IPF APIs

CSM Reachability

Settings Management API

Service API

Credit Transfer API

Clear and Settle API

Payment Initiation Api

Payment Status Api

Connector and Transaction Operations Api

Code Generation and use

Settings Management API

Credit Transfer API

Summary

Description

This API is used to execute credit transfer between a payment engine and a CSM Service.

Integration Pattern

Asynchronous

Reply Method

Separate request/response queues/topics

Integration Method

JMS, Kafka

Message Format

JSON

Message Model

ISO20022

Security

TLS 1.2

Resiliency Strategies

Circuit Breaker, Backpressure, Automatic Retries

Models

ExecutePaymentRequest

Version:

pacs.008.001.08

Details:

View the json schema

ExecutePaymentResponse

Version:

pacs.002.001.10

Details:

View the json schema

ReceivePaymentRequest

Version:

pacs.008.001.08

Details:

View the json schema

ReceivePaymentResponse

Version:

pacs.002.001.10

Details:

View the json schema

ReceivePaymentSettledRequest

Version:

pacs.002.001.10

Details:

View the json schema

ReceivePaymentSettledResponse

Version:

N/A

Details:

View the json schema

ReceivePaymentStatusInquiryRequest

Version:

pacs.028.001.03

Details:

View the json schema

ReceivePaymentStatusInquiryResponse

Version:

pacs.002.001.10

Details:

View the json schema

ReceivePaymentTimeout

Version:

pacs.002.001.10

Details:

View the json schema

TechnicalResponse

Version:

pacs.002.001.10

Details:

View the json schema

Service API

Clear and Settle API

Clear and Settle API

Summary

Description

This API is used to exchange settlement information and confirmation between a payment engine and a CSM Service.

Integration Pattern

Asynchronous

Reply Method

Separate request/response queues/topics

Integration Method

JMS, Kafka

Message Format

JSON

Message Model

ISO20022

Security

TLS 1.2

Resiliency Strategies

Circuit Breaker, Backpressure, Automatic Retries

Models

ClearAndSettleRequest

Version:

pacs.008.001.08

Details:

View the json schema

ClearAndSettleResponse

Version:

pacs.002.001.10

Details:

View the json schema

CreditorRequestToPayRequest

Version:

pain.013.001.07

Details:

View the json schema

CreditorRequestToPayResponse

Version:

pain.014.001.07

Details:

View the json schema

DebtorRequestToPayRequest

Version:

pain.014.001.07

Details:

View the json schema

DebtorRequestToPayResponse

Version:

pain.013.001.07

Details:

View the json schema

TechnicalResponse

Version:

pacs.002.001.10

Details:

View the json schema

ValidateSchemeRulesRequest

Version:

pacs.008.001.08

Details:

View the json schema

ValidateSchemeRulesResponse

Version:

N/A

Details:

View the json schema

ValidationFailure

Version:

N/A

Details:

View the json schema

Credit Transfer API

Payment Initiation Api

Payment Initiation Api

Summary

Description

This API is used to initiate payment between a payment engine and a CSM Service.

Integration Pattern

Asynchronous

Reply Method

Separate request/response queues/topics

Integration Method

JMS, Kafka

Message Format

JSON

Message Model

ISO20022

Security

TLS 1.2

Resiliency Strategies

Circuit Breaker, Backpressure, Automatic Retries

Models

PaymentInitiationRequest

Version:

pain.001.001.09

Details:

View the json schema

PaymentInitiationResponse

Version:

pain.002.001.10

Details:

View the json schema

TechnicalResponse

Version:

pacs.002.001.10

Details:

View the json schema

Clear and Settle API

Payment Status Api

Payment Status Api

Summary

Description

This API is used to monitor/query status of the payment between a payment engine and a CSM Service.

Integration Pattern

Asynchronous

Reply Method

Separate request/response queues/topics

Integration Method

JMS, Kafka

Message Format

JSON

Message Model

ISO20022

Security

TLS 1.2

Resiliency Strategies

Circuit Breaker, Backpressure, Automatic Retries

Models

StatusRequest

Version:

pacs.028.001.03

Details:

View the json schema

StatusRequestResponse

Version:

pacs.002.001.10

Details:

View the json schema

TechnicalResponse

Version:

pacs.002.001.10

Details:

View the json schema

Payment Initiation Api

Connector and Transaction Operations Api

Connector and Transaction Operations

View the json schema

1. Endpoints

1.1. ConnectorOperationsController

1.1.1. getConnector

GET /connectors/{connectorName}

Description

Parameters

Path Parameters

Name

Description

Required

Default

Pattern

connectorName

X

null

Return Type

ConnectorInfo

Content Type

application/json

Responses

http response codes

Code

Message

Datatype

200

OK

ConnectorInfo

Samples

1.1.2. getConnectors

GET /connectors

Description

Parameters

Return Type

-

Responses

http response codes

Code

Message

Datatype

200

OK

<<>>

Samples

1.1.3. start

POST /connectors/{connectorName}/start

Description

Parameters

Path Parameters

Name

Description

Required

Default

Pattern

connectorName

X

null

Return Type

-

Responses

http response codes

Code

Message

Datatype

200

OK

<<>>

Samples

1.1.4. startReceivingFromAll

POST /connectors/start-receiving

Description

Parameters

Return Type

-

Responses

http response codes

Code

Message

Datatype

200

OK

<<>>

Samples

1.1.5. stop

POST /connectors/{connectorName}/stop

Description

Parameters

Path Parameters

Name

Description

Required

Default

Pattern

connectorName

X

null

Return Type

-

Responses

http response codes

Code

Message

Datatype

200

OK

<<>>

Samples

1.1.6. stopReceivingFromAll

POST /connectors/stop-receiving

Description

Parameters

Return Type

-

Responses

http response codes

Code

Message

Datatype

200

OK

<<>>

Samples

1.2. TransactionOperationsController

1.2.1. resumeTransaction

POST /transactions/{transactionId}/resume

Description

Parameters

Path Parameters

Name

Description

Required

Default

Pattern

transactionId

X

null

Return Type

-

Responses

http response codes

Code

Message

Datatype

200

OK

<<>>

Samples

1.2.2. stopTransaction

POST /transactions/{transactionId}/abort

Description

Parameters

Path Parameters

Name

Description

Required

Default

Pattern

transactionId

X

null

Body Parameter

Name

Description

Required

Default

Pattern

StopInfo

StopInfo

X

Return Type

-

Responses

http response codes

Code

Message

Datatype

200

OK

<<>>

Samples

2. Models

2.1. ConnectorInfo

Field Name

Required

Type

Description

Format

name

String

running

Boolean

2.2. StopInfo

Field Name

Required

Type

Description

Format

reason

String

Payment Status Api

Learn Home

Tutorials

Tutorial Introduction

Getting Started

Using an Icon Provided AWS Environment

When using the Icon supplied AWS environments, the setup is completed for you and there's nothing else to do. The core project has been downloaded onto the desktop for you in the 'ipf-tutorial' folder. At various times, the tutorials will refer to the '<install-root>' of the tutorial project - for you this is just his desktop/ipf-tutorial folder.

If you have a look in there, you'll see a 'solutions' folder containing the various solutions to each step of the tutorial. You'll start by using the 'initial' solution.

Setting up your own Environment

If you're using your own environment, you'll need to make sure you have the following installed before starting this tutorial:

Maven - version 3.8.4 or higher (Maven)

MPS/ Flo-designer - version 2021.3 (MPS)

Intellij (or other IDE) (IntelliJ)

Java 11 (JDK11)

You'll also need access to the project repositories are stored in bitbucket. We recommend you use a git client for retrieving the projects, although they can be downloaded through the website also. Please take a note of the location you install the project too as this will be needed in the later tutorials.

If you can't access the tutorial repository, please contact Icon support (support@iconsolutions.com) who will be able to assist.

The preferred approach to run the sample application is using containers: docker containers. However, the tutorial can run as a standard Java application providing that access is available to MongoDB and Kafka.

Concepts to be Aware of

Please note that this tutorial assumes a basic working knowledge of:

The sample application uses the Spring Framework

Event Sourced applications - this means that for every change made to a payment we will raise an event. These events represent a definition of the change, including all the data points that have been altered.

Maven as a build tool

At the core of IPF is the DSL - this is Icon's payment language which allows you to define payment flows. It's used within Jetbrains MPS to create flows and will generate a set of Java components that

form the basis of any implementation. These components use Hexagonal Architecture (aka Ports and Adapters) which separates areas of concerns by interfaces (Ports) and then implementing them using adapters.

Introducing the Basic Application

This tutorial starts by building a simple application that uses some of the core components of IPF. Let's start by defining a few key things:

A Flow - a flow is a single payment process.

An IPF Domain - an IPF Domain is the container around one or more flows. It provides the API for accessing and working with those flows.

An IPF Application - an IPF Application in this case is a simple spring boot application that provides an environment onto which an IPF domain can execute.

An Event - the IPF processing module works by processing events.

The Journal Data Store - the journal data store is the master IPF Data store that contains all the events. For this tutorial, we use Mongo.

Putting this all together, our initial application is setup like this:

Using the Tutorial Code

The tutorial application is broken into three folders:

These contain:

ipf-tutorial-docs : this contains this tutorial documentation in ascii doc format!

solutions : in here are the solution code to every section of this tutorial.

supporting-apps : some sections of this tutorial require access to simulator apps. The sections in here explain how to use these.

So to start, open in IntelliJ <install-root>/solutions/initial/pom.xml. This will load up the initial version of the application. We can use this throughout our work with the tutorial, building upon it at each step of the tutorial to enrich our application.

However, if at any stage you need to change to the expected code, you can simply jump to the solution by opening: install-root>/solutions/<tutorial-name>/pom.xml instead!

Navigating through the Tutorial

This tutorial is broken into a number of different learning paths.

It's recommended to start here: Reviewing the initial tutorial application

Once complete you can work your way through the DSL Path:

DSL 1 - Introducing Icon's DSL

DSL 2 - Opening the Sample Project

DSL 3 - Using a Domain Function

DSL 4 - Using an external domain

DSL 5 - Using a decision

DSL 6 - Using an aggregate function

DSL 7 - Handling Timeouts

DSL 8 - Versioning

DSL 9 - Using Subflows

DSL 10 - Calling other flows (Part One - Within a model)

DSL 11 - Using additional events

DSL 12 - Using custom business data

DSL 13 - Using shared concepts (Part One - Within a Solution)

DSL 14 - Using shared concepts (Part Two - Across Solutions)

DSL 15 - Using shared concepts (Part Three - remote models)

DSL 16 - Dynamic Error Text

These DSL tutorials build up a complex flow to demonstrate how to use different features of the DSL.

Once complete, we delve a little further into implementation and focus on interacting with other systems in the connector path:

CON1 - Adding payment initiation

CON2 - Writing your own connector (Kafka)

CON3 - Writing your own connector (HTTP)

We then look at some of the runtime considerations in:

RES1 - Resiliency and retry settings (HTTP)

RUN1 - Running your application on Kubernetes

And finally we look at how to boostrap your own application through:

BUILD1 - Creating a new project

The tutorial is principally aimed at engineers, however there are many sections - particularly the DSL path that much of the work can be done without any engineering work, and those engineering sections could be skipped if required. Throughout the tutorials it is made clear where engineering knowledge is required.

Learn IPF

Reviewing the Initial Application

Reviewing the initial tutorial application

Getting Started

The tutorial step uses the "initial" solution of the project as it's starting point. You can open it by selecting <install-root>/solutions/initial/pom.xml in Intellij.

Business Orientation

This tutorial uses a common application which we will build upon in a step by step manner to investigate different features and capabilities of the IPF Platform. The initial application is a very simple flow that simply does:

You'll move on from here to add calls to other services whilst looking at advanced concepts like reusability and versioning.

