogNotificationMessage =
ContextVariable.newDataMap(CATALOG_NOTIFICATION_MESSAGE_PROPERTY_PATH);
        builder.addBlock(new PrepareCatalogNotificationMessageBlock(

ContextVariable.newFixedValue(MessageConstants.RequestState.COMPLETED),
                catalogNotificationMessage));
        builder.addBlock(new CatalogNotificationBlock(cnPublisher,
                // notification message
                catalogNotificationMessage,
                // entity ID is 'id' in the message
                ContextVariable.newMessageString(MessageConstants.ID),
                // notification type - always request
                ContextVariable.newFixedValue(NotificationType.REQUEST)
        ));
        return builder.build();
    }
...
```

Beside the operation execution, a notification to the catalog is performed. If you now call the approval endpoint, the approval will still fail, this time with the following exception:

```
java.lang.IllegalArgumentException: JIRA: invalid operation requested: 'approve'
```

So implement the approve operation:

operations.json

```
...
"approve": [
        {
            "label": "Approve order",
            "requestUrlTemplate": "approveOrderUrl.ftl",
            "requestTemplate": "approveOrderRequest.ftl",
            "method": "POST"
        }
    ],
...
```

The REST call will simply move the task to the **In Progress** state and add the approval message as a new comment.

Now if you call the approval endpoint, you should observe the following:

- The task is now in the **In Progress** status.
- Catalog notifications have been sent, which you can check in `notification.log`:

```
{
  "@type" : "urn:x-hp:2014:software:cloud:data_model:sx:notification",
  "displayName" : "approve request SE-6469",
  "state" : "completed",
  "id" : "b131bc6f-f289-4d91-8a07-c502aa4a8f16"
}
{
  "@type" : "urn:x-hp:2014:software:cloud:data_model:sx:notification",
  "id" : "d4cac721-15bc-43f7-87ec-6ea7d3748537",
  "remoteId" : "SE-6469",
  "displayName" : "I need 'Service Exchange Task'",
  "requestor" : "consumer",
  "state" : "approved",
  "subscription" : {
    "id" : "N/A"
  }
}
```

- A notification mail has been sent to the reporter, reporting that the request has been approved.

Support for the deny operation is similar and you can find it in the example sources.

How to create CX content (HP SM Problem entity)

Introduction and purpose

The purpose of this section is to provide a simple introduction to HP SX Case Exchange (CX) content creation based on one such existing content - the Problem CX. Find information about this content and how to install and run it in SX Problem Case Exchange Content Installation. In this section, one "operation" of the existing content is used - **the Problem items' duplication among HP SM instances** - and using this as an example the whole configuration and coding process is demonstrated.

This section expects that you know the basics of HP SX functionality and content creation (including for example the OO Flow integration into HP SX content etc.) See How to extend HP SX Content (HP SM Problem entity) as a starting point if you need more basic information.

HP SM database triggers

This example works with Problem items in HP SM, so HP SX needs to know when changes of these items occur. This is handled by using SM database triggers. In this example, one trigger (script) for Problem items **creation** is added and one for Problem **update**. Add the following triggers to the **rootcause** table (representing the Problem UI entity) in the HP SM Client application **System Definition > Tables > rootcause > Triggers**.

SX.rootcause.after.create trigger

```
lib.SX_EntityChangeV2.entityAfterAdd('id', record);
```

SX.rootcause.after.update trigger

```
lib.SX_EntityChangeV2.entityAfterUpdate('id', oldrecord, record);
```

Work will also be done with **Problem** items' **Activity Lines** (watching for changes). These are stored in a separate table **activityproblem**. Add the following triggers for this:

SX.activityproblem.before.add

```
lib.SX_EntityChangeV2.onActivityCreateOrUpdate(record);
```

SX.activityproblem.after.add

```
lib.SX_EntityChangeV2.entityAfterAdd('thenumber', record);
```

HP SM external access (Web Service Configuration)

The next step is to make all the mentioned content accessible via remote APIs (REST, SOAP).

For Problem items, the required web service is already setup in HP SM, but it needs to be modified. In the HP SM Client application go to **Tailoring > Web Services > Web Service Configuration** and search for the Object name **Problem**. Add the following entries into the table under the **Fields** tab. It must be possible to remotely access those fields in order to create copies of Problem items with such fields filled.

Field	Caption
current.phase	CurrentPhase
status	Status
description	Description
opened.by	OpenedBy
root.cause	RootCause
id	Id
brief.description	Title
expected.resolution.time	ResolutionTime
initial.impact	Impact
severity	Urgency
subcategory	Area
category	Category
affected.item	Service
assignment	AssignmentGroup
product.type	Subarea
rcStatus	rcStatus

Now the **Expressions** tab needs to be updated too. Enter the following code, which is needed when updating Problem items (or other HP SM items. By default HP SM requires that when some of the updates (like Status changes) are made, a Change is created and written as a Journal Update. This code ensures that no **Activity Line** is written in such a case - it would be excessive to add these each time the data is automatically updated remotely - but the proper **Journal Update** is entered.

Expressions tab content for Problem web service

```
cleanup($pm.activity);cleanup($rc.update);$rc.update={"external case-exchange update"}
```

case-exchange.json

Now to start creating the `case-exchange.json` configuration file.

It is assumed that the content pack build structure already exists - see How to extend HP SX Content (HP SM Problem entity) topic for how to create it.

The `case-exchange.json` config file needs to be placed in the `src/main/resources` directory of your build module. Its content must be defined in relation to the operation needed - Problem items creation (duplication) in HP SM by HP SX CX. The configuration file defines the following 3 important sections - **events**, **eventActions** and **eventGroups**.

In the **events** section reactions to specific items that are of interest are defined, and their specific statuses. In this case it is necessary to react (create a Problem copy) when the source Problem Status field (**rcStatus**) changes its value to **Pending Vendor**. To this end, add the following **events** section to the `case-exchange.json` file:

"events" section in case-exhange.json config file

```
"events": {
        "SM": {
            "problem.referringEntityStatusChanged": {
                "changeType": [ "update" ],
                "entityFilter": "OLDRECORD['rcStatus']!=NEWRECORD['rcStatus'] &&
NEWRECORD['rcStatus']=='Pending Vendor'",
                "entityType": "rootcause"
            }
    }
```

Next is the **eventActions** section. Here you define the actions to be executed when a given change occurs (as defined in previous section). In this case you need to:

- Retrieve the entity (Problem) from HP SM
- Convert it to the Canonical (Generic) Model
- Send it to OO for further processing.

The following code ensures this. You will see more about the first two operations' definitions in the `operations.json` file description section.

"eventActions" section in case-exchange.json config file

```
"eventActions": {
        "problem.referringEntityStatusChanged": [
            {
                "action": "executeOperation",
                "operationName": "retrieveProblemCX",
                "message": {}
            },
            {
                "action": "executeOperation",
                "operationName": "convertProblemToCanonicalModel",
                "message": {}
            },
            {
                "action": "executeOoFlow",
                "backendSystemType": "SM",
                "messageType": "problemCx",
                "operationName": "referringEntityStatusChanged",
                "message": {
                    "messageHeader": {
                        "optionalCXProblemNameAppendix": " problem copy"
                    }
                }
            }
        ]
    }
```

The last action required is pretty straightforward - it sends the **operationName** parameter (and others) as the OO Flow Operation discriminator to OO.

NOTE: The results from previous operations in this event actions block are sent to OO with this message too.

What is done with the message on the OO side is further examined in the **OO Flow** section.

Lastly there is an **eventGroups** section. This groups common **eventActions** from the previous section together, and is further used in the `external-systems.json` file. It is useful that the **eventActions** be grouped together by default if they belong to one way of operations. As only one operation is created for now, this configuration is simple:

"eventGroups" section in case-exchange.json config file

```
"eventGroups": {
        "problem.ReferringEntityEvents": [
            "problem.referringEntityStatusChanged"
        ]
    }
```

More information

To know more about SX Case Exchange configuration, go to the Configuration topic.

external-systems.json, Group Alias Mappings

As stated in the `case-exchange.json` description section, the `external-systems.json` configuration file contains the mapping of actual server machines to Event Groups (from the `case-exchange.json`). The **externalSystems** section describes this:

externalSystems section from external-systems.json config file

```
"externalSystems": [
        {
            "instanceType": "SM",
            "instance": "your_sm_instance_1",
            "registeredEventGroups": [ "problem.ReferringEntityEvents" ]
        }
    ]
```

As you can see, the problem.ReferringEntityEvents Event Group (the same as in `case-exchange.json`) is mapped to *'**your_sm_instance_1**'*. i.e. HP SX will be listening out for changes on the server *'your_sm_instance_1'* which conform to the problem.referringEntityStatusChanged event entityFilter, and executing operations from the problem.referringEntityStatusChanged **eventActions** part.

Next, you need to define **externalSystemAliases** in this configuration file.

externalSystemAliases section from external-systems.json config file

```
"externalSystemAliases": [
       {
             "sourceInstanceType": "SM",
             "sourceInstance": "your_sm_instance_1",
             "targetInstanceType": "SM",
             "targetInstance": "your_sm_instance_2",
             "targetAlias": "sm2directionAlias"
         }
    ]
```

As you can see, *'your_sm_instance_1'* is defined as a **source instance** and *'your_sm_instance_2'* as a **target instance,** and named with an **alias** 'sm2directionAlias'. This alias is used in the **Group Alias Mappings file**, name it for example `smGroupAliasMappings.json`:

smGroupAliasMappings.json

```
{
    "your_sm_instance_1":{
        "Application": "sm2directionAlias"
    }
}
```

So in this file, Problem entities from 'your_sm_instance_1' server have been defined, which have the **'Application' Assignment Group** mapped to 'sm2directionAlias'. This mapping is used when **converting an entity to canonical model**.

More info

To know more about SX CX configuration, see Configuration.

Converting to the Canonical Model

Before moving to the conversion template alone, the Problem Mapping file - `problem-mappings.json` - needs to be understood as it is used during the conversion to the canonical model.

problem-mappings.json file

```
{
    "entityType": {
        "Problem": "rootcause"
    },
    "Problem": {
        "rcStatus": {
            "Accepted": "Accepted",
            "Open": "Open",
            "PendingVendor": "Pending Vendor",
            "PendingCustomer": "Pending User",
            "Referred": "Deferred",
            "Rejected": "Rejected",
            "WorkInProgress": "Work In Progress",
            "Closed": "Closed"
        }
    }
}
```

The first part, the **entityType** section, is important for the canonical model conversion as it maps the UI **name** of the entity ('Problem' in this example) to the database name (here 'rootcause'.) The second part, the Problem section with the **rcStatus** subsection in this case, contains the mapping of the **Status** UI field's possible database values (represented as the **rcStatus** field in the database) to its UI values. This mapping will be used in the `convertProblemToCanonicalModelResult.ftl` template too, so view it below as a whole and then the important parts will each be explained separately.

convertProblemToCanonicalModelResult.ftl template file

Expand source

```
<#assign writeJson='com.hp.ccue.serviceExchange.adapter.freemarker.WriteJson'?new()/>
<#assign
loadConfig='com.hp.ccue.serviceExchange.adapter.freemarker.LoadConfig'?new()/>
<#assign
findKey='com.hp.ccue.serviceExchange.adapter.freemarker.FindKeyForValue'?new()/>
<#assign
findExtSystemForAlias='com.hp.ccue.serviceExchange.caseex.freemarker.FindExternalSystemF
/>
<#assign problemMapping=loadConfig(context.contentStorage,
"sm-problem-cx/problem-mappings") />
<#assign smGroupAliasMapping=loadConfig(context.configuration,
"sm/smGroupAliasMappings") />
<#escape x as x?json_string>
{
  "event": "${message.entityChange.changeReason}",
  "entity": {
    "instanceType": "${message.entityChange.instanceType}",
    "instance": "${message.entityChange.instance}",
    "entityType": "${findKey(problemMapping.entityType,
message.entityChange.entityType)}",
    "entityId": "${message.entityChange.entityId}",
    "properties": {
      <#noescape>${writeJson(message.entityChange.entity, true)}</#noescape>,
      "Description": "${message.entityChange.entity.Description?join(", ")}",
      "Status": "${findKey(problemMapping.Problem.rcStatus,
message.entityChange.entity.rcStatus)}"
    }
  },
  <#if message.entityChange.entity.AssignmentGroup?has_content &&
message.entityChange.changeReason == 'problem.referringEntityStatusChanged'>
    <#-- (A) NEW REFERENCE IS SET => RETURN NEW REFERENCE BASED ON MAPPINGS -->
      <#assign
instanceAssignmentGroups=smGroupAliasMapping[message.entityChange.instance]!"" />
      <#if instanceAssignmentGroups?has_content>
          <#assign
groupCorrespondingAlias=instanceAssignmentGroups[message.entityChange.entity.AssignmentGr
/>
      <#else>
          <#assign groupCorrespondingAlias="" />
      </#if>
      <#assign alias=findExtSystemForAlias(context.appContext,
message.entityChange.instanceType, message.entityChange.instance,
groupCorrespondingAlias)!"" />
      "linkedEntities": [{
        "instance": "${alias.targetInstance}"
        ,"instanceType": "${alias.targetInstanceType}"
        ,"entityType": "Problem"
        ,"entityId": "${message.entityChange.entity.VendorTicket!""}"
        ,"instanceAlias": "${alias.targetAlias}",
        "properties": {
            "Attachments": [],
            "Status": ""
        }
      }],
  </#if>
}
</#escape>
```

Firstly, notice the **findExtSystemForAlias, problemMapping** and **smGroupAliasMapping** "imports" in the initial **assign** section. The first one is an external function used to search for the destination HP SM server in order to place the created Problem copy on. The second two are previously mentioned mappings - `problem-mappings.json` and `smGroupAliasMappings.json` - and they are used further down in the file.

Now you can see the **message.entityChange.changeReason** input being copied into the **event** parameter. After this comes the first large important part:

"entity" object creation

```
"entity": {
    "instanceType": "${message.entityChange.instanceType}",
    "instance": "${message.entityChange.instance}",
    "entityType": "${findKey(problemMapping.entityType,
message.entityChange.entityType)}",
    "entityId": "${message.entityChange.entityId}",
    "properties": {
      <#noescape>${writeJson(message.entityChange.entity, true)}</#noescape>,
      "Description": "${message.entityChange.entity.Description?join(", ")}",
      "Status": "${findKey(problemMapping.Problem.rcStatus,
message.entityChange.entity.rcStatus)}"
    }
  },
```

The **entity** object is created (prepared to be sent to oo) with the following important fields:

- **instanceType, instance** and **entityId** that are simply copied from the incoming HP SM message.
- **entityType** is searched for in the `problemMapping` config, as the incoming UI value needs to be mapped to the outgoing database value ('Problem' to 'rootcause' in this case).

The **properties** section is populated with all the source attributes from the incoming HP SM Problem item's representation (using the `writeJson` call). The exceptions are the **Description** field (this is split into a multi-line array when read from HP SM with a remote call), and the **Status** field (mapped with the usage of `problemMapping` config.)

The next section deals with the creation of **External References** - it is called **linkedEntities.** The data is used to convert them to External References in the **OO Flow** section.

"linkedEntities" creation

```
<#if message.entityChange.entity.AssignmentGroup?has_content &&
message.entityChange.changeReason == 'problem.referringEntityStatusChanged'>
    <#-- (A) NEW REFERENCE IS SET => RETURN NEW REFERENCE BASED ON MAPPINGS -->
      <#assign
instanceAssignmentGroups=smGroupAliasMapping[message.entityChange.instance]!"" />
      <#if instanceAssignmentGroups?has_content>
          <#assign
groupCorrespondingAlias=instanceAssignmentGroups[message.entityChange.entity.AssignmentGr
/>
      <#else>
          <#assign groupCorrespondingAlias="" />
      </#if>
      <#assign alias=findExtSystemForAlias(context.appContext,
message.entityChange.instanceType, message.entityChange.instance,
groupCorrespondingAlias)!"" />

      "linkedEntities": [{
        "instance": "${alias.targetInstance}"
        ,"instanceType": "${alias.targetInstanceType}"
        ,"entityType": "Problem"
        ,"entityId": "${message.entityChange.entity.VendorTicket!""}"
        ,"instanceAlias": "${alias.targetAlias}",
        "properties": {
            "Attachments": [],
            "Status": ""
        }
      }],
...
```

Here one important object is prepared - the **alias** field.

First, look to see if the incoming message contains a filled **AssignmentGroup** field, as the target HP SM instance will be mapped according to it. Also note if the **changeReason** is `problem.referringEntityStatusChanged` as only at this moment is the mapping created.

Next, assign an `InstanceAssignmentGroups` variable (for mapping) according to `smGroupAliasMapping` and the source instance where the Problem entity was created (from which the incoming message is routed).

If such an instance mapping exists (the `instanceAssignmentGroups` variable was filled), search in the *instanceAssignmentGroups* for the concrete alias according to the **AssignmentGroup** incoming parameter. All these prepared input parameters are then sent to the findExtSystemForAlias function, which does the search for the **alias** object.

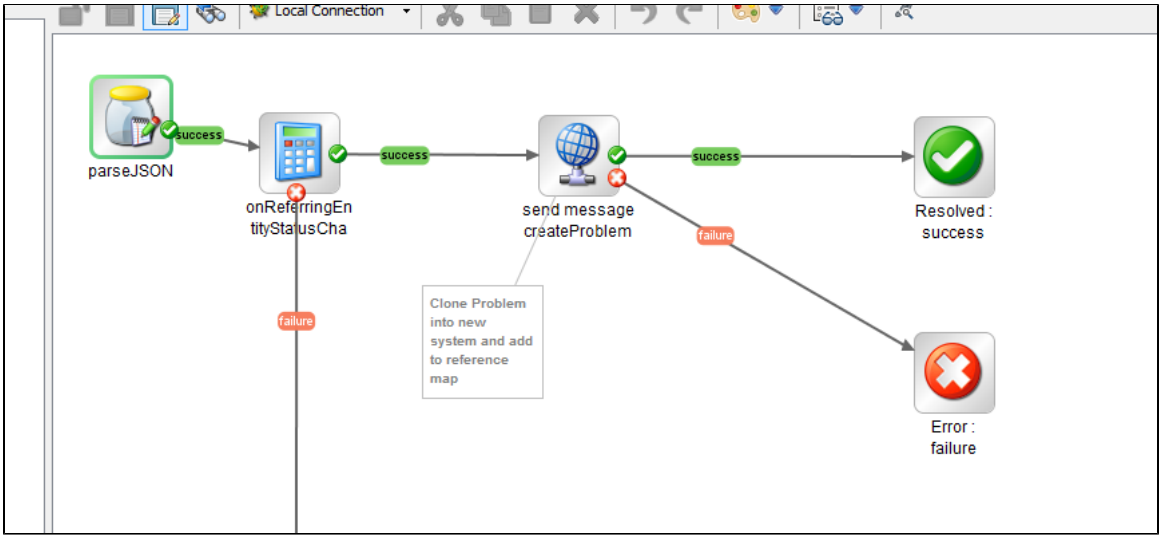
Next, the **linkedEntities** field creation is done mostly by using the objects whose population was just explained.

OO Flow

Now, creating a simple OO Flow consisting of a parameter parsing and sending a message back to HP SX for Problem item duplicate creation, will be explained.

For basic information about creating OO Flows for HP SX, see How to extend HP SX Content (HP SM Problem entity), the **OO flow** section.

To create a simple Flow like the one in this screenshot:



Start OO Studio and create a new empty Flow (Project).

First, add a `parseJSON` operation. This parses incoming JSON messages and sets the parameters needed as OO Flow input parameters. In this example the **event, entity and linkedEntity** input parameters are important, so enter them as a comma separated values list into the **propertyNamesToJson** field in the parseJSON step (Single Value - Constant Value.)

For the second step, which operation should be performed now in OO must be decided. This is decided based on the incoming **operationName** (as set in `case-exchange.json` config). This example case is waiting for the **referringEntityStatusChanged** operation. Add an 'Equal' OO Operation (located in a place like `/Base [1.1.1]/Library/Utility Operations/Math and Comparison/Simple Evaluators/Equal`) and enter the Constant value **referringEntityStatusChanged** as value2 (value1 not set), operation is **==**. This way, the **referringEntityStatusChanged** parameter existence is checked for, and if it exists, the flow moves on to the next step (the *success* branch of the flow). Otherwise, it can end in an error state for now (not pictured on the screenshot, but could for example be connected to the final error step on the right).

Now the most important step - sending a message back to SX, to do a Problem item duplication:

Add a **sendMessageToMQ** step (just like in How to extend HP SX Content (HP SM Problem entity).) As its *messageText* input, set the following code into the Constant Value field. Also ensure you have set the **operationName** Input to *'x:batch* - this provides an opportunity to call more operations in a row. It is not required immediately, but will be needed in the future. For example, if an External Reference mapping creation is called after the new Problem entity comes into existence on another HP SM server.

Create Problem opearation call

```
{
  "operations": [
    {
      "operationName": "createProblemCX",
      "message": {
        "messageHeader": {
          "backendSystemType": "${linkedEntity.instanceType}",
          "targetInstance": "${linkedEntity.instance}"
        },
               "args":{
                    "event":"${event}",
                    "entity":${entity},
                    "linkedEntity":${linkedEntity}
                },
        "name": "${entity.properties.Title} problem copy",
        "description": "${entity.properties.Description}",
        "urgency": "${entity.properties.Urgency}",
        "impact": "${entity.properties.Impact}",
        "area": "${entity.properties.Area}",
        "subarea": "${entity.properties.Subarea}",
        "assignmentGroup": "${entity.properties.AssignmentGroup}",
        "service": "${entity.properties.Service}"
      }
    }
  ]
}
```

As you can see from the code itself, the first and most important part is the **operationName** parameter, that says which SX operation should be called. For the definition of the example operation - **createProblemCX** - see the `operations.json` config description, under the subsection **Problem entity creation**.

Next a **targetInstance** and backendSystemType must be set, based on the incoming parameters in the **linkedEntity** object. See its creation in **Converting to the Canonical Model** section.

Next all three incoming objects are forwarded for further processing - **event, entity and linkedEntity**.

Lastly the Problem specific fields are set, based on the incoming **entity** object. See its creation in the Canonical Model section again, and see the usage of the created fields in the `createProblemCX` operation templates later.

This is everything needed from the OO Flow for now, so after you have created the final 'success' and 'failure' states and the appropriate transitions as pictured in the screenshot, close OO Studio.

More info

To know more about OO Flows in relation to CX functionality, please refer to OO flows.

operations.json

Problem entity retrieval

As described in the `case-exchange.json` file description section, the following operations need to be defined: Problem entity retrieval from HP SM and conversion to the Canonical Model.

Beginning with the retrieval operation:

Problem entity retrieval operation definition