Understanding the application structure

You'll now have a brief dive into the actual application and the way it's structured.

Let's start by looking at the application we have downloaded. If we open the project in Intellij we can see that it has 3 major folders inside it:

domain-root - this contains all the artefacts relating to the DSL and flow definitions.

ipf-tutorial-app - this contains all of the application runtime code, this is the code that will be deployed and use the IPF generated domain to process transactions.

docker - this contains all the script and configuration needed to run the application in a docker environment.

When the application receives a request it will perform a number of simple steps:

It will generate a unique ID for the process.

It will initiate a new flow.

It will generate a new FlowInitiated event.

It will move the flow to the "Complete" state (note that this is simply because of how our flow is current configured, this is investigated in more detail in)

The application is also using the processing data module within IPF. You'll look at this in greater depth in a later tutorial, but for now what we need to know is that the data being generated by IPF is being packaged and sent out over HTTP. In our case, we've added a small "ipf-developer-app" container that has a published endpoint for that data to be pushed to. This we're using as a way to make it easier to see and understand how our IPF Application is working.

Lets now set the application up to actually process a payment.

Application Configuration

The application uses hocon to define properties. You can see the application configuration within the src/main/resources directory of the main ipf-tutorial-app project. In there let's have a look at a few settings within the application.conf file:

```
ipf {

  mongodb.url = "mongodb://localhost:27017/ipf"
```

```
  processing-data.egress {
    enabled = true
    transport = http
    http {
      client {
        host = "localhost"
        port = 8081
        endpoint-url = "/ipf-processing-data"
      }
    }
  }
  system-events.exporter {
    type = ipf-processing-data-egress
  }
  behaviour.retries {
    initial-timeout=250ms
  }
}
```

Let's take each section of this and work through what it is telling us.

Firstly, all the ipf application specific properties are configured to start with "ipf".

The first property we come across is ipf.mongodb.url, in the tutorial case we're using mongo as our database provider so this is simply the connection url. We're assuming in the above that there will be a mongo service running on localhost:27017,

Next we come to the "processing-data.egress". The processing data egress module is used to export data from IPF onto a desired location, for example a Kafka topic or a HTTP endpoint. Here we are defining:

ipf.processing-data.egress.transport to be http, as the tutorial application is using HTTP as it's protocol for sending out processing data.

ipf.processing-data.egress.http contains the HTTP configuration of where we are going to send our data. So in this case we're sending:

ipf.processing-data.egress.http.client.host - this is the host we are sending the data too, in our case the ipf-developer-app we'll talk about below.

ipf.processing-data.egress.http.client.port - the port we're sending to.

ipf.processing-data.egress.http.client.endpoint-url - the URL of the endpoint we're sending to.

We will use the "ipf-developer-app" to consume the data generated from the processing data services. This is a lighweight simple example application that provides a view across the data generated by IPF. It's not a production utility, but is used to assist development. Details to use the developer app are listed here: Running the developer app.

The next section is for "system-events.exporter". System events are IPF's way of emitting information about things that are happening within IPF itself. You'll come onto these in greater depth later. For now the key is simply that we are setting ipf.system-events.exporter.type to be "ipf-processing-data-egress", telling our application to send the system events to the processing data setup we have deployed.

The next configuration is the "behaviour.retries.initial-timeout". Behaviour retries are a mechanism to resubmit commands into IPF if for any reason a reply has not been received back. The default configuration is set to wait 100ms before a reply, however here we're increasing this to 250ms simply under the assumption that this tutorial is being run on a lower grade machine and hence to give it a little more time to complete any requests.

Building the application

To build the application we use Maven, we'll need to run this from the root of our initial module:

mvn clean install

The build will use the defined flows to generate application code. The generated code can be seen in a number of key places:

The generated domain code will be in "domain-root/domain/target" - this will include all the Akka event sourced behaviour implementations together with all the ports that the domain will communicate with.

The generated test code will be in "domain-root/test/target" - this contains the generated test framework code to provide a testing capability against our domain.

The generated sample adapters will be in "domain-root/sample/target - this contains a set of simple sample implementations for all the generated ports. These can be used to quickly bootstrap a complex flow into being executable. At this stage as our flow is so simple, all we have in here is an example configuration file.

Environments

The tutorial supports both running the code using docker or as a standalone spring boot application. Both of these approaches are detailed below:

Docker

Creating the docker container

It is also possible to build a docker image for the tutorial application. To do this we need to use a profile that has been setup to generate the image. If you're intending to use docker to run the application, you can build this now by running:

mvn clean install -rf :ipf-tutorial-app -Pjenkins

The key things to note here is the -Pjenkins flag, this tells Maven to activate the profile that contains the docker image generation plugin. Go and have a look at the pom file of the ipf-tutorial-app to see how this is done!

Note in the above the use of -rf :ipf-tutorial-app → this tells maven to only rebuild the ipf-tutorial-app project. You could rebuild everything and skip this, but since we just generated all our code a moment ago there's no need to rebuild those projects here.

The docker image that is generated is the container that is expected by the example docker configuration in the /docker/application.yml file namely:

```
ipf-tutorial-app:
  image: registry.ipf.iconsolutions.com/ipf-tutorial-app:latest
  container_name: ipf-tutorial-app
  ports:
    - 8080:8080
    - 8559:8558
    - 5006:5005
    - 55002:55001
    - 9002:9001
  volumes:
    - ./config/ipf-tutorial-app:/ipf-tutorial-app/conf
    - ./logs:/ipf/logs
  environment:
    - IPF_JAVA_ARGS=-Dma.glasnost.orika.writeClassFiles=false -
Dma.glasnost.orika.writeSourceFiles=false -Dconfig.override_with_env_vars=true
  depends_on:
    - ipf-mongo
    - ipf-developer-app
  healthcheck:
    test: [ "CMD", "curl", "http://localhost:8080/actuator/health" ]
```

A few things to note around this setup:

We expose the port "5006" - this is the "debug" port, it will allow us to attach a remote debugger.

We map the "logs" directory on our local machine to the "/ipf/logs" directory on the running docker container. This means that we can see all the application logs at docker/logs/ipf-tutorial-app.log

We're using mongo as the database for this tutorial series

Starting the docker environment

You'll start by looking at how to run the application using the docker setup provided. All the docker environment setup is in the docker directory of the root project. Let's have a quick look what is in here:

application.yml - this is the docker file for running the full ipf application. It includes definitions for mongo, ipf-developer-app and the tutorial-app itself.

the config directory - in here we maintain all the docker based configuration.

The important part of the configuration to look at here is that actual configuration for the ipf-tutorial-app itself. Let's look at this by going to docker/config/ipf-tutorial-app. You'll note here we have an application.conf. When we open this we see:

ipf.mongodb.url = "mongodb://ipf-mongo:27017/ipf"

ipf.processing-data.egress.http.client.host="ipf-developer-app"

So here we have just two properties - these are both overriding properties we defined in the application code itself to provide the docker environment specific properties - the difference here being the names of the servers running mongo and the developer-app. We don't need to define the other properties here as they will still be brought through from the main application!

Let's now start up our environment. From the docker directory, run the following command from the docker directory:

docker-compose -f application.yml up -d

This will start the application, a mongo database and an instance of the developer application. After a short while the environment will be ready. We can see the current state of the actual ipf-tutorial-app application by invoking the health check URL:

curl localhost:8080/actuator/health

When this returns a success code, the application is healthy and ready to process work.

Without Docker

As a Spring Boot application

Previously we looked at running the application as a docker container, but we may wish to bring it up as a traditional spring boot application and use some existing infrastructure, for example a shared mongo environment. To do this all we need to do is change the values of "localhost" in the application.conf file within the ipf-tutorial-app project:

ipf.mongodb.url = "mongodb://localhost:27017/ipf"

ipf.processing-data.egress.http.client.host="localhost"

ipf.processing-data.egress.http.client.port=8081

Note that if we're not using docker at all, we'll need to run the ipf-developer-app another way.

For this we have provided a runnable version of the ipf-developer-app, this can be downloaded from the ipf-releases Nexus repository, for example version 1.0.29 is:

nexus.ipf.iconsolutions.com/repository/ipf-releases/com/iconsolutions/ipf/developer/ipf-developer-app/1.0.29/ipf-developer-app-1.0.29-runnable.jar

You must choose and download the correct version of the ipf-developer-app for the IPF Release you are using.

The versions must be compatible, please check and replace the numbers based on IPF's release documentation and/or the details for your target environment!

To run this, we simply need to fire the following command (replacing the version you are using):

java -cp "ipf-developer-app-1.*.**.jar:config" -D"ma.glasnost.orika.writeClassFiles"=false -D"ma.glasnost.orika.writeSourceFiles"=false -D"config.override_with_env_vars"=true -D"loader.main"="com.iconsolutions.ipf.developer.app.IpfDeveloperApplication" "org.springframework.boot.loader.PropertiesLauncher"

If using windows we need to replace the first : with a ; ,namely: "ipf-developer-app-1.0.29.jar;config"

This will run without default configuration namely:

assuming mongo is on localhost:27017

will run itself on 8081

It is possible to change these by adding under a config directory (from the location you're running the file) and override application.conf file. An example is provided in the dev-app folder of the project. This folder also contains the run command and a helper script to execute it.

So once we have a mongo instance we can connect to together with a running implementation of our developer app jar, we can simply start the application just as with any other spring boot application. In Intellij we do this by right clicking on the "Application.java" file and choosing Run > Application (main).

Running a payment

Now that we've started our application, it's time to run some payments through it. Let's start by taking a quick look at the InitiationController.

This class has been built simply as a quick way to interact with the IPF application system and depending on your solution requirements may well not be needed. It uses a simple rest endpoint to provide healthcheck and submission capabilities. In a "real" application, initiation may come from many different channels and over any number of protocols.

If we look at the main initiation method on the controller:

```
    @RequestMapping(value = "/submit", method = RequestMethod.POST)

    public Mono<InitiationResponse> submit(@RequestBody(required = false) InitiationRequest request) {

        final String unitOfWorkId = UUID.randomUUID().toString(); // this is the common id used to link all flow elements and uniquely identify a request.

        String clientRequestId = request != null && request.getRequestId() != null ? request.getRequestId() : UUID.randomUUID().toString();  // this is a client id (if required)
```

```
    String entityId = UUID.randomUUID().toString(); // this is the entity id to uniquely identify the current flow.

    var dummyPacs008 = PayloadGenerator.generatePacs008();

    if (request != null && request.getValue() != null) {

       dummyPacs008.getCdtTrfTxInf().get(0).getIntrBkSttlmAmt().setValue(request.getValue());

    }

    return Mono.fromCompletionStage(IpftutorialmodelDomain.initiation().handle(new InitiateIpftutorialflowInput.Builder(entityId)

          .withProcessingContext(ProcessingContext.builder()

             .unitOfWorkId(unitOfWorkId)

             .clientRequestId(clientRequestId)

             .build())

          .withCustomerCreditTransfer(dummyPacs008)

          .build()).thenApply(done -> InitiationResponse.builder().requestId(clientRequestId).uowId(unitOfWorkId).aggregateId(done.getAggregateId()).build()));

  }
```

We can see here that our controller takes an optional "InitiationRequest", let's look at that too:

```
  @Data

  public class InitiationRequest {

    private String requestId;

    private BigDecimal value;

    private String version;

  }
```

We can see here we've provided 3 properties on our initiation request - a requestId, a value and a version. This will be used in later tutorial sessions in order to test various things.

This tutorial series assumes you have a little utility called jq installed on your machine, this simply will format the returned json from our commands in a pretty fashion. In all of these therefore you can simply remove the | jq to return pure json!