```
"retrieveProblemCX": [
        {
            "label": "Retrieve SM entity details",
            "requestUrlTemplate": "problemCx/retrieveSmEntityUrl.ftl",
            "responseTemplate": "problemCx/retrieveSmEntityResponse.ftl",
            "method": "GET"
        },
        {
            "label": "Retrieve external references",
            "requestUrlTemplate": "problemCx/retrieveExternalReferencesUrl.ftl",
            "responseTemplate": "problemCx/retrieveExternalReferencesResponse.ftl",
            "method": "GET"
        }
    ]
```

It is necessary to fetch 2 objects:

- The Problem **entity itself**
- The **External References table**, which will be used to write the mapping to the created Problem on the other HP SM server instance (*'your_sm_instance_2'* in this example.)

Furthermore, the **requestUrlTemplate**, **responseTemplate** or **requestTemplate** (which is not used in our case) can be defined. The **(FTL) Templates** used are described in the next section.

To finish the current config file:

convertProblemToCanonicalModel operation definition

```
"convertProblemToCanonicalModel": [
        {
            "label": "Convert Problem to canonical model",
            "resultTemplate": "problemCx/convertProblemToCanonicalModelResult.ftl"
        }
    ]
```

Another kind of template is used here, the **resultTemplate**. This template content is described in **Converting to the Canonical Model** section.

Problem entity creation

The following code is used for the **createProblemCX** operation basic configuration:

createProblemCX operation definition

```
"createProblemCX": [
        {
            "label": "Create problem",
            "requestUrlTemplate": "smSoapUrl.ftl",
            "requestTemplate": "createProblem.ftl",
            "responseTemplate": "createProblemResponse.ftl",
            "header-SOAPAction": "Create",
            "header-Accept": "text/xml"
        }
    ]
```

A SOAP message is sent to HP SM to create a Problem entity. Most importantly the **requestUrlTemplate** (to direct into the right HP SM server host and the correct URL) and the **requestTemplate** (to send the correct data) fields are needed. The response from HP SM is parsed - in the **responseTemplate** part. Lastly, notice the **header-SOAPAction** parameter. It is needed to distinguish between the called operation on the HP SM side and the proper value found from the **Action Names** field. See the Web Service Configuration page for the appropriate HP SM object (*go to **Tailoring > Web Services ->Web Service Configuration*** and search for Object name *Problem*).

The **(FTL) Templates** used are described in the next section.

More info

To know more about the HP SX Operations usage for CX use cases, consult Operations.

Calling external systems (HP SM) and FTL templates

Problem entity retrieval

First, it is necessary to write the FTL templates needed for the Problem retrieval operation.

As explained in the previous section, this includes both the Problem item alone and the external references.

For the Problem entity first step, the URL from which the data will be obtained must be constructed. The `retrieveSmEntityUrl.ftl` file handles that.

retrieveSmEntityUrl.ftl

```
<#-- @ftlvariable name="instanceConfig" type="java.util.Map" -->
<#-- @ftlvariable name="message" type="java.util.Map" -->
<#escape x as x?url>
<#if message.entityChange.entityType == 'probsummary'><#assign
entityRestCollection='sxce_incidents'>
</#if>
<#if message.entityChange.entityType == 'rootcause'><#assign
entityRestCollection='problems'>
</#if>
<#if entityRestCollection??
><#noescape>${instanceConfig.endpoint}/9/rest/${entityRestCollection}/${message.entityCha
```

Notice that this file can be used to handle the Incident (probsummary) item too, the actual second 'if' handles the rootcause (Problem) object. The difference to the probsummary URL is in the part behind the `.../rest/...` There is a name of the retrieved objects collection stored in the *entityRestCollection* variable, **problems** in this example. The REST collection name for the object can be found in HP SM.

In the HP SM Client application go to **Tailoring > Web Services > Web Service Configuration** and search for the Object name '*Problem*'. Then go to the **Restful** tab and see the collection name in the **Resource Collection Name:** field.

Next, parse the received *result in the* `retrieveSmEntityResponse.ftl` file:

retrieveSmEntityResponse.ftl

```
<#assign writeJson='com.hp.ccue.serviceExchange.adapter.freemarker.WriteJson'?new()/>
<#escape x as x?json_string>
<#if message.entityChange.entityType == 'probsummary'><#assign
entityUiType='Incident'>
</#if>
<#if message.entityChange.entityType == 'rootcause'><#assign entityUiType='Problem'>
</#if>
<#if doc.result.ReturnCode==0 && entityUiType?? >
{
  "entityChange": {
    "entity": <#noescape>${writeJson(doc.result[entityUiType])}</#noescape>
  }
}
</#if>
</#escape>
```

Here all the input is written into the output JSON structure as the entityChange.entity object. The writeout `<#noescape>${writeJson(doc.result[entityUiType])}</#noescape>` ensures that. There is a conditional switch first, based on **message.entityChange.entityType** ('rootcause' in this case). The *entityUiType* variable ('Problem'), is set under the key where the data is stored in the incoming message (from HP SM.)

Now the External References - the retrieval code is similar to the Problem item itself, with a few minor differences:

retrieveExternalReferencesUrl.ftl

```
<#-- @ftlvariable name="instanceConfig" type="java.util.Map" -->
<#-- @ftlvariable name="message" type="java.util.Map" -->
<#escape x as x?url>
<#noescape>${instanceConfig.endpoint}</#noescape>
/9/rest/sxexternalreferences/?query=${"internalEntityType=\"" +
message.entityChange.entityType + "\" and internalEntityId=\""+
message.entityChange.entityId + "\""}&view=expand
</#escape>
```

As you may guess, the only difference in the `retrieveExternalReferencesUrl.ftl` template is the name of the retrieved objects collection - **sxexternalreferences** this time.

retrieveExternalReferencesResponse.ftl

```
<#assign writeJson='com.hp.ccue.serviceExchange.adapter.freemarker.WriteJson'?new()/>
<#escape x as x?json_string>
<#if doc.result.ReturnCode==0>
{
  "entityChange": {
    "SXExternalReferences": <#if doc.result.content??
><#noescape>${writeJson(doc.result.content)}</#noescape><#else>[]</#if>
  }
}
</#if>
</#escape>
```

Again, the code is very similar, only this time the incoming content is re-written into the **entityChange.SXExternalReferences** object. The result does not appear in this section, but see the complete source code of Problem CX in SX Problem Case Exchange Content Installation, and note the `convertProblemToCanonicalModelResult.ftl` template section.

Problem entity creation

This example use the HP SM SOAP interface for the Problem object creation. See the SOAP interface URL from `smSoapUrl.ftl`:

smSoapUrl.ftl

```
<#-- @ftlvariable name="instanceConfig" type="java.util.Map" -->
${instanceConfig.endpoint}/7/ws
```

If you point your browser to the URL: **`http://your_sm_intance:your_sm_port/SM/7/ProblemManagement.wsdl`** (specified in the **Service Name** field in the HP SM **Web Service Configuration** section), you can see the SOAP interface WSDL and all operations, object structures etc.. As you will see in `createProblem.ftl`, the **CreateProblem** operation is called:

createProblem.ftl

```
<#escape x as x?xml>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
    <Body>
        <CreateProblemRequest xmlns="http://schemas.hp.com/SM/7">
            <model>
                <keys/>
                <instance>
                    <Title>${message.name}</Title>
                    <Description>
                        <Description>${message.description}</Description>
                    </Description>
                    <Impact>${message.impact!"4"}</Impact>
                    <Urgency>${message.urgency}</Urgency>
                    <Service>${message.service!"Applications"}</Service>

<AssignmentGroup>${message.assignmentGroup!"Application"}</AssignmentGroup>
                    <Area>${message.area!"data"}</Area>
                    <Subarea>${message.subarea!"data or file incorrect"}</Subarea>
                </instance>
            </model>
        </CreateProblemRequest>
    </Body>
</Envelope>
</#escape>
```

The structure needing to be sent is defined by the wsdl, it is only necessary to fill in fields required by this example:

- Title
- Description
- Impact
- Urgency
- Service
- Assignment Group
- Area
- Subarea

How these input values came into existence can be seen from the OO flow operation call (**Create problem operation call** code block), and the c `onvertProblemToCanonicalModelResult.ftl`, which prepares the data for the OO flow.

Appendix A: Service Exchange - API

Table of Contents

	CCUE API Specification
Owner:	Ales Jerabek, Petr Fiedler
Status:	**ACTIVE**
Reviewers:	
Last COMPLETED revision:	
Last REVIEW revision:	

Revision History

Version	Date	Remarks
1.0	2014/Apr/7	Initial proposal.

Introduction

This is a proposal of the specification for **SX API**. This API design follows *REST architectural style principles* with including custom media types and HATEOAS.

Overview

Client's Entry Points

Resource	Method	URI	Description
Requests	POST	`http://[host]:[port](/[context]?)/api/request`	Create a new request to fulfill
Operations	POST	`http://[host]:[port](/[context]?)/api/operation`	Invoke operation

Callback	POST	generated	Send a notification about a request state
Tickets	POST	`http://[host]:[port](/[context]?)/api/ticket`	Create new ticket
	POST	`http://[host]:[port](/[context]?)/api/ticket/[id]/attachment`	Adds attachment into the ticket
	POST	`http://[host]:[port](/[context]?)/api/ticket/filter`	List tickets
	GET	`http://[host]:[port](/[context]?)/api/ticket/[id]/attachment`	List ticket attachments
	GET	`http://[host]:[port](/[context]?)/api/ticket/[id]/attachment/[attachmentId]`	Get ticket attachment
	GET	`http://[host]:[port](/[context]?)/api/ticket/[id]`	Ticket detail
	POST	`http://[host]:[port](/[context]?)/api/ticket/[id]/comment`	Add ticket comment
	GET	`http://[host]:[port](/[context]?)/api/ticket/property`	Ticket property descriptors
	GET	`http://[host]:[port](/[context]?)/api/ticket/property/[propertyName]`	Ticket property value enumeration
Content Packs	GET	`http://[host]:[port](/[context]?)/api/content/`	List content packs
	GET	`http://[host]:[port](/[context]?)/api/content/[id]`	Content pack detail
	DELETE	`http://[host]:[port](/[context]?)/api/content/[id]`	Delete content pack
	POST	`http://[host]:[port](/[context]?)/api/content/`	Upload content pack

Resources

Requests

The resource allows to list existing offerings and create new one.

URI	`(/[context]?)/api/request`
Methods	POST

Create Request

Request

Method	URI	Request Media Types	Response Media Types	Description
POST	`(/[context]?)/api/request`	application/json	application/json	Create a new request to fulfill

Template:

```
{
   "@type": "urn:x-hp:2014:software:cloud:data_model:sx:request",
   "messageType": ${messageType},
   "name": ${name},
   "description": ${description},
   "startDate": ${startDate},
   "endDate": ${endDate},

   "items" : [
       {
         "id" : ${itemId},
         "name" : ${itemName},
         "quantity" : ${quantity},
         "recipient": {
             "@self": ${selfToRecipient},
             "@type": "urn:x-hp:2014:software:cloud:data_model:sx:person",
             "name": ${userName}
         },
         "properties": [
             ...
         ],
         "route": {
            "@type": "urn:x-hp:2014:software:cloud:data_model:sx:route",
            "system_type": ${systemType},
            "target_instance": ${targetInstance}
         }
      }
   ]
}
```

Description:

Param	Description
messageType	message type, it is "**order**" or "**bundle**" for now.
name	order name
description	order description
startDate	subscription start date (optional)
endDate	subscription end date (optional)
itemId	identifier of ordered item, back end system must know it (CSA offering id or SM catalog item name)
itemName	human readable item name (Custom Laptop Provisioning)
quantity	quantity
selfToRecipient	uri to recipient (it is optional)
userName	name of requester, it should be user of back end system
systemType	provider type, it is "**SM**" or "**CSA**" for now
targetInstance	URI of provider instance

Single request example:

```
POST /sx/api/request HTTP/1.1
Host: example.com
Content-Type: application/json
Content-Length: NNN

{
   "@type": "urn:x-hp:2014:software:cloud:data_model:sx:request",
   "messageType": "order",
   "name": "Ordering new desktop",
   "description": "My desktop is more than 5 years old.",
   "startDate": "2014-04-09T16:00:00Z",
   "endDate": "2015-04-09T16:00:00Z",

   "items" : [
       {
         "id" : "Custom Desktop Provisioning",
         "name" : "Custom Desktop Provisioning",
         "quantity" : 1,
         "recipient": {
             "@self": "https://example.org/idm/person/432423423432",
             "@type": "urn:x-hp:2014:software:cloud:data_model:sx:person",
             "name": "HUGHES, JULIE"
         },
         "properties": [
            {
                "@type": "urn:x-hp:2014:software:cloud:data_model:property:text",
                "name": "description",
                "value": "add also optical mouse please"
            },
            {
                "@type": "urn:x-hp:2014:software:cloud:data_model:property:number",
                "name": "memory",
                "value": 16
            },
            {
                "@type": "urn:x-hp:2014:software:cloud:data_model:property:boolean",
                "name": "withMouse",
                "value": "true"
            },
            {
                "@type": "urn:x-hp:2014:software:cloud:data_model:property:select",
                "name": "model",
                "value": "model1"
            }
         ],
         "route": {
            "@type": "urn:x-hp:2014:software:cloud:data_model:sx:route",
            "system_type": "SM",
            "target_instance": "http://16.60.183.57:13080/SM"
         }
      }
   ]
}
```

Response

Http status 202 Accepted

```
Content-Type: application/json
Content-Length: NNN

{
    "id": ${requestId}
}
```

RequestId is generated by SX, all request notifications will include it as well.

Bundle Example:

```
{
    "@type":"urn:x-hp:2014:software:cloud:data_model:sx:request",
    "messageType":"bundle",
    "name" : "new engineer bundle",
    "description":"",
    "startDate":"2014-04-09T16:00:00",
    "endDate":"2015-04-09T16:00:00",
    "items":[
        {
            "id":"90b73e8d4720508f01472074b1430094",
            "name":"2 Tier Platform",
            "quantity":1,
            "recipient":{
                "@self":"",
                "@type":"urn:x-hp:2014:software:cloud:data_model:sx:person",
                "name":"consumer"
            },
            "properties":
            [
                {
                    "@type":"urn:x-hp:2014:software:cloud:data_model:property:select",
                    "name":"JBossType",
                    "value":"jboss"
                },
                {
                    "@type":"urn:x-hp:2014:software:cloud:data_model:property:select",
                    "name":"OracleDBType",
                    "value":"oracle"
                }
            ],
            "route":{
                "@type":"urn:x-hp:2014:software:cloud:data_model:sx:route",
                "system_type":"CSA",
                "target_instance":"https://mpavm0011.hpswlabs.adapps.hp.com:8444/csa"
            }
        },
        {
            "id":"iPaq",
            "name":"iPaq",
            "quantity":1,
            "recipient":
            {
                "@self":"",
```

```
    "@type":"urn:x-hp:2014:software:cloud:data_model:sx:person",
    "name":"CONSUMER"
},
"properties":
[
    {
        "@type":"urn:x-hp:2014:software:cloud:data_model:property:text",
        "name":"memory",
        "value":"128"
    },
    {
        "@type":"urn:x-hp:2014:software:cloud:data_model:property:option",
        "name":"battery",
        "value":"true"
    }
],
"route":
{
    "@type":"urn:x-hp:2014:software:cloud:data_model:sx:route",
    "system_type":"SM",
    "target_instance":"http://mpavmsm06.hpswlabs.adapps.hp.com:13080/SM"
}
```

```
        }
    ]
}
```

Response to Bundle Request

Http status 202 Accepted

```
Content-Type: application/json
Content-Length: NNN

{
    "id": ${bundleId},
 "items":[
  {
   "id": ${itemId},
   "index": ${itemIndex},
  },
  {
   "id": ${itemId},
   "index": ${itemIndex},
  }
 ]
 }
```

Both BundleId and ItemId are generated by SX, all request notifications will include them as well.

ItemIndex marks the position of the item in the request array.

Operations

Invoke operation

Request

Method	URI	Request Media Types	Response Media Types	Description
POST	`(/[context]?)/api/operation`	application/json	application/json	Invoke operation such as cancel subscription or custom operation on realized components

Template:

```
{
   "@type": "urn:x-hp:2014:software:cloud:data_model:sx:invoke",
   "entityId": ${entityId},
   "entityType": ${entityType},
   "operationName": ${operationName},
   "recipient": {
       "@self": ${selfToRecipient},
       "@type": "urn:x-hp:2014:software:cloud:data_model:sx:person",
       "name": ${userName}
   },

   "parameters": [
       {
         "@type": ${parameterType}
         "name": ${parameterName},
         "value": ${parameterValue}
       }
   ]
}
```

Description:

Param	Description
entityId	id of affected entity such as subscription id or realized component id
entityType	type of entity allowed values are **request**, **subscription**,**ticket**
operationName	name of operation to invoke e.g. cancel, approve, deny
selfToRecipient	uri to recipient (it is optional)
userName	name of approver, it should be user of back end system
parameters	optional operation parameters

Example:

```
POST /sx/api/operation HTTP/1.1
Host: example.com
Content-Type: application/json
Content-Length: NNN

{
   "@type": "urn:x-hp:2014:software:cloud:data_model:sx:invoke",
   "entityId": "287237842384523",
   "entityType": "subscription",
   "operationName": "cancel",
   "recipient": {
       "@self": "https://example.org/idm/person/432423423432",
       "@type": "urn:x-hp:2014:software:cloud:data_model:sx:person",
       "name": "joe.manager"
   },
   "parameters" : [
   ]
}
```

Response

Http status 202 Accepted

```
Content-Type: application/json
Content-Length: NNN

{
    "id": ${requestId}
}
```

RequestId is generated by SX, all request notifications will include it as well.

Supported operations and their parameters

- Request
  - approve (message)
  - deny (message)
  - closeOrder (no parameters)
- Subscription
  - cancel (no parameters)
- Component
  - operations described in service notification

Callback

We expects that calling system provides following rest endpoint to the SX, there will be one endpoint for all notifications coming to single calling system (e.g. single CCUE catalog instance).

URI	generated
Methods	POST

Request State Notification

Request

Method	URI	Request Media Types	Response Media Types	Description
POST	generated	application/json	n/a	Send a notification about a request state

Template:

```
{
   "@type": "urn:x-hp:2014:software:cloud:data_model:sx:notification",
   "id": ${requestId}
   "remoteId": ${requestRemoteId}
   "state" : ${requestState},
   "subscription": {
        "id": ${subscriptionId}
   }
}
```

Description:

Param	Description
requestId	request id
requestRemoteId	backed id of request ("human readable"). It is not mandatory.
requestState	request state, see State code bellow
subscriptionId	id of subscription, it is combination of fulfillment system identifier and id in external system separated by : when fulfillment system doesn't create subscription id is reported as **" N/A"**

State codes:

Items can be in the following states:

- **submitted**
- **pending_approval**
- **approved**
- **rejected -** (end state)
- **in_progress**
- **completed -** (end state)
- **failed -** (end state)
- **cancelled -** (end state)

Bundles can be in the following states:

- **in_progress**
- **completed -** (end state)
- **failed -** (end state)

States can come in any order, but they have semantic order defined by previous list. Consumption will create subscription if state later than pending_approval (approved ....) comes and subscription id is present (some id or "N/A" for quotas).

Example, single item request:

```
Length: NNN
{
  "@type": "urn:x-hp:2014:software:cloud:data_model:sx:notification",
  "id": "1783512783512873",
  "remoteId": "SD10396",
  "state": "completed",
  "subscription" : {
    "id" : "CSAPrague:90b73e8d456ace310145704d17142db1"
  }
}}
```

Bundle request notification:

```
Length: NNN
{
  "@type": "urn:x-hp:2014:software:cloud:data_model:sx:bundleNotification",
  "id": "1783512783512873",
  "state": "completed",
  "displayName" : "new engineer bundle"
  "items" : [
    {
      "@type": "urn:x-hp:2014:software:cloud:data_model:sx:notification",
      "id": "6466623423523523",
      "remoteId": "90cefbb74720f4c40147b5cee41b4d8b",
      "state": "completed",
      "displayName" : "new engineer bundle - item 1/2 (2 Tier Platform)"
      "subscription" : {
        "id" : "CSAPrague:90b73e8d456ace310145704d17142db1"
      }
    }
  ]
}
```

Only includes items that have had their state changed.

Pending approval notification:

```
Length: NNN
{
  "@type": "urn:x-hp:2014:software:cloud:data_model:sx:notification",
  "id": "1783512783512873",
  "state": "pending_approval",
  "approvers" : [
    {
      "name": "joe.manager"
    },
    {
      "name": "jim.manager"
    }
  ]
}}
```

Response

Http status 200 when notification was properly consumed. Anything else will cause another attempt for notification in future.

Subscription State Notification

Request

Method	URI	Request Media Types	Response Media Types	Description
POST	generated	application/json	n/a	Send a notification about a subscription state

Template:

```
{
   "@type": "urn:x-hp:2014:software:cloud:data_model:sx:subscriptionNotification",
   "id": ${subscriptionId},
   "state": ${subscriptionState},
   "components": [
        {
            "@type": ${componentType},
            "id": ${componentId},
            "displayName": ${componentDisplayName},
            "properties": [
                {
                   "@type": ${propertyType}
                   "name": ${propertyName},
                   "displayName": ${propertyDisplayName},
                   "value": ${propertyValue}
                }
            ],
            "operations": [
                {
                    "name": ${operationName},
                    "displayName": ${operationDisplayName},
                    "parameters": [
                        {
                           "@type": ${parameterType}
                           "name": ${parameterName},
                           "displayName": ${parameterDisplayName},
                           "value": ${parameterDefaultValue}
                        }
                    ]
                }
            ]
        }
   ]
}
```

Description:

Param	Description
subscriptionId	id of subscription
subscriptionState	state of subscription
componentType	realized component type e.g. server group

componentId	realized component id (usable for calling operations on it later)
componentDisplayName	realized component label
properties/parameters	realized component properties or operation parameter descriptors

State codes:

Subscription can be in the following states:

- **pending**
- **active**
- **expired -** (end state)
- **cancelled -** (end state)
- **terminated -** (end state)

Example:

```
Length: NNN
{
  "@type" : "urn:x-hp:2014:software:cloud:data_model:sx:notification",
  "state" : "active",
  "subscription" : {
    "id" : "90b73e8d456ace310145704d17142db1",
    "state" : "active",
    "components" : [ {
      "name" : "VcenterServer0001",
      "displayName" : "VcenterServer 1",
      "properties" : [ {
        "name" : "requestedMemorySize",
        "displayName" : "requestedMemorySize"
      }, {
        "name" : "vmNamePrefix",
        "displayName" : "vmNamePrefix",
        "value" : "vmmmm"
      } ]
    } ]
  }
}}
```

Response

Http status 200 when notification was properly consumed. Anything else will cause another attempt for notification in future.

Tickets

The resource allows to list existing tickets and create new one. All ticketing related calls are authenticated using IdM token passed in X-Auth-Token header.

Headers

Name	Value	Description
X-Auth-Token	JWT IdM token	token used for authorization, token is validated using shared secret key and also checked for expiration date

Create Ticket

Request

Method	URI	Request Media Types	Response Media Types	Description
POST	`(/[context]?)/api/ticket`	application/json	application/hal+json	Create a new ticket

Template:

```
{
   "@type": "urn:x-hp:2014:software:cloud:data_model:sx:ticket",
   "name": ${name},
   "description": ${description},
   "properties": [
       {
         "@type": ${propertyType}
         "name": ${propertyName},
         "value": ${propertyValue}
       }
   ]
}
```

Response

```
Content-Type: application/hal+json
Content-Length: NNN

{
    "_links": {...},
    "name": "asdasd",
    "description": "asdasdasd",
    "properties": [
    ]
}
```

Create Ticket Attachment

Request

Method	URI	Request Media Types	Response Media Types	Description
POST	`(/[context]?)/api/ticket/[id]/attachment`	depends on attachment type	application/hal+json	Adds attachment to ticket

Headers

Name	Value	Description
Content-Disposition	attachment;filename=[filename];	pass original file name which is later used as name of attachment presented to user

Response

Http status 201 (Created)

```
Content-Type: application/hal+json
Content-Length: NNN

{
    "_links": {
        "@self": {
           "href": ${attachmentUri}
        }
    },
    "id": "cid:28735123782",
    "name": "foo.jpg",
    "type": "image/jpg",
    "length": 347523
}
```

Create Ticket Comment

Request

Method	URI	Request Media Types	Response Media Types	Description
POST	`(/[context]?)/api/ticket/[id]/comment`	application/json	application/hal+json	Adds comment to ticket

```
{
    "description": "New comment text"
}
```

Response

Same as get ticket response.

List Tickets

Request

Method	URI	Request Media Types	Response Media Types	Description
POST	`(/[context]?)/api/ticket/filter`	application/json	application/json	List tickets

Query parameters

Param	Description
start-index	Index of first row returned, 1 = first row

page-size	Maximum number of rows returned

Template:

```
{
    "sort": {
       "field": ${sortField},
       "direction": ${sortDirection}
    },
    "filter": {
       "status": "${ticketStatus}",
       "nameAndDescription": "${textSearch}"
    }
}
```

Response

```
Content-Type: application/hal+json
Content-Length: NNN
{
 "_links":{
  "self":{
   "href":"/sx/api/ticket/filter"
  }
 },
 "@startIndex":1,
 "@itemsPerPage":10,
 "@totalResults":34,
 "_embedded":[
  {
   "_links":{
    "self":{
     "href":"/sx/api/ticket/SD10348"
    }
   },
   "id":"SD10348",
   "name":"My laptop is broken",
   "description":"It doesn't work at all.",
   "openTime":"2014-05-13T19:38:18+00:00",
   "updateTime":"2014-05-14T09:08:30+00:00",
   "status":"completed"
  }
 }
}
```

List Ticket Attachments

Request

Method	URI	Request Media Types	Response Media Types	Description

GET	(/[context]?)/api/ticket/[id]/attachment	N/A	application/hal+json	List ticket attachments

Response

```
Content-Type: application/hal+json
Content-Length: NNN
{
   "_links": {
       "self": {
          "href": "/sx/api/ticket/SD65466/attachment"
       }
   },
   "_embedded": [
    {
        "_links": {
           "self": {
               "href": "/sx/api/ticket/SD65466/attachment/cid:12836712468"
           }
        },
        "name": "foo.jpg",
        "id": "cid:12836712468",
        "type": "image/jpeg",
        "length": 134124
    }
  ]
}
```

Get ticket attachment

Request

Method	URI	Request Media Types	Response Media Types	Description
GET	(/[context]?)/api/ticket/[id]/attachment/[attachmentId]	N/A	attachment media type	Get ticket attachment

Response

Headers:

Content-type:

Content-disposition:

Body: attachment

Ticket Detail

Request

Method	URI	Request Media Types	Response Media Types	Description

GET	(/[context]?)/api/ticket/[id]	N/A	application/json	Ticket detail

Response

```
Content-Type: application/hal+json
Content-Length: NNN
 {
    "_links": {
       "self": {
          "href": "/sx/api/ticket/SD78364"
       },
       "attachments": {
          "href": "/sx/api/ticket/SD78364/attachment"
       },
       "createComment": {
          "href": "/sx/api/ticket/SD78364/comment"
       },
       "uploadAttachment": {
          "href": "/sx/api/ticket/SD78364/attachment"
       },
       "close": {
          "href": "/sx/api/operation"
       }
    },
    "@type": "urn:x-hp:2014:software:cloud:data_model:sx:ticket",
    "id": "SD78364",
    "name": "asdasd",
    "description": "asdasdasd",
    "properties": [
       {
        "@type": "urn:x-hp:2014:software:cloud:data_model:property:select",
        "name": "contactMethod",
        "value": "email"
       }
    ],
 "status":"completed",
 "openTime":"2007-09-06T08:06:00+00:00",
 "updateTime":"2008-09-18T17:11:06+00:00",
 "comments" : [
    {
       "id": "002136128746",
       "author": "falcon",
       "time": "2007-09-06T08:06:00+00:00",
       "description": "New information here"
    }
 ]
 }
```

Links in ticket detail

The "_link" section of ticket detail contains links to resources and available "actions". Their absence signifies that the resource/action is not available for current user. Attempts to visit such (missing) resources/actions will most likely result in 404 or 403 HTTP errors.

Link Name	Link Description
`self`	Link to itself (ticket detail)
`attachments`	Resource listing all the attachments of the ticket (see API-ListTicketAttachments)
`createComment`	Resource allowing to create new comments (see API-CreateTicketComment)
`uploadAttachment`	Resource allowing to upload new attachments (see API-CreateTicketAttachment)
`close`	Operation resource you can use to close the ticket (see API-Operations.1)

Operations

All operations returns ticket detail reflecting the state after operation finished. All ticket operation calls are synchronous.

Close Ticket

Available for tickets in all states except completed

```
POST /sx/api/operation HTTP/1.1
Host: example.com
Content-Type: application/json
Content-Length: NNN

{
   "@type": "urn:x-hp:2014:software:cloud:data_model:sx:invoke",
   "entityId": "SD78364",
   "entityType": "ticket",
   "operationName": "close",
   "parameters" : [
      {
         "@type": "urn:x-hp:2014:software:cloud:data_model:property:text",
         "name":  "description",
         "value": "I have already solved this issue."
      }
   ]
}
```

Reopen Ticket

Available for tickets in completed state

```
POST /sx/api/operation HTTP/1.1
Host: example.com
Content-Type: application/json
Content-Length: NNN

{
   "@type": "urn:x-hp:2014:software:cloud:data_model:sx:invoke",
   "entityId": "SD78364",
   "entityType": "ticket",
   "operationName": "reopen",
   "parameters" : [
      {
         "@type": "urn:x-hp:2014:software:cloud:data_model:property:text",
         "name":  "description",
         "value": "I do not agree, issue was not solved."
      }
   ]
}
```

Ticket Property Descriptors

Request

Method	URI	Request Media Types	Response Media Types	Description
GET	`(/[context]?)/api/ticket/property`	N/A	application/json	List ticket property descriptors

Additional request headers

Name	Value	Description
Accept-Language	language tag	Used to localize property's *displayName* parameters (**en** by default)

Response

```
Content-Type: application/hal+json
Content-Length: NNN

{
   "_links": "...",
   "properties": [
            {
                "@type": "urn:x-hp:2014:software:cloud:data_model:property:text",
                "name": "description",
                "displayName": "Description",
                "description": "Fill in your question",
                "length": 3000
            },
            {
                "@type": "urn:x-hp:2014:software:cloud:data_model:property:number",
                "name": "someNumber",
    "displayName": "Some Number"
            },
            {
                "@type": "urn:x-hp:2014:software:cloud:data_model:property:boolean",
                "name": "someCheckbox",
    "displayName": "Some Checkbox"
            },
            {
                "@type": "urn:x-hp:2014:software:cloud:data_model:property:select",
                "name": "someStaticList",
    "displayName": "Some Static List",
                "values": [
                   "label": "A Label"
                   "value": "aValue"
                ]
            },
            {
                "@type": "urn:x-hp:2014:software:cloud:data_model:property:select",
                "name": "someDynamicList",
    "displayName": "Some Dynamic List",
                "valuesUrl": "/api/ticket/property/someDynamicList"
            },
            {
                "@type":
"urn:x-hp:2014:software:cloud:data_model:property:select_from_many",
                "name": "someOtherDynamicList",
    "displayName": "Some Other Dynamic List",
                "searchUrl": "/api/ticket/property/someOtherDynamicList",
    "default": "default value for this field"
   },
            {
                "@type": "urn:x-hp:2014:software:cloud:data_model:property:text",
                "name": "status",
    "displayName": "Status",
                "readOnly": true
            }
     ]
}
```

Properties describing Ticket Properties

Describing Property Name	Describing Property Explanation
`@type`	Type of the property.
`name`	Internal name of the property.
`displayName`	Name of the property. This name can be displayed to users. Can be localized based on "Accept-Language" header.
`description`	Description of the property. This description can be displayed to users. Can be localized based on "Accept-Language" header.
`length`	Used with properties of the type: "text". Max. allowed length the value of this property can have.
`values`	Used with properties of the type: "select". Contains all the possible values this property can have.
`valuesUrl`	Used with properties of the type: "select". Contains URL of resource containing all the possible values this property can have (see select]).
`searchUrl`	Used with properties of the type: "select_from_many". Contains URL of resource containing possible values this property can have. Requires a query as an input (see select_from_many]).
`default`	Used with properties of the type: "select", "select_from_many". Contains default value for this property. Can be user-specific.
`readOnly`	Specifies whether this property is only used in Ticket Detail (meaning it is not to be used when creating the ticket). When "readOnly" is absent, its default value is false.
`required`	Specifies whether the user is required to fill in this property when creating ticket.
`requiredOutsidePropertiesList`	Only applies to properties with names: "name", "description". Specifies that this property is not present in the standard property array along with other properties, but is still present elsewhere in request and response messages (see https://rndwiki2.atlanta.hp.com/confluence/display/HPUI/Service+Exchange+-+API#ServiceExchange-API-CreateTicket , https://rndwiki2.atlanta.hp.com/confluence/display/HPUI/Service+Exchange+-+API#ServiceExchange-API-TicketDetail). These properties are always required.

Ticket Property Values [type: select]

Request

Method	URI	Request Media Types	Response Media Types	Description
GET	`(/[context]?)/api/ticket/property/[propertyName]`	N/A	application/json	List ticket property values (for dynamic lists)

Response