Let's create a payment in our new environment. To do so we simply need to hit the example InitiationController's submit endpoint, which we can do by running the following command:

```
curl -X POST localhost:8080/submit | jq
```

This will return the details of the payment that has been created, such as:

{ "requestId": "220b4868-a96c-4d9a-8b17-b1aa714f05e8", "uowId": "8995c642-a887-4676-a690-2bf261fba172", "aggregateId": "Ipftutorialflow|44d0f1df-eb66-4826-8b1f-63eac604b730" }

Here we can see a number of key things:

The "requestId" - this identifies the request being made. Normally this would be supplied by the calling system, but in our case it is auto-generated by the test "Initiation Controller".

The "uowId" - this is the "unit of work id", it is the single ID that can be used throughout the application to track the payment. This becomes particularly useful for example when considering a payment that spans multiple flows.

The "aggregateId" - this is the ID of an the individual payment flow that has been kicked off.

If we wanted to, we could supply the additional parameter's into our command for example:

curl -X POST localhost:8080/submit -H 'Content-Type: application/json' -d '{"value": "150", "requestId":"abc"}' | jq

This would submit a request with a value of "150" and a requestId of "abc". If you try this now, we'll see the "requestId" in your response having the value "abc". Later we can look to see that the value of our payment has also been set to 150 USD.

Querying the payment

The tutorial application also provides a simple query controller that allows us to make basic calls to IPF to retrieve data. So for example, you can retrieve the status of the payment by calling:

curl -X GET localhost:8080/query/Ipftutorialflow%7C44d0f1df-eb66-4826-8b1f-63eac604b730/status | jq

This should return the value "COMPLETE" - note we'll need to swap the aggregateId given in the command to that which was returned when you created your payment! You'll also need to replace the | character with the encoded '%7C'. Alternatively you could just paste the path into your favourite browser.

The table below shows the list of available endpoints that can be queried. To use them simply swap "aggregate/status" in the command above to the endpoint path given in the table.

Endpoint Path

Provides….

query/{aggregateId}

A full copy of the aggregate for your payment

query/{aggregateId}/status

The current status of your payment

query/{aggregateId}/eventNames

A list of events that have happened to your payment

query/{aggregateId}/flowName

The flow name of your payment

query/{aggregateId}/processingContext

The processing context of your payment

Try a few of these now to get a feel for using the controller and what sort of data it provides!

Note that the query service at this point in the tutorial is a very simple controller based implementation. In later sessions of this tutorial series we'll work on providing much greater depth and possibilities through data projection and introduce IPF's standalone ODS Service.

The payment journal

As IPF performs it's processing, it writes all it's events to the journal. This is a collection which is stored in mongo, under the IPF database. If you use your favourite mongo viewer (For example, our engineers often use Robo3t)

Then you can see the events that have been persisted down to the journal. You should be able to find a record like:

{ "_id" : ObjectId("62bb257a29d64365337f88be"), "eventPayloads" : [{ "deleted" : false, "manifest" : "", "payload" : { "type" : "om.iconsolutions.ipf.tutorial.ipftutorialflow.events.FlowInitiated", "value" : "{"createdAt":1656431994.593398000,"originalCommandId":"HandleInitiateIpftutorialflow|07adee15-80ea-44d0-ab3b-efe3be1ba20e|be7cf0a8-29af-4d96-9018-8efecb8b3bc6","status":{"originatingStatus":"Initial","resultingStatus":"Complete","globalStatus":"ACCEPTED"},"eventId":"Ipftutorialflow|44d0f1df-eb66-4826-8b1f-63eac604b730|1","processingContext":{"associationId":"Ipftutorialflow|44d0f1df-eb66-4826-8b1f-63eac604b730","unitOfWorkId":"8995c642-a887-4676-a690-2bf261fba172","clientRequestId":"220b4868-a96c-4d9a-8b17-b1aa714f05e8","processingEntity":"UNKNOWN"},"initiatingId":"44d0f1df-eb66-4826-8b1f-63eac604b730","causedByEventId":null,"responseCode":null,"originalResponseCode":null,"reasonCode":null,"originalReasonCode":null,"reasonText":null,"originalReasonText":null,"failureResponse":false,"customerCreditTransfer":\{"grpHdr":\{"msgId":"42708ab4-a5ee-4c07-9331-3c34a1438bfa","creDtTm":1656431994.184934000,"btchBookg":null,"nbOfTxs":"1","ctrlSum":null,"ttlIntrBkSttlmAmt":\{"value":10.45,"ccy":"USD"},"intrBkSttlmDt":[2022,6,28],"sttlmInf":\{"sttlmMtd":"CLRG","sttlmAcct":null,"clrSys":null,"instgRmbrsmntAgt":null,"instgRmbrsmntAgtAcct":null,"instdRmbrsmntAgt":null,"instdRmbrsmntAgtAcct":null,"thrdRmbrsmntAgt":null,"thrdRmbrsmntAgtAcct":null},"pmtTpInf":null,"instgAgt":\{"finInstnId":\{"bicfi":"IPSTFRP0","clrSysMmbId":null,"lei":null,"nm":null,"pstlAdr":null,"othr":null},"brnchId":null},"instdAgt":\{"finInstnId":\{"bicfi":"ICSLGBL1","clrSysMmbId":null,"lei":null,"nm":null,"pstlAdr":null,"othr":null},"brnchId":null}},"cdtTrfTxInf":[\{"pmtId":\{"instrId":null,"endToEndId":"866afe26-c173-4921-9587-468f2f974b73","txId":"cecbe387-3906-4351-9fab-dd8b1ad48333","uetr":"1e97e521-d7d6-4913-b941-63e437f884f9","clrSysRef":"2c3b0220-1df4-4a3e-8372-0d7a5a15b3c3"},"pmtTpInf":null,"intrBkSttlmAmt":\{"value":10.45,"ccy":"USD"},"intrBkSttlmDt":null,"sttlmPrty":null,"sttlmTmIndctn":null,"sttlmTmReq":null,"accptncDtTm":1656431994.191809000,"poolgAdjstmntDt":null,"instdAmt":null,"xchgRate":null,"chrgBr":null,"chrgsInf":[],"prvsInstgAgt1":null,"prvsInstgAgt1Acct":null,"prvsInstgAgt2":null,"prvsInstgAgt2Acct":null,"prvsInstgAgt3":null,"prvsInstgAgt3Acct":null,"instgAgt":null,"instdAgt":null,"intrmyAgt1":null,"intrmyAgt1Acct":null,"intrmyAgt2":null,"intrmyAgt2Acct":null,"intrmyAgt3":null,"intrmyAgt3Acct":null,"ultmtDbtr":null,"initgPty":null,"dbtr":\{"nm":"7f0d8165-9c67-4989-865e-15fce48689c1","pstlAdr":null,"id":null,"ctryOfRes":null,"ctctDtls":null},"dbtrAcct":\{"id":\{"iban":"GB

26MIDL40051512345674","othr":null},"tp":null,"ccy":"EUR","nm":null,"prxy":null},"dbtrAgt":\{"finInstnId":\{"bicfi":"ICSLGBL1","clrSysMmbId":null,"lei":null,"nm":null,"pstlAdr":null,"othr":null},"brnchId":null},"dbtrAgtAcct":null,"cdtrAgt":\{"finInstnId":\{"bicfi":"ICSLGBL2","clrSysMmbId":null,"lei":null,"nm":null,"pstlAdr":null,"othr":null},"brnchId":null},"cdtrAgtAcct":null,"cdtr":\{"nm":"bc67af37-93f1-452b-918e-c4fb65f2d6f2","pstlAdr":null,"id":null,"ctryOfRes":null,"ctctDtls":null},"cdtrAcct":\{"id":\{"iban":"GB26MIDL40051512345675","othr":null},"tp":null,"ccy":"EUR","nm":null,"prxy":null},"ultmtCdtr":null,"instrForCdtrAgt":[],"instrForNxtAgt":[],"purp":null,"rgltryRptg":[],"tax":null,"rltdRmtInf":[],"rmtInf":null,"splmtryData":[]}],"splmtryData":[]},"paymentJourneyType":null,"paymentType":null}" }, "persistenceId" : "Ipftutorialflow|44d0f1df-eb66-4826-8b1f-63eac604b730", "sequenceNumber" : NumberLong(1), "tags" : ["tag-1"], "timestamp" : NumberLong(0), "writerUuid" : "06cf3f62-65fe-43db-80f2-7a98a483fea4" }], "hiSeq" : NumberLong(1), "loSeq" : NumberLong(1), "persistenceId" : "Ipftutorialflow|44d0f1df-eb66-4826-8b1f-63eac604b730" }

Let's break the payload down a little so we can see the constituent parts:

"type" : "com.iconsolutions.ipf.tutorial.ipftutorialflow.events.FlowInitiated" - here can see the name of our event.

"status":\{"originatingStatus":"Initial","resultingStatus":"Complete","globalStatus":"ACCEPTED"} - here we can see that this event was received in the "Initial" state and caused the payment to move to the "Complete" state. It's transformed the payment to the global "ACCEPTED" state.

"associationId":"Ipftutorialflow|44d0f1df-eb66-4826-8b1f-63eac604b730" - this is the aggregateId.

"unitOfWorkId":"8995c642-a887-4676-a690-2bf261fba172" - this is the overall payment unit of work id.

"clientRequestId":"220b4868-a96c-4d9a-8b17-b1aa714f05e8" - this is the originating request id.

Note how for the data shown above, these ids correspond to the Ids received as the response to the initial submit request.

The Developer GUI

The IPF Tutorial app has been configured to publish its data so that it can be read by the IPF Developer GUI. This is a lightweight GUI used for testing and to demonstrate the ability to create different views of the data being published by IPF. We can access it by going to localhost:8081/explorer.html. Let's explore a little what it tells us about our simple application.

Search

You'll start by searching for our transaction by using the unit of work ID that was returned to us from our initiation request:

Flows

We should see one record returned, with the basic details of that payment. Let's now press the "view" button to see some more details:

Here we see the "flows" that make up our payment. A payment can have many legs through different flows in it's lifecycle. For the moment we have the simplest setup available and it's just a single flow hence why we only see one record. In for example we'll start adding multiple flows.

In the meantime, we can see many things about our flow so let's try a few of those out.

Graphs

Firstly we can see the graph - click the graph button and we should see the graph we had from our MPS setup before:

Note how this time we are colouring in the flow - the path taken through is highlighted in orange. This will become more obvious later as we build the complexity of our flow and can see our payment taking decisions about which route to run.

Aggregate Data

Let's close the graph and then click to view the aggregate data.

Here we can see all the "live" data of the flow. It tells us everything IPF has in memory about that flow, including all the data available to the flow. This can be either data supplied to the flow via events or calculated by IPF itself (we'll cover this in DSL 6 - Using an aggregate function)

Domain Events

The next interesting view is the domain events one, let's click the orange "Domain Events" button:

This shows us all the domain events that have been recorded during the payment journey. There are a couple of interesting points to note here:

Sequence Number - this is an ordering of the events as they arrive. In our case it's trivial, but in more complicated multi flow cases this will be the ordering across all flows not just one.

Process Flow - this becomes useful when multi-flow are in place.

Content - this provided a full view of the entire domain event.

System Events

Now let's look at the system events view:

Here we can see that our flow has generated 7 system events. System events are a way of tracking what has happened within an IPF application. You'll discuss these in greater depth at a later tutorial, but for now let's just quickly think about what these are telling us. They are effectively providing a story of the "events" that have occurred to our "system" that are associated to this particular payment. It tell's us a story:

_The flow started > A command was received (the initiation instruction) > A domain Event was persisted (the Flow Initiated event) > There was a state transition (from Initial to Complete) > The Flow Finished. _

Have a look at the different events generated here, and view the details they are providing.