```
Content-Type: application/hal+json
Content-Length: NNN

{
   "_links": "...",
   "values": [
    {
                "label": "A",
                "value": "a"
            },
            {
                "label": "B",
                "name": "b"
            }
   ]
}
```

Ticket Property Values [type: select_from_many]

Request

Method	URI	Request Media Types	Response Media Types	Description
GET	`(/[context]?)/api/ticket/property/[propertyName]`	N/A	application/json	List ticket property values (for dynamic lists)

Query parameters

Param	Description
q	Property value label filter (only property values where label contains the query String q will be returned - case insensitive)
start-index	Index of first property value returned, 1 = first value (1 by default)
page-size	Maximum number of property values returned (10 by default)

Response

```
Content-Type: application/hal+json
Content-Length: NNN

{
   "_links": "...",
   "@pageSize": 10,
   "@startIndex": 1,
   "@totalCount": 100,
   "values": [
    {
                "label": "A",
                "value": "a"
            },
            {
                "label": "B",
                "name": "b"
            }
   ]
}
```

Example request:

```
(/[context]?)/api/ticket/property/Contact?q=con&start-index=1&page-size=2
```

Example response

```
{
  "_links":{"self":{"href":"/sx/api/ticket/property/Contact"}},
  "@pageSize":2,
  "@startIndex":1,
  "@totalCount":5,
  "values":[
    {
       "value":"CONLAN, CONNIE",
       "label":"CONLAN, CONNIE"
    },
    {
       "value":"CONROY, COLLEEN",
       "label":"CONROY, COLLEEN"
    }
  ]
}
```

Ticket Callback

We expects that ticketing micro service provides following rest endpoint to the SX.

URI	generated
Methods	POST

Ticket State Notification

Request

Method	URI	Request Media Types	Response Media Types	Description
POST	generated	application/hal+json	n/a	Send a notification about a ticket state

Example:

```
Content-Type: application/hal+json
Content-Length: NNN

 {
    "_links": {
       "self": {
          "href": "/sx/api/ticket/SD78364"
       }
    },
    "@type": "urn:x-hp:2014:software:cloud:data_model:sx:ticket",
    "id": "SD78364",
    "name": "asdasd",
    "description": "asdasdasd",
    "properties": [
       {
        "@type": "urn:x-hp:2014:software:cloud:data_model:property:select",
        "name": "contactMethod",
        "value": "email"
       }
    ],
 "status":"completed",
 "openTime":"2007-09-06T08:06:00+00:00",
 "updateTime":"2008-09-18T17:11:06+00:00"
 }
```

Description:

State codes:

Ticket can be in the following states:

- **submitted**
- **pending_approval**
- **approved**
- **rejected -** (end state)
- **in_progress**
- **completed -** (end state)
- **failed -** (end state)
- **cancelled -** (end state)

Content Packs Management

The resource allows to list existing content packs and upload new one. All calls are authenticated using IdM token passed in X-Auth-Token header.

Headers

Name	Value	Description
X-Auth-Token	JWT IdM token	token used for authorization, token is validated using shared secret key and also checked for expiration date

List Content Packs

Request

Method	URI	Request Media Types	Response Media Types	Description
GET	`(/[context]?)/api/content`	N/A	application/hal+json	List content packs

Response

```
Content-Type: application/hal+json
Content-Length: NNN
{
    "_links":{
        "self":{
            "href":"/sx/api/content"
        }
    },
    "_embedded":[
        {
            "_links":{
                "self":{
                    "href":"/sx/api/content/csa-r2f"
                }
            },
            "id":"csa-r2f",
            "name":"CSA request to fullfilment",
            "description":"",
            "adapter":"CSA",
            "version":"1.0.0",
            "features":[
                "r2f",
                "csa-r2f"
            ],
            "ooContent":{
                "name":"oo-csa-r2f-cp",
                "version":"1.2.0"
            }
        }
    ]
}
```

Content Pack Detail

Request

Method	URI	Request Media Types	Response Media Types	Description

GET	(/[context]?)/api/content/[id]	N/A	application/hal+json	Content pack detail

Response

```
Content-Type: application/hal+json
Content-Length: NNN
{
    "_links":{
        "self":{
            "href":"/sx/api/content/csa-r2f"
        }
    },
    "id":"csa-r2f",
    "name":"CSA request to fullfilment",
    "description":"",
    "adapter":"CSA",
    "version":"1.0.0",
    "features":[
        "r2f",
        "csa-r2f"
    ],
    "ooContent":{
        "name":"oo-csa-r2f-cp",
        "version":"1.2.0"
    }
}
```

Content Pack Delete

Request

Method	URI	Request Media Types	Response Media Types	Description
DELETE	(/[context]?)/api/content/[id]	N/A	N/A	delete content pack

Response

Http status 204 No Content

```
Content-Type: NNN
Content-Length: NNN
```

Content Pack Upload

Request

Method	URI	Request Media Types	Response Media Types	Description

POST	`(/[context]?)/api/content`	application/octet-stream	application/hal+json	Upload Content pack

Stream is zipped content pack archive.

Response

```
Content-Type: application/hal+json
Content-Length: NNN
{
    "_links":{
        "self":{
            "href":"/sx/api/content/csa-r2f"
        }
    },
    "id":"csa-r2f",
    "name":"CSA request to fullfilment",
    "description":"",
    "adapter":"CSA",
    "version":"1.0.0",
    "features":[
        "r2f",
        "csa-r2f"
    ],
    "ooContent":{
        "name":"oo-csa-r2f-cp",
        "version":"1.2.0"
    }
}
```

Content pack archive format

Content pack contains configurations for specific part of SX functionality. It does not contain java code.

Structure

path	is required?	description
/metadata.json	yes	File in json format. It contains basic information about SX content pack.
/oo/[cpName].jar	no	OO content pack. SX can upload it to oo server automatically. Update policy depends on sx.content.oo.upload property.
/sm/[unloadName].unl	no	Unload files exported from SM. User have to unzip the files and import it to SM manually.
/sx/flows.json	no	Mapping between operations and oo flows in json format.
/sx/operations.json	yes	Mapping between operations and freemarker templates in json format.

/sx/templates/[freemarkertemplateName].ftl	no	Freemarker templates transform SX canonical format to external systems formats (SM, CSA, ...)
/sx/[configName].json	no	More json configurations for specific part of SX functionality.

Metadata example

```
{
    "id": "sm-r2f-change",
    "name": "SM request to fullfilment by change",
    "description": "",
    "version": "1.0.0",
    "adapter": "SM",
    "features":[
        "r2f",
        "sm-r2f",
        "sm-r2f-change"
    ]
}
```

Metadata structure

property	description
id	id must be unique
name	content pack name
description	content pack description
version	content pack version
adapter	Adapter specifics which external system the CP uses. Now SX contains CSA and SM adapters.
features	Features which CP provides. SX uses this information for enabling/disabling its REST APIs.

Appendix B: Operation executors

Operation executors

`com.hp.ccue.serviceExchange.operation.OperationExecutor` interprets `operations.json`. Here is an example `operations.json` content:

```
{
    "getCatalogItems": [
        {
            "label": "Get catalog items - changes",
            "requestUrlTemplate": "smSoapUrl.ftl",
            "requestTemplate": "retrieveCatalogItemsChanges.ftl",
            "responseTemplate": "retrieveCatalogItemsChangesResponse.ftl",
            "header-SOAPAction": "RetrieveKeysList",
            "header-Accept": "text/xml"
        },
        {
            "label": "Get catalog items - changes",
            "requestUrlTemplate": "smSoapUrl.ftl",
            "requestTemplate": "retrieveCatalogItemsChanges2.ftl",
            "responseTemplate": "retrieveCatalogItemsChangesResponse.ftl",
            "header-SOAPAction": "RetrieveKeysList",
            "header-Accept": "text/xml"
        },
        {
            "label": "Get catalog items - quotes",
            "requestUrlTemplate": "smSoapUrl.ftl",
            "requestTemplate": "retrieveCatalogItemsQuotes.ftl",
            "responseTemplate": "retrieveCatalogItemsQuotesResponse.ftl",
            "header-SOAPAction": "RetrieveKeysList",
            "header-Accept": "text/xml"
        }
    ],
    "createOrder": [
        {
            "label": "Create cart",
            "requestUrlTemplate": "smSoapUrl.ftl",
            "requestTemplate": "createCart.ftl",
            "responseTemplate": "createCartResponse.ftl",
            "header-SOAPAction": "Create",
            "header-Accept": "text/xml"
        },
    ...
}
```

More about the `operations.json` file format can be found in HOWTO Sample SX Content. As you can see, an operation is named and it consists of steps. The operation executor is able to *interpret* this file and execute operation/steps one by one. It has a *simple* method for operation execution and an *introspective* method for determining whether the operation is recognized:

```
public interface OperationExecutor {
    /**
     * Executes named operation with the given execution context.
     * The execution context is to be used during backend-specific step execution.
     * It is not intended to hold data values - data values are to be put
     * into message.
     *
     * @param operationName operation to be executed
     * @param message message to be processed and returned
     * @param context execution context (may be null)
     *
     * @return processed message
     */
    public Map<String, Object> executeOperation(String operationName, Map<String,
Object> message,
            Map<String, Object> context);


    /**
     * @param operationName name of the operation to recognize
     * @return true if the operation is recognized (and can be executed), false
otherwise
     */
    public boolean isOperationRecognized(String operationName);