Payment Objects

Next lets look at the payment objects tab:

This is showing us the different pre-configured payments objects that are available to the flow. We'll discuss this much more in later tutorials but for now we can see that we have the full pacs.008 available together with the individual credit transaction that is for our payment (noting for now we

are only dealing with 1-1 relationships between pacs008 and credit transfers but later this may become 1-n).

The other tabs - messaging and custom objects are blank for now. These aren't being used by our simple flow as yet. The messaging tab will show us any messages being sent to or from IPF, whilst the custom objects allows a user to define non-payment objects to be stored in this view.

Conclusion

In this section we have successfully:

Built the application.

Started a docker environment.

Executed a payment.

Queried the result of our payment

Reviewed the data in the mongo journal

Looked at the Developer GUI for viewing a transaction.

Tutorials

DSL 1 - Introducing Icon's DSL

DSL 1 - Introducing Icon's DSL

Concepts

This section of the tutorial introduces the concepts within Icon's payments DSL. It is a theoretical walk through and does not require access to any IPF components.

Business Data

We start by considering data. Data is what drives processing and decision making throughout IPF.

The first concept we consider therefore is the "Business Data Element". It has four properties:

A name

A description

A "data type" - the data type can be any Java type, whether it be standard classes like String, Integer etc or your own bespoke types.

A "data category" - an optional field, the possible values are an enumerated set that refers to the type of data that is being represented by this BusinessDataElement. This Data Category label is used by various IPF components such as IPF Data Egress and Operational Data Store, which can automatically record data captured from Process Flows automatically, depending on the Data Category. There are four core data categories:

PAYMENT - This is payment data that is modelled as ISO20022 message components within IPF.

PAYMENT_PROCESSING - This is data that relates to the processing of payments, such as meta-data and payment type information

CUSTOM - This represents custom data which may be attached to the payment

ADDITIONAL_IDENTIFIER - This applies to data elements that represent additional identifiers to be associated with the payment

Any MPS project can have as many different business data elements as you need. These elements are defined within a "Business Data Library" which is simply a collection of related business data and as many different business data libraries can be defined as needed.

IPF provides a number of pre-configured business data libraries. By default, any process is given the "error" library which provides default elements for handling flow failures, namely:

Failure Event Name - this is the name of the event that registered the first failure in a flow.

Failure Response Code - this is the IPF response code for the failure.

Failure Reason Code - this is the IPF reason code for the failure.

Failure Reason Text - this is the IPF text description of a failure.

Failure Original Response Code - This allows specification of any original response code involved (which may have then been mapped to an IPF one)

Failure Original Reason Code - This allows specification of any original reason code involved.

Failure Original Reason Text - This allows specification of any original reason text involved.

The concepts of reason and response codes are discussed later in this document.

Within the lifetime of a payment each business data element is unique and it can be updated as required.

Flow

The processing of a payment is performed by a "Flow". A flow represents a single business process from end to end and is designed to specify the lifetime of a single payment. A single flow might have many paths through it, each representing a different way of processing an individual payment based on the data provided for that flow. A flow contains a number of things:

A name

A description

A version

A global state set

A list of "States"

A list of "Events"

An "Initiation Behaviour"

A list of "Input Behaviours"

A list of "Event Behaviours"

A list of "Aggregate Functions"

A definition of each of these aspects are discussed in the following sections.

The combination of "Flow Name" and "Flow Version" uniquely identify a flow. The version is just an optional numeric identifier, so for example a flow may be called "Test" and have version 3. Then the flow can be unique identified as "TestV3". If there was no version defined it can be identified simply by the name "Test". This identifier is known as the "FlowId"

Global States

First we consider the "Global State Set". The global state set is a set of states that represent the overall state of a payment. It is particularly used where a payment may span multiple flows (for example if the payment processing is split into "initiation" and "execution" parts) but can also apply an overall grouping type state to the individual flow parts to simplify the apparent state transitions from a payment level. Each flow level state can be mapped to a global state such that multiple flow level states can all be considered to leave the payment in the same overall global state.

A default global state set is provided which provides the following standard states: Pending, Accepted, Rejected, Manual Action Required and Cancelled.

States

The next concept to consider within our flow is a "State". This is very simply a resting point on the flow that the payment can pass through in it’s journey, so for example we may have a very simple flow that goes from "State A" to "State B".

A state itself has a number of properties:

A name

A description

A global state

A terminal flag - the terminal flag is used to indicate that this ends the flow to which the state belongs.

Each flow can contain many different states.

Events

When a flow moves from one state to another, this is known as a "State Transition". Within IPF, for a state transition to occur then the system needs to receive an "Event" on the processing journey of the payment. In this case, it is actually a specific type of event known as a "Domain Event". A domain event is a persisted factual occurrence - the arrival of an event means that something explicit has occurred which may cause some form of change to the processing of our payment.

An event has a number of properties:

A name

A description

A list of business data elements.

When an event is formed, then the system will check it's own behaviour to determine what actions should be performed. Whilst this behaviour is explored later in this document, it is worth noting here that there are three occasions when an event can cause a change to the processing, these are known as the "Event Criteria" conditions and are defined as:

On - this movement will happen upon the arrival of a single event (e.g. we may transition when receiving "Event 1")

On any of - this movement will happen upon the arrival of one of multiple events (e.g. we may transition when receiving either of "Event 1" or "Event 2")

On all of - this movement will only occur upon the arrival of multiple events (e.g. we may transition only after receiving both "Event 1" and "Event 2")

Here we have described the "Domain Event" which is the type of event that is explicitly declared within any MPS solution. However, IPF as a whole uses a number of different types of event:

"System Event" - these occur when something happens to the system and can be tailored to be specific to individual needs.

"Action Timeout Events" - these events occur during processing when configured timeout settings are broken.

"Decision Events" - these event are used as a response to receiving results from decisions.

All these event types are discussed later in this document.

External Domains

After an event is processed, the application can then perform one or more activities to determine what happens next on a payment. So for example on receipt of "Event A" we may wish to do some validation and call some other application to ask it to validate our data.

To support this post-event processing, the most important concept is the "External Domain". This represents some business domain - not our current flow's - that we need to interact with.

For example, let's assume that we need to talk to a sanctions system during part of the flow. To support this, we would model that sanctions system as an external domain.

Each external domain consists of the three types of interaction we can make with it:

"Instructions" - instructions are the simplest thing we receive from an external domain. It can be triggered by the external domain at any time and we will start processing. This can be thought of as the external domain pushing information to us.

"Notifications" - notification are the opposite of instructions. These are used when we want to push our data out to an external domain.

"Requests" - a request is used when we need a "response" back from the external domain in reply.

Instructions

Firstly let's consider the instruction. These can be initiated by an external system and contain the following properties:

A name

A description

A list of "Business Data Elements"

When the IPF application receives an instruction it will raise a corresponding event (the decision of which event to raise is described later). The event's business data is then populated with all the matching business data elements.

Notifications

Next up is the notification, like an instruction it has the following properties:

A name

A description

A list of "Business Data Elements"

When the IPF application sends out a notification it will populate on it all the business data elements it has a matching record for.

Requests

Finally, we consider the requests. The request can be thought of to have to parts, the request itself and the corresponding response.

The request part contains:

A name

A description

A list of business data

A list of responses

The response part is slightly different and has some new features to consider:

A name

A description

A list of business data

A "response code set" - this is a group of response codes. A "Response Code" is an expected outcome code for a response that could be used for onward processing. In ISO terms this is analogous with a Status.

A "reason code set" - this is a group of reason codes. A "Reason Code" is a reason why the response is set the way it. So for example your response code could be "Rejected" with a reason "Incorrect Currency". In ISO terms a reason code with a Status Reason.

A completing flag - this defines whether the calling request should be considered completed when this response arrives. So for example consider a request where the external system sends a technical acknowledgement following by a final business response. In this case we would define two responses - one to represent the technical ack (non completing) and one the business response (completing).

In ISO terms, a response code is analogous with a "Status", whilst a reason code is analogous with a "Status Reason"

The system provides a default "AcceptOrReject" response code set which is used for standard pass / fail type responses. It also provides a set of all the ISO reason codes.

Now let’s put these elements together and form the basis of any flow:

So here we can see that when IPF receives something from an external domain (either an instruction or a response), it leads to an event being raised which may cause a state transition followed by the invocation of a notification or request to an external domain.

Domain Functions

It’s possible that we don’t want to have to call an external domain in order to continue processing our flow. This might happen because either we know what to do next or we can calculate what to do next. For this there are two other concepts that we need to consider:

In this case, one option is to use the "Domain Function" capability that the flow itself offers. It works in a very similar way to a request / response pair in an external domain call except that in the case of a domain function the IPF application itself is a domain so the actual call stays internal (consider for example creating an external domain that represents our current flow - this would work the same way as a domain function but would be a mis-representation of the actual control logic). So when we call a domain function, we will expect to get a response and then that response will be transformed into an event which can then cause onward processing.

Like a request, the domain function has a number of properties:

A name

A description

A list of business data

A list of responses

Additional Events

The second option is an "Additional Event"- these can also be used to move the flow on.

When an additional event is raised, the system will process it as though it has been received into the application via an instruction or response.

Let’s add these to our diagram:

Decisions

What however if we want to perform some logic conditionally. So for example, we may only want to run a fraud check if the value of the payment is over £50. In this case we can use a "Decision".

A decision allows us to perform some logic and then take different processing routes subsequently based on the outcome of that decision. A decision has a number of properties:

A name

A description

A list of business data - this is the data that is sent when calling the decision so that it can process based upon it.

A list of "Decision Outcomes" - these are the possible results of running the decision, each decision can have as many different outcomes as needed and these outcomes are unique to the decision. They are defined simply by providing a name.

The decisions themselves are stored within a "Decision Library". The libraries are flow-independent and as such the same decision can be used in multiple flows.

We can use a decision in two places:

To determine which event needs to be raised in response to an input (response or instruction)

To determine which actions need to be performed after a state transition.

Lets add these to our diagram:

A special type of event "A Decision Outcome Event" will also be raised so that the fact the decision has been invoked and a result returned will be audited and can be used on onward processing.

Aggregate Functions

Another useful utility to consider is the "Aggregate Function". An aggregate function is a piece of logic that can be executed upon receipt of an event to perform some kind of logic upon the data received. This data is considered "in flight" and thus is not persisted by the application.

So a good example of this is say a counter that tracks the number of times something has occurred during a flow - each time the function is called we may update that counter. The outcome of the aggregate function then becomes available down the flow.

Another good use case may to perform a data mapping exercise to transform the data into something that can be used downstream.

Let’s add the aggregate function to our diagram:

Behaviours

The next concepts to consider are both types of grouping. In order to separate the logic we need to define when processing an input to the system (from a response or instruction) and generating the event to the logic required when processing the actual behaviour of the system based off that event we have two grouping concepts:

"Input Behaviour" - this is a the behaviour that specifies for each input what event will be generated.

"Event Behaviour" - this is the behaviour that specifies what actions should be taken on receipt of an event.

Input Behaviour

An input behaviour has a number of properties:

An input - this is the input (instruction or response) upon which the behaviour is triggered.

A response code - this is the response code (linked to the response if the response is an input, otherwise this field is not applicable) for which the behaviour applies

An event - this can be either an event directly or via the execution and resulting outcome of a decision.

Note that when using response codes, if one is not defined on an input behaviour this will be considered the "default" behaviour for all response codes.

Event Behaviour

The event behaviour is a little more complicated. It has a number of properties:

A "Current State" - this is the state upon which the flow must be in for the behaviour to apply.

A "Criteria" - this is when the behaviour applies (on / on all of / on any of)

A list of events - one or more events, these can be any type of event (e.g. domain, timeout etc)

A "Move to State" - the destination state of the behaviour

A list of actions - these are the actions that should be performed after the state transition, i.e. requests, notifications etc.