}
```

The subject for the operation is always the *JSON message,* and optionally the *Java context*.

BaseOperationExecutor basics

The default implementation of OperationExecutor is in `com.hp.ccue.serviceExchange.operation.BaseOperationExecutor`.

BaseOperationExecutor can interpret these step types:

- **SubmitHttpRequest step**: used for executing an HTTP request, while performing FreeMarker transformations when creating inputs and processing outputs of the HTTP request.
    - BaseOperationExecutor recognizes a step as a SubmitHttpRequest step if the requestUrlTemplate step property is supplied.
- **SetupNotifications step**: used for setting up listening for changes of an entity in the external system.
    - BaseOperationExecutor recognizes a step as a SetupNotifications step if it is not recognized as a SubmitHttpRequest step and the notifyTemplate step property is supplied.
- **PerformFtlTransformation step**: used for performing a FreeMarker transformation only.
    - BaseOperationExecutor recognizes a step as a PerformFtlTransformation step if it is not recognized as a SubmitHttpRequest or a SetupNotifications step. and the step resultTemplate property is supplied.
- **custom step**: used for subclass-specific steps.
    - BaseOperationExecutor recognizes a step as a custom step if is not recognized as a step of the three above types.

*NOTE: Steps of all types can be **executed repeatedly** if set up using the inputSelector and inputName step properties.*

When implementing an SX adapter, you can either use this default implementation of you can extend it. The BaseOperationExecutor provides the final implementation of the **executeOperation** method of the OperationExecutor interface. Its constructor needs:

- *adapter type* (String, e.g. SM, CSA,...)
- instances.json config path (because this file is read and passed to particular steps as part of an FTL data model.)

It has the following overridable methods to customize its behavior:

- **addStepDecorator**: Any step can be decorated via **StepDecorator** (add/remove properties) before being executed, whether the step is to be decorated or not is determined by **StepFilter.**
- **beforeExecuteOperation**: Invoked before the actual execution of operation steps. Override this method if you need a pre-invocation hook. Default implementation is empty.
- **afterExecuteOperation**: Invoked after the actual execution of operation steps. Override this method if you need a post-invocation hook. Default implementation is empty.
- **finallyAfterExecuteOperation**: Invoked in the final clause of a try-finally block surrounding the execution of operation steps (including beforeExecuteOperation and afterExecuteOperation invocations.) Exceptions thrown by this method are not propagated, they are only logged. Override this method if you need a tear-down hook. Default implementation is empty.
- **executeNotificationSetup**: Invoked when executing the SetupNotifications notification step - see below for the SetupNotifications step type. You will need to override this method so that your operation executor supports the SetupNotifications step type. Default implementation of this method throws an UnsupportedOperationException.
- **executeCustomStep**: Invoked to execute a subclass-specific operation step. You will need to override this method for your operation executor to support steps of a custom type. Default implementation of this method throws an UnsupportedOperationException.
- Others (see JavaDoc): **setDefaultHttpRequestContentType, skipLoggingForOperation, beforeHttpRequestSubmit, checkProcessingError, handleAuthentication, afterHttpResponseReceived, recognizeResponseError, getDetailErrorMessage, isResponseSuccess, createDefaultFtlDatamodel**.

Overview of step properties recognized by BaseOperationExecutor

Properties common for all step types

Property	Description
label	Step display name used for logging purposes
inputSelector	Allows iteration over items in input message. Contains JSONPath expression. Current step is invoked for each item separately
inputName	Specifies input key name during input selector iteration. Currently processed item is available with this key in the input message

Properties for SubmitHttpRequest step type

Property	Description
requestUrlTemplate	FreeMarker template for request URL
requestContentSelector	Selects request body using JSONPath directly from the input message
requestHeaderTemplate	FreeMarker template for JSON object containing additional request headers
requestTemplate	FreeMarker template for request body
method	Http method used for REST requests (GET/POST/PUT/DELETE), defaults to POST for operations with header-SOAPAction, GET otherwise
header-*	Arbitrary HTTP header (header-SOAPAction, header-Content-Type,...)
responseTemplate	FreeMarker template for response transformation
useIntegrationAccount	Boolean flag indicating whether to execute the request under an integration account

Properties for SetupNotifications step type

Property	Description
idSelector	JSON path expression returning ID of the given entity
operationName	Name of check operation invoked when entity is changed in external system
notifyTemplate	FreeMarker template for input message for the check operation
callbackTemplate	FreeMarker template for catalog notification
firstRunImmediately	Boolean flag saying if the first notification is invoked at registration time

Properties for PerformFtlTransformation step type

Property	Description
resultTemplate	FreeMarker template for the transformation

More about BaseOperationExecutor step types

Notes applying to more step types

Message merging

The PerformFtlTransformation step and optionally also SubmitHttpRequest modify the input message by merging a result map into it. The merging is performed by doing the following for every key of the result map:

- If the key is contained in the message and both values are maps then the maps are merged using the same algorithm as the message and the result map
- If the key is contained in the message and both values are lists then the list from the result map is appended to the list from the message
- Otherwise, the value from the result map is simply put into the message (overwriting the original value if present.)

Here is an example:

```
{
    "owner": "paul",
    "approvers": ["john", "marry"],
    "properties": {
        "width": 10,
        "height": 20
    }
}
```

Assuming the following map results from the transformation in the PerformFtlTransformation step:

```
{
    "owner": "bob",
    "approvers": ["fred"],
    "properties": {
        "height": 25,
        "depth": 30
    }
}
```

The message will look like this:

```
{
    "owner": "bob",
    "approvers": ["john", "marry", "fred"],
    "properties": {
        "width": 10,
        "height": 25,
        "depth": 30
    }
}
```

Repeated step execution

If you need to iterate over an array and execute a step repeatedly for each item, use the inputSelector and inputName step properties. The inputSelector property contains a JSONPath to the array to iterate over. The executor will look up the array, and for each item, it will put the item under the key specified by inputName to the message, perform one step execution, and remove the key from the message. Here is an example:

```
{
    "orderInfo": {
        "approvers": ["john", "mary"]
    }
}
```

For an initial message and inputSelector=$.orderInfo.approvers and inputName=approver, then the given step will be executed twice, once against the following message:

```
{
    "orderInfo": {
        "approvers": ["john", "mary"]
    },
    "approver": "john"
}
```

And once against this message:

```
{
    "orderInfo": {
        "approvers": ["john", "mary"]
    },
    "approver": "mary"
}
```

FTL transformation

All built-in step types perform FreeMarker transformations. The data model (unless stated otherwise) contains the following default keys:

- message: the message (Map<String, Object>) passed to the step
- context: the context object (Map<String, Object>) passed to the step
- instanceConfig: instance configuration (Map<String, Object>) of the external system passed to the step
- infrastructureConfig: infrastructure configuration (Map<String, Object>)
- bundle: a resource bundle as copied from the context.bundle (is missing if missing in the context)

The context contains the following keys:

- targetInstance: instance name (String) of the external system
- configuration: Configuration Spring bean
- contentStorage: ContentStorageApi Spring bean

You can also override the beforeOperationExecution() method to add additional keys to the context object.

Here is an example of a template for request URL that can be used in a SubmitHttpRequest step:

```
${instanceConfig.endpoint}/api/mpp/mpp-request/${message.requestId?url}
```

PerformFtlTransformation step type

This step type has a single property resultTemplate that refers to a FreeMarker template which is assumed to produce a JSON document. The step retrieves the template, performs the transformation against a dataModel with the default keys, converts the result to a Java map, and merges the result into the input message.

SubmitHttpRequest step type

This step leverages a generic HTTP client (`com.hp.ccue.serviceExchange.http.HttpClient` - JavaDoc link). You can submit REST/SOAP requests by using this step. The step is identified as SubmitHttpRequest if it uses the **requestUrlTemplate** property. In order to submit an HTTP request one needs to know at least the:

- URL
- method: defaults to POST for steps with header-SOAPAction property, otherwise GET.

A request might also have a body. The body can be created via **requestTemplate** or **requestContentSelector**. HTTP headers including Content-Type can be specified via:

- **header-*** step property, all fields starting with this prefix are put into the request as HTTP headers
- **requestHeaderTemplate:** generated map is put into the request as headers
- **StepDecorator**: a generic advanced concept for setting up step-defaults (JavaDoc link.)

Content-Type in particular can also be specified via setDefaultHttpRequestContentType(String mimeType), for example if you know that your backend system communicates almost exclusively via JSON. After the request body and headers are built, BaseOperationExecutor calls:

- *handleAuthentication* (JavaDoc link)
- *beforeHttpRequestSubmit* (JavaDoc link.)

After the request is submitted and the response is received back, the following methods are called:

- *afterHttpResponseReceived* (JavaDoc link)
- *recognizeResponseError* (JavaDoc link) which calls:
    - *isResponseSuccess* (JavaDoc link)
    - if there is an error, it also calls *getDetailErrorMessage* (JavaDoc link.)

The response is then transformed via the **responseTemplate** if specified. The result of the transformation is merged back to the message. If this template is not specified, the HTTP response is merged back into the message as follows:

- "result": response body
    - if the response has no body, this key is missing
    - if the body is JSON it is automatically converted to map
    - if it is XML it is converted to `freemarker.ext.dom.NodeModel`
    - if it is a zero-length string or a whitespace-only string, the value is null
    - otherwise the response is put there as is - as one long string
- "resultHeaders": result HTTP headers, this key is always present.

*NOTE: the **responseTemplate** gets exactly the same input except that it is nested under the "doc" key in the incoming data model.*

SetupNotifications step type

This step is used to set up listening for changes of an entity in an external system. First it extracts the ID of the entity to listen to by resolving the JSONPath expression in the idSelector step property against the input message. Then it retrieves the FreeMarker template referred to by the notifyTemplate property, performs the transformation against a dataModel with the default keys, and converts the result to a Java map. Finally it executes the overridable method.

```
void executeNotificationSetup(
            String entityId, String checkOperation, Map<String, Object>
checkInputMessage,
            String catalogCallbackTemplate, EntityRegistrationMode mode,
            Map<String, Object> context, Map<String, Object> stepConfig)
```

This is done using the following actual parameters:

- entityId: the extracted entity id
- checkOperation: value of operationName step property
- checkOperationInputMessage: map resulting from notifyTemplate transformation
- catalogCallbackTemplate: value of callbackTemplate step property
- mode: REGISTER_AND_NOTIFY_IMMEDIATELY if the firstRunImmediately step property is set to true, and REGISTER_ONLY otherwise
- context: context object passed to the step
- stepConfig: map of step properties.

The semantics of the executeNotificationSetup method is to make sure that subsequent changes of the given entity in the external system will result in the check operation being executed (with checkOperationInputMessage as input message), and in a catalog notification subsequently being sent. The catalog notification message will be produced by performing a FreeMarker transformation with a template referred to by the catalogCallbackTemplate argument against a data model, with the message coming from the check operation under the key "message". Additionally, if mode is equal to REGISTER_AND_NOTIFY_IMMEDIATELY, a first notification is performed immediately.

Appendix C: Ticket management operations messages

Overview

This is the documentation of message formats passed to and returned from ticket management operations. These messages are based on requests to /ticket and /operation resource of SX REST API. Rest resource classes modify requests into messages documented here.

Use this documentation when implementing ticketing operations ftls, as a reference on the formats of messages that will be passed in and the format that you need to return.

Generally all ticket resource requests get the following message header before being passed to operation execution:

```
"messageHeader" : {
    "backendSystemType" : "${backendSystemType}",
    "userId" : "${userId}",
    "targetInstance" : "${targetInstanceName}"
  }
```

Your ftl transformations also have access to operation execution context. For details see Appendix B: Operation executors.

Operations doc

createTicket

Input

```
{
  "description" : "${ticketName}",
  "name" : "${ticketDescription}",
  "properties" : [ {
    "@type" : "urn:x-hp:2014:software:cloud:data_model:property:text",
    "name" : "name",
    "value" : "${ticketName}"
  }, {
    "@type" : "urn:x-hp:2014:software:cloud:data_model:property:text",
    "name" : "description",
    "value" : "${ticketDescription}"
  },
...
 ],
  "messageHeader" : {
    "backendSystemType" : "${backendSystemType}",
    "userId" : "${userId}",
    "targetInstance" : "${targetInstanceName}"
  },
  "recipient" : {
    "name" : "${userId}"
  }
}
```

The list of properties corresponds to the properties defined by the listTicktеProperties operation.

Output

Return the ticket created, see retriveTicket operation response.

NOTE: `createTicket` is usualy implemented as a two phase operation where the second phase is the same as the `retrieveTicket` operation.

retrieveTicket

Input