Lets update our diagram to show these:

Note that the aggregate function, as a self contained unit of calculation is not considered as either the event or input behaviour but as a functional capability of it's own

Initiation Behaviour

There is one more key type of behaviour to consider, that is the "Initiation Behaviour". The initiation behaviour is a specialised version the previously defined input behaviour but is only used to start a flow. It is not linked to an external domain so that we can initiate the flow potentially from many different sources.

An initiation behaviour has a number of properties:

A list of business data

An initial state to move to

A list of actions to perform

Note that when the initiation behaviour is invoked, a flow will start and the "FlowInitiated" event will be raised.

We have now reviewed all the components that make up a single flow.

Subflows

The next thing to consider is reusable segments of flow.

For example, consider a sanctions check that may be required to be called from various different places within the flow. We could specify each section of the flow separately and write out the logic each time but ideally we would like to be able to simply reuse common logic each time. This is where the "Subflow" concept is introduced.

A subflow is a reusable flow component. It is essentially the same as a flow in that it has states, input behaviours and event behaviours. However, a subflow has no life of it's own and is only used as a

mechanism of reuse and therefore MUST be contained within an outer flow. When calling a subflow it is very similar in behaviour to receiving an event:

The key thing to understand here that is instead of moving to a state and then calling an action like the normal processing above, here we move to a pseudo-state that acts as an entry point into the subflow. "Control" of the flow is then handed over into the subflow, at this point it will process through the input and event behaviours until it reaches a terminal state in the subflow. When it reaches a terminal state, control will be passed back to the calling flow with a result of the name of the terminal state. This can then be used for onward processing.

Note that in order to achieve reuse of the subflow in multiple places, then when a subflow is placed within a main flow it's state's will be displayed as "<subflowid> <subflow state name>" where <subflowid> is the psuedo-state name of the calling flow and <subflow state name> is the name of the state within the subflow.

Flow Calls

Finally, it's also possible to directly call one flow from another. In this case control is handed over to the secondary flow and we wait for a result execution back. The child flow can report that it has reached any state back to the parent flow. Most commonly, this will be when a terminal state is reached and the child flow has finished processing, but it also allows for feedback from the child flow for intermediary states before it finishes. This provides an opportunity to pass control back and forth between the child and parent as required.

Conclusions

In this section, we have discussed the core concepts that make up the Icon payments DSL and how they fit together. These concepts will be key going forward and the next sections of this tutorial series starts to take us through how to use different ones of these to create our payments application!

Reviewing the Initial Application

DSL 2 - Opening the Sample Project

DSL 2 - Opening the Sample Project

Getting Started

The tutorial step uses the "initial" solution.

In Reviewing the initial tutorial application we looked at the basic project setup and started a flow and ran a payment through it whilst in DSL 1 - Introducing Icon's DSL we learnt about the basic concepts that make up the payments DSL.

Flow's are maintained through the Jetbrains product MPS which then uses Icon's DSL. If you need to download it, it is available as an open source product here: Jetbrains MPS 2021.3.

This section of the tutorial is a beginner's guide to opening the sample flow in MPS and learning basic navigation around it's constituent parts.

Opening the Flow

To start with, let’s open our flow. For this we start the Jetbrains MPS product and you should be prompted to open a project:

As we are using an existing project, we’ll click "Open" here to view our ipf-tutorial project. To open our project we need to navigate to the domain-root/mps directory in our project [<tutorial-install-root>/solutions/initial/domain-root/mps]. It will be highlighted with a small black square in the corner of the directory name, indicating that MPS can identify the directory as the root of an MPS project.

Project Structure

When the project opens up you should be able to see in the project view on the left:

Troubleshooting

If you haven’t previously built your application (for example if you’ve come straight from the solution) you may see this when you open the project:

This means that the language hasn’t been built yet, to fix this simply run from the root of the project.

mvn clean install

If you need to, building and running instructions are covered in more depth during: Reviewing the initial tutorial application

Let’s take a quick moment to discuss the way this is structured.

You’ll start at the lowest level - the Ipftutorialflow. Note the "F" symbol here which denotes that the "Ipftutorialflow" is a flow, we’ll see as we expand and add concepts into our solution that other components have different symbols to declare easily what they are!

Our flow is contained in ipftutorialmodel - a model is a grouping container that allows us to associate flows together. From an application perspective, we work at the model level so all the interaction is done at that level. You’ll introduce the concept of the domain class - which represents the single interface into a model - later.

Our model is contained in Ipftutorialsolution - a solution is a grouping of models, the key thing to be aware of solutions is that they are built together and so it is easy to share components of different models within different solutions.

Finally our solution is contained in "IPF Tutorial Project" - a project is simply a collection of solutions.

Generally speaking, most applications we’ll build will effectively works as a project - single solution - single model - one (or more) flows setup. You’ll introduce in Reviewing the initial tutorial application how we can share models across different projects.

At the top of the project view, we can see that we’re currently in the "logical view". This is the best view for navigating around and working with the DSL. There are also other views available:

These allow us to view our components in different ways, for example looking at the raw files on the File System. This is most useful when trying to review how the generated code looks and is structured.

Viewing the Flow

Now, let's have a look at our flow. If we double click on it in the project view then we should bring up our flow in the main window:

As you scroll down the flow, hopefully we'll start seeing sections to represent the concepts we reviewed in the previous section of the tutorial series.

The Flow Viewer

Lets start by clicking anywhere on the flow tab, and using the main toolbar to choose Tools > Open Flo Viewer. This should show a new tab on the screen which displays a diagram of our flow:

So here we can see we have a very simple flow that receives an initiation request, creates the "Flow Initiation" event and then immediately completes. It's effectively the simplest flow we can have.

The flo-viewer by default opens inline a separate tab. If you want, you can click on the little cog in the top right hand corner:

And then choose to for example undock it which allows you to move the flo viewer. This is great when working with multiple screens!

One of the key features of the flo viewer is that is is updated in real time as your editing your flow! That means you can see live how the changes you are making to your flow definition impacts the flow in a graphical manner.

Intentions

For those familiar with JetBrains products, you may have already come across intentions. These are short cuts functions that are there to provide quick and easy access to common utilities. The Icon DSL has a number of different intentions throughout to try and help make things that little bit easier. You can access the intentions from anywhere by pressing ALT+ENTER. If you do this now we'll see:

So here we can see we have two intentions, one to create a new version of the flow (we'll do this in DSL 8 - Versioning) and one to validate the flow (we'll use this a lot throughout the tutorial series!)

The other thing to note about intentions is that they are specific to what you are currently doing, so as we'll see in later sections the intention list can provide a whole host of useful little tricks that are specific to what you're trying to do at that time!

Inspecting the Flow

Now that we have a flow it's time to build it. First lets check our settings, on the top menu go to build and ensure that both "Save Transient Models" and "Check models before generation" are selected.

The "Check models before generation" option makes MPS validate your model before it tries to build it. Any errors will then be displayed back to you - you can then click on them to be taken to the problem so you can resolve it.

The "Save Transient Models" option makes MPS show you all the steps it's taken to generate the code. Normally we don't need this, but we're setting it here as a way to help you understand what has been done during the build.

Let's build our flow by right-clicking on the Ipftutorialmodel model and click rebuild model ipftutorialmodel. This will take a few seconds, then once finished in the navigator expand out the final transient:

Inspecting the generated artefacts

Here you can see some of the artefacts that have been generated for the flow, these include:

All of the application code - this includes

The generated AKKA Event Sourced Behaviour

All the generated ports that the application needs to integrate to

The generated domain class that allows external interaction to the flow.

The generated documentation

The generated BDD Test Classes

The generated sample implementations of the ports created.

Have a look at some of these artefacts, they will be discussed in greater depth in future sections.

Conclusions

In this section of the tutorial we looked at:

Opening flows within MPS.

Basic MPS Project structures.

Flow Viewer

Intentions

Basic Generation

DSL 1 - Introducing Icon’s DSL

DSL 3 - Using a Domain Function

DSL 3 - Using a Domain Function

Getting Started

The tutorial step uses the "initial" solution of the project as it’s starting point.

If at anytime you want to see the solution to this step, this can be found in "add_domain_function" under solutions!

What is a Domain Function?

A "Domain Function" is any process that you need to undertake within the boundaries of your domain.

These could be a number of things like simple calculations, mapping or invoking business rules.

For the scope of this tutorial, we will assume that the first thing we want to do on our flow is perform a duplicate check.

It will be a very simple check that we send our pacs.008 too and in return we expect either a success result (not a duplicate) or a failure (is a duplicate).

DSL Set Up

Adding the Domain Function

First of all lets create a new domain function library to hold our function within. From the navigator on the left right click on the model:

And then, right click, select New > v2Flo > Domain Function Library.

When prompted enter a name and description for the library like this:

Now lets click the "Add Domain Function" button, this will populate the table with a new row.

To start with lets add the name and description of our domain function - these are simple free text entry boxes:

Name - Duplicate Check

Description - A simple duplicate check

Next we have to choose the business data we will send to our domain function.

In this case we're going to send the customer credit transfer (pacs.008) itself.

To do this we will use the MPS selection capability by pressing "CTRL + SPACE", this will present a list of possible data elements we have available.

Select the "Customer Credit Transfer".

After using "CTRL+SPACE" to bring up the options, you can start typing and this will narrow down the options to only those that match what you have entered.

Now we need to define the response to our domain function. to do this we simply press "RETURN" in the response box to generate a new response, and then for the properties of our response:

For the name we'll use "Duplicate Check Response"

For the description we'll use "A response from the duplicate check"

For our duplicate check, we are not expecting it to return any business data so we can leave it blank, but here you could add business data simply by clicking "CTRL + SPACE" in the box again and selecting from the business data options.

For response codes, as we simply need a positive and negative response we can use the inbuilt "AcceptOrReject" codes. So again here we use "CTRL + SPACE" to bring up the options and choose the "AcceptOrReject" option.

For our duplicate check, we don't need to worry about reason codes so we'll ignore these for now.

For completing flag, we will leave the default selected.

You'll come back to completion flags in a later tutorial.

If you've entered everything correctly your domain function definition should look like:

That’s our domain function defined, so how do we use it within our flow?

Calling the Domain Function

Currently in our flow, on initiation we simply move directly to the complete state:

Now we’ll change this so that we call our domain function, and if successful we’ll move to the complete state but on failure we’ll move to a new rejected state.

So first of all we’ll need to create two new states:

Duplicate Checking - this will be the state the flow is in whilst the duplicate check is being performed.

Rejected - this will be our failure state for when the duplicate check has failed.

Let’s add these to the flow by clicking the "Add State" button within the state section of the flow and then setting them up as follows:

Let’s briefly talk about the other values we’ve set up here:

In our case:

The duplicate checking state is in the middle of the flow while the process is still in flight and more work is expected - hence we select the PENDING global state and mark the state as not terminal.

The rejected state happens when the flow has finished unsuccessfully, so we use the REJECTED global state and as no more work is expected we mark it as a terminal state.

The other thing you might notice in the screenshot above (and your implementation) is that the two new states are underlined in red.

This is a common theme within MPS that errors are marked as red underlined, if we hover over the underline we can see that both errors are because the states are not yet used in the flow.

You’ll fix this going forward.

Sometimes it can be hard to hover over the error, in these cases there are two other options:

On the right hand side of the flow will be little red lines indicating where in the flow the errors are, hover over these to see the error.

If you press "ALT+ENTER" anywhere in the flow we’ll get the intentions popup box, one of the options here will be "Validate Flow". If you select this the flow validation box will come up clearly displaying the errors like:

Here you can simply double-click on an error to be taken straight to the point of failure.

Now lets return to our "Initiation Behaviour" and change it to invoke our domain function.

To do this we change the "Move To State" from "Complete" to our new "Duplicate Checking" which should now be available in the options.

We then select the "Duplicate Check" function within the "Perform Actions" block.

Once done it should look like:

So now when our flow receives the customer credit transfer, we will move to the duplicate checking state and call our duplicate check function.

The next step is to process the response we receive from it.

Using the Domain Function Response

The first thing we need to define here is the events that will occur when we receive our duplicate check response.

In our case we want two events:

To signify that the duplicate check has passed and that this is a unique transaction.

To signify that the duplicate check has failed and that this is a duplicate of an existing transaction.

Let's create these two events by clicking the "Add Event" button, in the "Event Definitions" section of our flow.

Note that for these events, we're not receiving any data so the business data column is blank.

We can see again there are errors on our two events, have a look to see why.

Now that we have our events setup, we need to handle the receipt of the actual duplicate check response.

The response is a type of input, so to process it we need to go to "Input Behaviour".

Let's click the "Add Input Behaviour" button on our flow and we should see this:

So here for the input we need to enter two input behaviours, one that is based on a successful duplicate check response that forwards to the duplicate check passed event, and the other on the failure case going to the duplicate check failed event.

If we set these two up we should see:

Select our "Domain Function Response" (again using "CTRL + SPACE").

For our response code, we'll start on the "Accepted" case so select that.

Then we need to consider the event we wish to raise "Event Selection" (again using "CTRL + SPACE").

Again, check the errors ("ALT+ENTER" → "Validate Flow") and we'll see that it's now prompting us that we need need to use these inputs in our flow.

The final step is to add the "Event Behaviours".

This tells the flow how to act when it receives a particular event whilst in a given state (or states).

So lets click "Add Event Behaviour" and we should see:

So in our case:

"With Current State" - this processing the of the success state should only occur when we're in the "Duplicate Checking State" so we select that.

"When" - we'll leave this as "On" - this means that every time this event is received we will fire the behaviour.

"For Event" - that's our event we're interested in, so for the success case - "Duplicate Check Passed"

"Move to State" - for success, we're just going to move to "Complete" so select that.

"Perform Action" - for now this is the end of our flow, so we'll leave this blank.

When complete, our new line should look like:

Now we need to add the equivalent block for the failure case, see if you can enter this yourself and when ready compare to the solution below.

Once complete, we should be able to validate our flow again ("ALT+ENTER" → "Validate Flow") and see that all the errors have now been corrected.

That's it, we've now added our new step to our flow.

To see it in diagram form we can open the Flow Viewer (Tools > Open Flow Viewer`): and we'll see:

So here we can see that on initiation, our flow will move to the "Duplicate Checking" state, will call the duplicate check function and then depending on the result move to either the "Complete" or "Rejected" state.

Now that we've defined our domain function and told our flow how to use it within it's processing, it's time to look at how we provide an implementation for the domain function.

Java Implementation

Defining the Adapter

Let's switch to IntelliJ to work with the Java side.

First we need to regenerate the application code to pick up the changes we've made in our DSL editing.

You'll do this by running the following from the root of our project (ifp-tutorial):

mvn clean install

This should take a minute or so as all the code and dependencies are generated.

Once it's complete, navigate to the target directory of the domain-root/domain project and we should see a few key classes:

```
package com.iconsolutions.ipf.tutorial.adapters;

import om.iconsolutions.ipf.tutorial.actions.DuplicateCheckAction;

import om.iconsolutions.ipf.tutorial.inputs.DuplicateCheckResponseInput;

import java.util.concurrent.CompletionStage;

public interface TutorialDomainFunctionLibraryPort {

  CompletionStage<DuplicateCheckResponseInput> execute(DuplicateCheckAction var1);

}
```

This method is the definition of our interface to the domain function.

So when the application receives the initiation request, it will make a call to this interface to invoke the domain function and based on the response will raise the appropriate events.

Have a look at the "DuplicateCheckAction" and "DuplicateCheckResponseInput" classes.

You'll see that in our cases, the "DuplicateCheckAction" is a standard POJO containing some flow based meta data together with the pacs.008 input we defined.

Similarly, the key element on the "DuplicateCheckResponseInput" is the "AcceptOrReject" response code enum which allows us to define the outcome of our duplicate check.

To use this we need to supply an implementation of the interface.

This implementation could be as complex as required but for simplicity of this tutorial we're simply going to return a success response.

The implementation needs to be added to the definition of the domain, so let's start by opening the "IPFTutorialConfig" java class (in the ipf-tutorial-app).

The interesting part of this is the configuration of the domain which is currently:

```
@Bean
public IpftutorialmodelDomain init(ActorSystem actorSystem) {
  // All adapters should be added to the domain model
  return new IpftutorialmodelDomain.Builder(actorSystem).build();
}
```

With hexagonal architecture, our implementation of the "Port" is called an "Adapter".

So let's add an adapter to this configuration inline by providing a function that simply returns a new CompletableFuture holding a successful duplicate check response input.

This is just a simple solution for this tutorial step, we'll look at providing more enriched functions later in the tutorial series.

Our new code should look like:

```
@Bean
public IpftutorialmodelDomain init(ActorSystem actorSystem) {
// All adapters should be added to the domain model
return new IpftutorialmodelDomain.Builder(actorSystem)
.withTutorialDomainFunctionLibraryAdapter(input -> CompletableFuture.completedStage(new DuplicateCheckResponseInput.Builder(input.getId(), AcceptOrRejectCodes.Accepted).build()))
.build();
}
```

That's all we need to do to add the domain function into our flow execution.

Checking our Solution

As normal let's now check that the solution works. Start up the application as previously (instructions are available in Reviewing the initial application if you need a refresher!)

Then we can send in a payment:

curl -X POST localhost:8080/submit

And finally if we bring up the payment in the Developer GUI, using the uowId returned on the above curl POST (e.g. "uowId":"36d27cd1-c4c5-4cc2-9a2d-d4833beb718a") and bring up the domain events tab (search by unit of work id, click view, click domain events)

Here we can see that whereas before we only got the "Flow Initiated" Event, we now get both it AND the "Duplicate Check Passed" event! Hence we know everything is working and our duplicate check is being successfully invoked.

Conclusions

In this section we've learnt to use domain functions, both defining them within the DSL and implementing them in the Java solution.

DSL 2 - Opening the Sample Project

DSL 4 - Using an external domain

DSL 4 - Using an External Domain

Getting Started

The tutorial step uses the "add_domain_function" solution of the project as it's starting point.

If at anytime you want to see the solution to this step, this can be found on the "add_external_domain" solution!

In this section we're going to investigate how we can request information from external systems and receive their responses.

What is an External Domain?

From the DSL perspective, an external domain is any other domain with which we need to interact. This is what we refer to as an "External Domain". Usually this is another bank system like Fraud or Accounting but could be for example another IPF system.

We can interact with an external domain in three key ways:

Via an "Instruction" - these are when the external domain pushes information to us.

Via a "Request" - this is when we ask for information from an external domain.

Via a "Notification" - this is when we push information to an external domain.

As an example for our flow, we're going to add a step to our flow that allows us to make a call out to a Accounting System that will validate our account.

From a logic point of view, here we want to send out the account information and retrieve back whether the account is valid. To make it slightly more complex we'll say that the account system can return three responses:

Success

Failure

Not Found

And in the case of failure we'll say that the failure can be because the account has been blocked or some unknown exception has occurred.

Therefore in our example we are "asking" for data, hence we will use a request in this case.

DSL Set Up

Setting up the Response and Reason Codes

In DSL 3 - Using a Domain Function, we used the default "AcceptOrReject" response codes to determine the result of the domain function call. Here we have defined a slightly more challenging requirement which requires the use of both the request and response code concepts.

The Response Codes

Firstly, lets consider the response codes. We have three outcomes "Success", "Failure" or "Not Found". So lets add these as a new response code set that we can use.

If we right click on our model and choose New v2Flo Response Code Library we should be presented with this screen.

Let's go ahead and add a responde code set by clicking on the "Add Response Code Set" button. You'll then set the name of the new set to "Account Validation Response Codes".

Now click on the "Add Response Code" button - this will show us a new table containing the response codes. Let's set them up as follows:

Note here that for the "Failed" and "Not Found" options we've ticket the is failure box. This tells the DSL that these response codes are failure based and that it should store the failure reasons against the payment for analysis.

The Reason Codes

Having set up our response codes, let's now do the same for the reason codes - it's essentially the same process but using the reason code library instead of response codes. If you set it up correctly it should look like:

Alternate Definitions

In this tutorial we have defined the account validation outcomes as a combination of response and reason codes. There are many different ways you may choose to set up your flow depending on your requirements and what work's best for you. For example, just as valid would be to set up a set of four response codes where 'Blocked' is also represented in that way.

Now we've set up our response and reason codes, let set up the external domain and use them within it.

Adding an External Domain

The first step is to add the Accounting System as an external domain, similar to when we created the domain function library. To do this we select the process flow (IpftutorialFlow) and right click then select New v2Flo External Domain. This should bring up the following view:

Lets add a name and a description to start:

name - Accounting System

description - A sample account system

For our requirement we're going to ask the account system to validate our data, so we should model this as a "Request". So let's click the add request button and then supply:

a name of "Account Validation Request"

a description of "Validates the account details"

the business data containing the "Customer Credit Transfer"

This is now a function we could call from our flow, that will take the customer credit transfer we've received from the payment initiation and send it onto the accounting systems validation capabilities. Note here if the accounting system had a specific format that it required, we could define it here and use the IPF mapping capabilities to transform to it. More on that later…

Note that as per previous, we can see that our request has been highlighted as invalid as we haven't yet completed it.

Next we need to add our response, so like domain functions, press return in the "Responses" section and then we'll define:

a name of "Account Validation Response"

a description of "The result message from the accounting system"

no business data

a response code set of "Account Validation Response Codes" ("CTRL + Space" to list & select)

a reason code set of "Account Validation Reason Codes" ("CTRL + Space" to list & select)

a completing flag set to true

When complete, the external domain should look like this:

Updating the Flow

Having created our external domain we need to plug it into our flow, so lets go back to the flow and start by adding a new status for when our application is validating the account:

States

Top Tip

To shuffle the order of table entries hold down "Shift + Ctrl" and use the up/down arrows.

Events

Now we're going to add three new events on our flow:

one for when the check passes

one for failure

one for not known

We as normal do this by clicking the "Add Event" button on the flow and then entering the details as below:

Input Behaviours

As with domain functions, the handling of the responses from the accounting system are handled as "Input Behaviours". So let's click the "Add Input Behaviour" button and add input behaviours for each of our options and then entering the details as below. Again we'll use the CTRL + SPACE to select the input of the request we created above, then select both the first response code, i.e. Accepted, then select the relevant event "Fraud OK". Repeat for response code "Rejected" for event

Finally, we need to plug in both the call to the validation system and how we handle the receipt of our three events into the "Event Behaviour" section of the flow.

Event Behaviours

Let's start by adding in the call to the new validation request. Previously, on receipt of the duplicate checking confirmation (event - "Duplicate Check Passed"), we were completing the flow. Now let's change this so that instead we move to our new "Validating Account" state and call the "Account Validation Request". This should like the below when done:

Now it's calling the request, we need to handle the response. In our case, we're going to say that success goes to complete whilst both not found and failed will go to our rejected state.

For the failure case, we'll use a new feature of the event behaviour that we've not seen before - the ability to use the "On Any Of" when condition. This means we can say when either the "Account Not Found" or the "Account Validation Failed" event arrives, we can perform the same functionality.

We do this just as before, except choosing "On Any Of" from the "When" drop down. Once we've selected the first of our two events, we simple press return and then can use our normal selection approach to pick another.

Once complete this should like:

That's now all the flow definition work complete, but before moving on let's open the flow viewer again (Tools > Open Flow Viewer) and see our new diagram.