```
{
  "id" : "${ticketId}",
  "messageHeader" : {
    "backendSystemType" : "${backendSystemType}",
    "userId" : "${userId}",
    "targetInstance" : "${targetInstanceName}"
  },
  "recipient" : {
    "name" : "${userId}"
  }
}
```

Response

```
{
    "result": {
  "_links" : {
      "self" : {
         "href" : "/sx/api/ticket/${ticketId}"
      }
    },
    "id" : "${ticketId}",
    "name" : "${ticketName}",
    "description" : "${ticketDescription}",
    "openTime" : "${openTime}",
    "updateTime" : "${updateTime}",
    "status" : "${status}",
  "properties" : [ {
      "@type" : "${propertyType}",
      "name" : "exampleProp",
      "value" : "examplePropValue"
     },
  ...
   ],
     "comments" : [ {
      "id" : "${commentId}",
      "author" : "${commentAuthorId}",
      "time" : "${commentTime}",
      "description" : "${commentText}"
     }]
 }
}
```

Placeholder	Possible values
status	submitted, in_progress, completed

For ${propertyType}. See `listTicketProperties`.

listTickets

Input template

```
{
  "sort" : {
    "field" : "${sortField}",
    "direction" : "${sortDirection}"
  },
  "filter" : {
    "nameAndDescription" : "${textToFilterNameAndDecription}"
  },
  "startIndex" : "${startIndex}",
  "pageSize" : "${pageSize}",
  "messageHeader" : {
    "backendSystemType" : "${backendSystemType}",
    "userId" : "${userId}",
    "targetInstance" : "${targetInstanceName}"
  },
  "recipient" : {
    "name" : "${recipientName}"
  }
}
```

Placeholder	Possible values
sortField	name, openTime, updateTime
sortDirection	ascending, descending

Output

```
{
  "_links" : {
    "self" : {
      "href" : "/sx/api/ticket/filter"
    }
  },
  "@startIndex" : ${startIndex},
  "@itemsPerPage" : ${pageSize},
  "@totalResults" : ${count},
  "_embedded" : [ {
    "_links" : {
      "self" : {
        "href" : "/sx/api/ticket/${ticketId}"
      }
    },
    "id" : "${ticketId}",
 "name" : "${ticketName}",
 "description" : "${ticketDescription}",
 "openTime" : "${openTime}",
 "updateTime" : "${updateTime}",
 "status" : "${status}",
  },
  ...
]
}
```

listTicketProperties

Input

Message header. No input needed as the ticket properties are invariant for an adapter.

Output

Returns the definition of properties that you want to make visible to the portal user. The supported property types and their descriptors are described in SX API DOC. It is repeated here for convenience in the example output.

```
{
  "result" : {
    "@self" : "/sx/api/ticket/property",
    "properties" : [
      {
      "@type": "urn:x-hp:2014:software:cloud:data_model:property:text",
                 "name": "description",
                 "displayName": "Description",
                 "description": "Fill in your question",
                 "length": 3000
             },
             {
                 "@type": "urn:x-hp:2014:software:cloud:data_model:property:number",
                 "name": "someNumber",
                 "displayName": "Some Number"
             },
             {
                 "@type": "urn:x-hp:2014:software:cloud:data_model:property:boolean",
                 "name": "someCheckbox",
```

```
                "displayName": "Some Checkbox"
            },
            {
                "@type": "urn:x-hp:2014:software:cloud:data_model:property:select",
                "name": "someStaticList",
                "displayName": "Some Static List",
                "values": [
                    "label": "A Label"
                    "value": "aValue"
                 ]
            },
            {
                "@type": "urn:x-hp:2014:software:cloud:data_model:property:select",
                "name": "someDynamicList",
                "displayName": "Some Dynamic List",
                "valuesUrl": "/api/ticket/property/someDynamicList"
            },
            {
                "@type":
"urn:x-hp:2014:software:cloud:data_model:property:select_from_many",
                "name": "someOtherDynamicList",
                "displayName": "Some Other Dynamic List",
                "searchUrl": "/api/ticket/property/someOtherDynamicList",
                "default": "default value for this field"
            },
            {
                "@type": "urn:x-hp:2014:software:cloud:data_model:property:text",
                "name": "status",
                "displayName": "Status",
                "readOnly": true
            }
    ]
  },
  "messageHeader" : {
    "backendSystemType" : "${backendSystemType}",
    "userId" : "${userId}",
    "targetInstance" : "${targetInstanceName}"
  },
  "recipient" : {
```

```
    "name" : "${userId}"
  }
}
```

ticketProperty-${propertyName} [type: select]

Input

```
{
  "acceptLanguage" : "${acceptLanguageHeader}",
  "messageHeader" : {
    "backendSystemType" : "${backendSystemType}",
    "userId" : "${userId}",
    "targetInstance" : "${targetInstanceName}"
  },
  "recipient" : {
    "name" : "${userId}"
  }
}
```

Output

```
{
  "_links" : {
    "self" : {
      "href" : "/sx/api/ticket/property/${propertyName}"
    }
  },
  "values" : [ {
    "value" : "value1",
    "label" : "label1"
  },
...
]
}
```

ticketProperty-${propertyName} [type: select_from_many]

Input

```
{
  "startIndex" : "${startIndex}",
  "pageSize" : "${pageSize}",
  "q": "${q}",
  "acceptLanguage" : "${acceptLanguageHeader}",
  "messageHeader" : {
    "backendSystemType" : "${backendSystemType}",
    "userId" : "${userId}",
    "targetInstance" : "${targetInstanceName}"
  },
  "recipient" : {
    "name" : "${userId}"
  }
}
```

Output

```
{
   "_links" : {
    "self" : {
      "href" : "/sx/api/ticket/property/${propertyName}"
    }
  },
   "@pageSize": ${pageSize},
   "@startIndex": ${startIndex},
   "@totalCount": ${totalCount},
   "values": [
      {
                "label": "A",
                "value": "a"
            },
            {
                "label": "B",
                "name": "b"
            }
 ...
   ]
}
```

listTicketAttachements

Input

```
{
  "id" : "${id}",
  "messageHeader" : {
    "backendSystemType" : "${backendSystemType}",
    "userId" : "${userId}",
    "targetInstance" : "${targetInstanceName}"
  },
  "recipient" : {
    "name" : "${userId}"
  }
}
```

Output

```
"result" : {
 {
   "_links" : {
     "self" : {
        "href" : "/sx/api/ticket/${id}/attachment"
     }
   },
   "_embedded" : [
   {
      "_links" : {
         "self" : {
           "href" : "/sx/api/ticket/${ticketId}/attachment/${attachmentId}"
         }
      },
      "id" : "${attachmentId}",
      "length" : ${length},
      "name" : "${attachmentName}",
      "type" : "${attachmentMediaType}"
   },
 ...
  ]
 }
}
```

createTicketAttachment

Input

```
{
  "content" : <byte array>,
  "id" : "${ticketId}",
  "fileName" : "${fileName}",
  "messageHeader" : {
    "backendSystemType" : "${backendSystemType}",
    "userId" : "${userId}",
    "targetInstance" : "${targetInstanceName}"
  },
  "contentType" : "${contentType}",
  "recipient" : {
    "name" : "${userId}"
  }
}
```

Content is filled by the operation executor. See Appendix B: Operation executors for details about fileUpload requests configuration.

Output

```
"result" : {
    "_links" : {
      "self" : {
        "href" : "/sx/api/ticket/${ticketId}/attachment/${attachmentId}"
      }
    },
    "id" : "${attachmentId}",
    "length" : ${length},
    "name" : "${attachmentName}",
    "type" : "${attachmentMediaType}"
  }
}
```

retrieveTicketAttachment

Input

```
{
  "id" : "${id}",
  "attachmentId" : "${attachmentId}",
  "messageHeader" : {
    "backendSystemType" : "${backendSystemType}",
    "userId" : "${userId}",
    "targetInstance" : "${targetInstanceName}"
  },
  "recipient" : {
    "name" : "${userId}"
  }
}
```

Output

No response is needed, it returns the file itself.

createTicketComment

Input

```
 {
  "id" : "${ticketId}",
  "datestamp" : "${timeCommentAdded}"
  "description" : "${commentText}",
  "messageHeader" : {
    "backendSystemType" : "${backendSystemType}",
    "userId" : "${userId}",
    "targetInstance" : "${targetInstanceName}"
  },
  "recipient" : {
    "name" : "${userId}"
  }
}
```

Output

No need to return anything.

closeTicket

`closeTicket` is invoked as a general operation on the /operation resource of SX REST API and therefore the input message contains the operation identification as well.

Input

```
 {
  "@type" : "urn:x-hp:2014:software:cloud:data_model:sx:invoke",
  "entityId" : "${ticketId}",
  "entityType" : "ticket",
  "operationName" : "close",
  "id" : "${ticketId}",
  "datestamp" : "${timeClosed}",
  "description" : "${closingComment}"
  "parameters" : [ {
    "@type" : "urn:x-hp:2014:software:cloud:data_model:property:text",
    "name" : "description",
    "value" : "${closingComment}"
  } ],
  "messageHeader" : {
    "backendSystemType" : "${backendSystemType}",
    "userId" : "${userId}",
    "targetInstance" : "${targetInstanceName}"
  },
  "recipient" : {
    "name" : "${userId}"
  }

}
```

Output

No need to return anything.

Appendix D: Per instance operation definition

Writing custom operations.json files for specific instances

HP SX allows you to change the behavior of any operation for specific instances by overriding operations. You can do this by creating custom `operations.json` files. Names of these files need to follow the format: `operations-{instanceName}.json`, where the *instanceName* is the name of an instance specified in `instances.json`. Any operation defined in this file will override an operation of the same name from a generic `operations.json` file.

NOTE: The content of a custom `operations.json` file has exactly the same format as the generic one, and all the FTL files have to be in the same content pack as the custom `operations.json` file.

The custom file should be in the same directory inside the content pack where the generic file would generaly be, but it does not need to be in exactly the same content pack as the generic file that the custom file is trying to replace.

Example:

If your `instances.json` for HP SM looks like this:

```
{
    "SMInstance01": {
        "endpoint": "https://sm01.example.com:13080/SM",
        "user": {
            "loginName": "admin",
            "password": "password"
        }
    }

}
```

And you are using this HP SM instance for managing tickets (using the standard HP SM Ticketing SX content pack), then you can alter the way HP SX creates comments for tickets in this HP SM instance just by creating the file `operations-SMInstance01.json` in a new content pack, containing an override of operation **createTicketComment**, like this:

```
{
"createTicketComment":[

        {

            "label":"Create Custom Comment",

            "requestUrlTemplate": "customSmSoapUrl.ftl",

            "requestTemplate":"customCreateComment.ftl",

            "header-SOAPAction":"Create",

            "header-Accept": "text/xml"

        }

    ]
}
```