And as expected here, we can see that we have now included our account validation request in our happy path.

Now we look at the implementation side of this, if you're not interested in this then you can jump straight to the exercise.

Java Implementation

Let's now switch back to Intellij and look at how we plug this into our implementation code. As normal we'll start by running a build from a terminal window in the root directory of our project:

mvn clean install

Once built, we can again look at the generated code in /domain-root/domain/target and we should now find the port for calling out to our accounting system like this:

```
package com.iconsolutions.ipf.tutorial.adapters;

import om.iconsolutions.ipf.tutorial.actions.AccountValidationRequestAction;

import java.util.concurrent.CompletionStage;

public interface AccountingSystemActionPort {

   CompletionStage<Void> execute(AccountValidationRequestAction var1);

}
```

So just like with our domain functions, this is what we need to implement in order to be able to call out to our external systems. This is the point that in a typical implementation we would use the connector framework to make a call out to the external system over a protocol (e.g. kafka). You'll deal with the connector framework later in this tutorial series so for now we'll use another feature of our generated code - the sample application.

First we need to add the dependency to the sample app, we do this by adding the following dependency into the pom for the ipf-tutorial-app (ipf-tutorial/ipf-tutorial-app/pom.xml).

```
<dependency

    <groupId>com.iconsolutions.ipf.tutorial.{solution}.domain</groupId>

    <artifactId>sampleapp</artifactId>

    <version>${project.version}</version>

</dependency>
```

Note that solution will be whichever solution folder you are currently working in e.g. initial would be com.iconsolutions.ipf.tutorial.initial.domain

If we look inside the sample app project (domain-root/sampleapp/target/generated-sources/com/icon/tutorial/ipftutorialmodel/sample_app) we can see sample implementations of all of our ports. Functionally all these do is return with the first response code available to it but they are sufficient to be able to get a running application going. This makes them extremely useful as they can be used immediately to get our application running and then swapped out one by one as real implementations our developed.

Let's now add the sample AccountSystemActionPort implementation into our configuration. We do that by going back to our IPFTutorialConfig and adding an extra line:

```
@Bean

public IpftutorialmodelDomain init(ActorSystem actorSystem) {

   // All adapters should be added to the domain model

   return new IpftutorialmodelDomain.Builder(actorSystem)
```

```
        .withDomainFunctionAdapter(input -> CompletableFuture.completedStage(new
DuplicateCheckResponseInput.Builder(input.getId(), AcceptOrRejectCodes.Accepted).build()))
        .withAccountingSystemActionAdapter(new SampleAccountingSystemActionAdapter())
        .build();
}
```

Here we can see we've added a declaration to use our SampleAccountSystemActionAdapter.

That's everything we need to do to add a simple implementation of our call to the accounting system. In later sections we'll provide more complex examples and how we could use Icon's connector framework to call out to real systems.

Checking our Solution

We're now ready to go and run our application again, Let's start up the application as previously (instructions are available in Reviewing the initial application if you need a refresher!)

Then we can send in a payment:

curl -X POST localhost:8080/submit | jq

And finally if we bring up the payment in the Developer GUI and bring up the domain events tab (search by unit of work id, click view, click domain events)

Here we can see we now have three events coming through including our new "Account Validation Passed" as the 3rd event.

Exercise

As an exercise let's add another step to our flow. Let's imagine we need to run a fraud check as part of our flow. The requirements of the fraud check service we will call are:

It takes a pacs008 as input data.

It can return either a pass or fail.

We only run the fraud check if the account has been successfully validated.

If the check fails, the payment should be rejected.

If the check passes, the payment should be completed.

Go ahead and try to implement a solution for this. Use the query service to check that your new Fraud capability has been invoked.

When ready you can compare your solution to the "add_external_domain" solution.

Conclusions

In this section we:

successfully created our first external domain which allowed us to simulate making a call out to an external accounting service.

set up a request within our domain and integrated that into our event behaviour to send a request and later receive a response from the account system.

had a light touch introduction to response codes!

Now having configured our application to make calls to an external service, let's discover how to use routing logic in our flows via: DSL 5 - Using a decision

DSL 3 - Using a Domain Function

DSL 5 - Using a decision

DSL 5 - Using a Decision

Getting Started

The tutorial step uses the "add_external_domain" solution of the project as it's starting point.

If at anytime you want to see the solution to this step, this can be found on the "add_decision" solution!

What is a Decision?

So far we have added a domain function (internal call) and an external domain (external call). Now it's time to consider some of the processing options available to us to determine how to process our payments. For that we use "decisions".

A "decision" in the DSL is very simple, it is a function that takes in some data and returns a result. It is similar to a domain function and indeed you could perform most decision logic using a domain function, however decisions have the advantages of:

A decision does not change data

A decision does not require definition of responses

A decision does not require definition of input behaviour

A decision does not require events to be defined.

Effectively therefore a decision can be thought of as a lightweight type of domain function that is used only for directing processing.

In this section of the tutorial we're going to use a decision to perform the following logic:

If the value of the payment is < 10, then we skip the fraud check.

If the value of the payment is > 10, then we execute the fraud check.

It's a very simple use case but will illustrate the basics of the concept.

DSL Set Up

Adding a Decision

Let's start by adding a new decision library to our model. So as previously we right click on the model and go New v2Flo Decision Library.

This should bring up the new decision library page:

So lets go ahead and add our logic for our decision. Firstly we'll add the fields as we've done many times in previous steps:

For name we will use "Run Fraud Check"

For description we will use "Checks payment value requires a fraud check"

For business data we will use "Customer Credit Transfer"

The next field we have is "Outcomes" which is a new concept.

An "Outcome" is simply a possible result of the decision. A decision can have as many different outcomes as you like - these are very similar to the "Response Codes" we used earlier.

The outcome field is simple free text allowing us to enter a name, so let's specify two outcomes:

FRAUD_REQUIRED

SKIP_FRAUD

To enter new outcomes we simply press the return key after completing the previous one.

Once complete, our decision should look like:

That's our decision defined and ready to use, so the next step is to integrating it to the flow.

Using a Decision

A decision is handled slightly differently to the domain function or external domain calls that we have used to date. In those examples, we used specific concrete steps to represent the fact that the system was performing those functions. In the case of a decision however, it is only a routing position on the flow and as such we now introduce the concept of a pseudo state. These will be used in different cases later in the tutorial but for now we'll just use the one we need for decisions which we'll refer to as a decision state.

As mentioned in the introduction, the decision is very lightweight so there is no need to add events or input behaviours here. We can jump straight to the Event Behaviour processing.

We effectively now want to say "Once we have received the Account Validation Passed event, we should run the fraud check decision. If it requires a fraud check, we'll execute the fraud check as before otherwise we'll simply complete our payment".

So let's start by editing our handing of the "Account Validation Passed" event, before we had:

Now instead of moving to the "Checking Fraud" state we will move to the "Run Fraud Check" decision state and execute our decision. For this instead of selecting a concrete state like "Checking Fraud" in the "Move To State" section we choose to "Create Decision State".

Here we can enter the name of the state as "Run Fraud Check" (noting that this does not have to be the same name as the decision we intend to use. It is a good general rule to do so but by allowing any naming here we can reuse the same decision in multiple different use cases throughout our flow).

Then in the perform action, we choose "Routing Decision" and then we simply choose the "Run Fraud Check" decision (which we created in our "Decision Library".

Once complete, we should look like this:

Now let’s handle the result of our fraud check. So, for that we will to add Event Behaviours to either complete the flow (if the check is skipped) or run the fraud check (if the check is required).

So let’s start by adding the skip case. You’ll add a new event behaviour and then specify:

the "With Current State" to be our "Run Fraud Check" decision state.

the "When" to be "On"

For the event, we use a special type of event - the "Decision Outcome Event" so we select "Decision Outcome" and then choose our "SKIP_FRAUD" outcome. Then to complete our behaviour we simply move to "Complete". Once finished it should look like:

Next we need to do the behaviour when the decision says we need to run the fraud check. In that case we simply want to move to the "Checking Fraud" state and run the "Fraud Check" request. Try and add this, or if you prefer the solution is below:

That’s it, we’ve now integrated our decision into our flow. Let’s open it in the Flow Viewer (Tools > "Open Flo Viewer") and see what it looks like:

Top Tip

As our graphs get more complicated there’s a few useful things to be aware of.

The first is at the top we have an option to "Show Actions" by default this is on but unchecking it will change the view to only show state transitions which may be easier to view.

The second is that we have the ability to manipulate the view of the diagram:

Shortcut

Purpose

Ctrl+I/CTRL+O

OR

SHIFT+Mouse Right Click

Zoom In/Out

Arrow Keys

OR

SHIFT+Mouse Left Click

Move around the diagram

CTRL+Left Click

Zoom in to drawn selection box

CTRL+Right Click

Rotate diagram

Note: If you want to reset to the default view you will need to minimise/close the flo viewer and re-open

Java Implementation

The Decision Interface

Let's switch to Intellij to work with the java side.

First we need to regenerate the application code to pick up the changes we've made in our DSL editing. You'll do this by running the following from the root of our project (ifp-tutorial):

mvn clean install

This should take a minute or so as all the code and dependencies are generated. Once it's complete, navigate to the target directory of the domain-root/domain project and we should find a new Decision Interface in the decisions package (& emun for generated outcomes):

```
package com.iconsolutions.ipf.tutorial.ipftutorialmodel.decisions;

public interface DecisionLibraryPort {

  RunFraudCheckDecisionOutcomes performRunFraudCheck(RunFraudCheckDecisionParameters decision);

}
```

So as with other ports we have considered previously, the check takes in a set of parameters - in our case this is a holder for just the customer credit transfer but we may be sending multiple items to business data to a decision. It will return a new enum element (RunFraudCheckDecisionOutcomes) which has been generated for each of the outcomes we defined.

As per our requirements, if the value of the payment is <10 we should skip the fraud check and if > 10 we should run the fraud check. Have a go at implementing this logic and wiring it into the domain. Once ready, you can see the solution below:

```
@Bean

public IpftutorialmodelDomain init(ActorSystem actorSystem) {

   // All adapters should be added to the domain model

   return new IpftutorialmodelDomain.Builder(actorSystem)

        .withDomainFunctionAdapter(input -> CompletableFuture.completedStage(new DuplicateCheckResponseInput.Builder(input.getId(), AcceptOrRejectCodes.Accepted).build()))

        .withAccountingSystemActionAdapter(new SampleAccountingSystemActionAdapter())

        .withFraudSystemActionAdapter(new SampleFraudSystemActionAdapter())

        .withDecisionLibraryAdapter(input ->


input.getCustomerCreditTransfer().getCdtTrfTxInf().get(0).getIntrBkSttlmAmt().getValue().compareTo(BigDecimal.TEN)>0 ?
```

RunFraudCheckDecisionOutcomes.FRAUDREQUIRED :
RunFraudCheckDecisionOutcomes.SKIPFRAUD)

.build();

}

That’s it, our implementation is now complete.

Checking our Solution

Let’s check our application changes work, as usual we’ll start up the application as previously (instructions are available in Reviewing the initial application if you need a refresher!)

Fraud Required

Now we can use our ability to send in different payment values using the values on the initiation request (see for a refresher!).

curl -X POST localhost:8080/submit -H 'Content-Type: application/json' -d '{"value": "150"}' | jq

This will send in a payment value of 150 USD. So remembering our implementation in this case a fraud check should be required. Let’s bring up the payment in the Developer GUI and bring up the domain events tab (search by unit of work id, click view, click domain events)

So we are now getting the new event for "Run Fraud Check FRAUD_REQUIRED" followed by the fraud check running, and we can see that our code is working as we would expect.

For interest, let’s also look at the graph here (using the "Flows" button)

Here we can see the path being taken, including the decision itself (represented as a diamond).

Fraud Skipped

Let’s now try our reverse case by sending in a payment with a value < 10 USD and then look at the events for our returned payment (remembering to replace the aggregate id with yours!):

curl -X POST localhost:8080/submit -H 'Content-Type: application/json' -d '{"value": "5"}' | jq

This time, we would expect to see the decision execute and decide that we can skip fraud and let the flow complete immediately.

Let’s bring up the payment in the Developer GUI and bring up the domain events tab (search by unit of work id, click view, click domain events)

And we find exactly what we expected! Hence we’ve shown that we’re now invoking our decision and we’re successfully skipped our fraud check!

Conclusions

In this section, we’ve:

Added a decision

provided a default implementation that checks payment value

used our decision to drive flow routing

verified our decision is working and being called

Now having configured our application to make decisions on how to process the payment, let’s look an another capability: DSL 6 - Using an aggregate function

DSL 4 - Using an external domain

DSL 6 - Using an aggregate function

DSL 6 - Using an aggregate function

Getting Started

The tutorial step uses the "add_decision" solution of the project as it’s starting point.

If at anytime you want to see the solution to this step, this can be found on the "add_aggregate_function" solution!

What is an Aggregate Function?

An aggregate is the live data store for a particular payment, it’s kind of the summary of all the events received to date in one place so that it can be quickly used and passed on where needed. However, what happens if you want to perform an action against that aggregate, say transform some data, track updates or do a calculation? This is where aggregate functions come in, the provide the capability for us to run a function against the aggregate data when a new event comes in.

An aggregate function therefore is relatively simple to define:

An aggregate function occurs on receipt of a given event

An aggregate function has access to the events data and any data previously on the aggregate

An aggregate function defines which data it provides in return - this could either be an update of previous data or the generation of new data.

Data generated by an aggregate function however is not persisted to the master journal.

We’re going to use an aggregate function as a method of being more precise in what we send to our accounting system when we made our Validate Account request in. In that case we sent the entire pacs008 to the accounting system call, now we want to just send the part of the pacs008 that has the account information in. So let’s use an aggregate function as a way of splitting up the pacs.008 and just making the relevant information available.

DSL Set Up

Setting up a new Business Data Point

Up until now we haven’t really considered business data, only using the supplied customer credit transfer. Now that we want to use a different type of data we will need to define that to make it available to our flow. This is where business data comes in.

Let’s start by creating a new business data library by right-clicking on our ifptutorialmodel and selecting (New > v2Flo > Business Data Library).

You’ll be presented with the business data library screen where we’ll set:

A description of "A sample data library"

A business data element (clicking "Add Business Data") containing:

A name of "Creditor Account"

A description of the "the creditor details"

For the data type we'll select "CashAccount" (note - there are types for each ISO message type, you will need the pacs008 version)

Once done we should see:

Top Tip

Any java object can be used as a business data type. These java types can be imported into the project (we'll cover this in <LINK TO OBJECT RESUSE>. If the type is in scope but not showing in the type ahead, trying pressing CTRL+R as this will search across objects not currently imported that the model can see too!

Updating the Request

Now that we have our data type, the first job is simply to swap out the "Business Data" on the call to the "Account Validation Request" so that instead of taking the "Customer Credit Transfer" as input business data we now take the "Creditor Account". Once done the new request definition should look like:

That's all the supporting setup complete, now it's time to actually create our aggregate function. If we remember, the function was due to extract the creditor details from the customer credit transfer. In our case, the customer credit transfer details come in on the "Flow Initiation" event, so let's use that to set up our aggregate function.

Creating the Aggregate Function

So we start by clicking the "Add Function" button on our flow and then entering the following parameters:

A name of "Extract Creditor Account"

A description of "extracts the creditor account from the pacs008"

An event of "Flow Initiation"

the Input data as "Customer Credit Transfer"

the output data as "Creditor Account"

So here's we are simply telling the flow that when the customer credit transfer is received during initiation, to extract the creditor details so that it can be used later in the flow. Our new function should look like:

That's all our flow work done, the flow will now extract the creditor account details from the pacs008 and use that to send on to the accounting system.

Now let's look at the implementation side of this requirement.

Java Implementation

Let's now switch back to Intellij and look at how we plug this into our implementation code. As normal we'll start by running a build from a terminal window:

mvn clean install

Once built, we can again look at the generated code in /domain-root/domain/target and we should now find the port for calling out to our aggregate function like this (/domain-root/domain/target/classes/com/icon/tutorial/ipftutorialmodel/ipftutorialflow/aggregatefunction):

```
public interface IpftutorialflowAggregateFunctionPort {

  ExtractCreditorAccountForFlowIpftutorialflowAggregateFunctionInput
performExtractCreditorAccount(ExtractCreditorAccountForFlowIpftutorialflowAggregateFunctionPar
ameters parameters);

}
```

Here we can see the definition of our aggregate function port, it's pretty straight forward. So we now need an adapater implementation for it. We do this as normal by adding it to the domain declaration in the IPF Tutorial configuration. Try and do that now and an implementation is below:

```
@Bean

public IpftutorialmodelDomain init(ActorSystem actorSystem) {

    // All adapters should be added to the domain model

     return new IpftutorialmodelDomain.Builder(actorSystem)

          .withTutorialDomainFunctionLibraryAdapter(input ->
CompletableFuture.completedStage(new DuplicateCheckResponseInput.Builder(input.getId(),
AcceptOrRejectCodes.Accepted).build()))

          .withAccountingSystemActionAdapter(new SampleAccountingSystemActionAdapter())

          .withFraudSystemActionAdapter(new SampleFraudSystemActionAdapter())

          .withDecisionLibraryAdapter(input ->


input.getCustomerCreditTransfer().getCdtTrfTxInf().get(0).getIntrBkSttlmAmt().getValue().compareT
o(BigDecimal.TEN)>0 ?

                    RunFraudCheckDecisionOutcomes.FRAUDREQUIRED :
RunFraudCheckDecisionOutcomes.SKIPFRAUD)

          .withIpftutorialflowAggregateFunctionAdapter(input -> new
ExtractCreditorAccountForFlowIpftutorialflowAggregateFunctionOutput(input.getCustomerCreditTra
nsfer().getCdtTrfTxInf().get(0).getCdtrAcct()))

          .build();

}
```

Checking our Solution

Here we can see that we have added our aggregate function to our configuration and everything is now complete and ready to test, so as normal let's now check out solution works. Start up the application as previously (instructions are available in Reviewing the initial application if you need a refresher!)

Then we can send in a payment:

curl -X POST localhost:8080/submit -H 'Content-Type: application/json' -d '{"value": "5"}' | jq

Note - the choice of payment value is fairly arbitrary here as we're only interested in seeing the change to accounting data. To check our change is working, we'll need to look at the aggregate itself. So this time we'll query for the whole aggregate:

Let's bring up the payment in the Developer GUI and bring up the aggregate view (search by unit of work id → click "view" → "click to view aggregate data") and then towards the end of the aggregate details we should see the new creditor account details:

So there we have it, we have successfully extracted out the creditor details from our aggregate and can now use it elsewhere in the flow!

Conclusions

In this section we've:

learnt how to use aggregate functions to convert data types.

Now having configured our application to use an aggregate function, let's look an another capability in: DSL 7 - Handling Timeouts

DSL 5 - Using a decision

DSL 7 - Handling Timeouts

DSL 7 - Handling Timeouts

Getting Started

The tutorial step uses the add_aggregate_function solution of the project as it's starting point.

If at anytime you want to see the solution to this step, this can be found on the handling_timeouts solution.

Action Timeouts

In this section we'll look at how we handle action based timeouts. Suppose we have a simple flow that is making a call out to an external domain, the call is successfully made, but then we never receive the allotted response. In that scenario, we may want to perform some form of compensating action.

To illustrate this, we're going to use our Fraud system and look at how we cope with not getting a response from the downstream domain.

Configuring an Action Timeout (in the DSL)

Configuring action timeouts in the DSL is simple, all we need to do is add a special type of Event Behaviour line to tell the flow what compensating approach to take.

Let's start therefore by opening MPS and going to our flow. Then let's add a new Event Behaviour ("Add Event Behaviour").

Let's add the basics of our event behaviour by saying it's on the "Checking Fraud" state - that's the state we would be in had we successfully made our call to the fraud system.

Now from the "For Event", we choose a new type we haven't used before - the "Action Timeout"

Once selected, we have to choose the action that applies:

The first thing to note here is the default that has been applied "Any action" - this means that a timeout of "any action" on the "checking fraud" state will invoke the behaviour. In our case we only invoke the "Check Fraud" action on checking fraud so it doesn't make too much difference, but there maybe occasions we are firing multiple actions and want to handle the outcome of any of them timing out.

For our scenario, we will therefore simply select "Check Fraud".

Now we need to enter the "Move To State" and "Perform Actions" as per any other event behaviour. Here we could do anything we would normally like call other actions, decisions etc. However, for simplicity in this tutorial we'll simply move our flow to a "Timed Out" state.

So let's add a new state called "Timed Out", we'll mark is as terminal to treat it as the end of the flow and to let our initiation flow know what's happening!

and set that to our move to state. We won't perform further actions.

Our event behaviour will look like:

That's all our DSL work done, now let's look at our graph for our flow.

Open the flow by going to Tools Open Flo Viewer. This will open the flow and at first glance, this doesn't look any different. However, at the top of the panel we'll see:

So here we now have a new option "Show Timeouts". Let's click that box and "Apply" the updates (note - if you have Flo Viewer open already, you may have to close and re-open to see "Show Timeouts").

The new graph will show:

Here we can see that on the timeout from check fraud we're routing to the new Timed Out state.

Note also that the graph shows that this transition will occur on the "CheckFraudActionTimeoutEvent" event. This event is generated by the application when the fraud call times out.

Java Implementation

As normal, let's start our implementation by regenerating our code base.

mvn clean install

Let's start by looking a little at how the code works.

You'll start by opening the "SchedulerPort" interface (part of the IPF Core):

```
public interface SchedulerPort {

   void schedule(ScheduleItem var1, Function<ScheduleItem, CompletableFuture<Void>> var2);

   void schedule(ScheduleItem var1, Function<ScheduleItem, CompletableFuture<Void>> var2,
Duration var3);

   void cancel(ScheduleItem var1);

   void cancelAll(String var1);

   int getRetries(ScheduleItem var1);

}
```

It is these functions that the generated code will invoke whenever it needs to set a schedule. In our case, whenever an action is called it will invoke the schedule method and provide a Schedule Item which contains the action details and a type of "TIMEOUT".

So if you specify to the scheduler that you want a timeout in 10s on this action, then it will return a failure after 10s if the schedule is still active. However, if in that time a cancel is called then this will close out the scheduler.

The IPF application needs an implementation of the scheduler port to be provided as part of the domain. However, so far we haven't had to specify one! Why is this? It's because by default a no-op scheduler is provided. What we need to do now is provide an appropriate implementation for our case.

The Akka Scheduler

You could here use any scheduler that conforms to the interface definition above, for this tutorial we'll use another one provided by the IPF framework, the AkkaScheduler.

Let's start by adding a dependency into our ipf-tutorial-app's pom (ipf-tutorial/ipf-tutorial-app/pom.xml):

```
<dependency>

   <groupId>com.iconsolutions.ipf.core.platform</groupId>

   <artifactId>ipf-flo-scheduler-akka</artifactId>

</dependency>
```

(note - you may need to reload Maven to pull the dependency down)

By adding the dependency, spring will automatically inject an instance of the scheduler into our application. So all we need to do is configure our domain to use, it for this we simply need to update the domain declaration in IPFTutorialConfig.java to specify the scheduler adapter:

```
@Bean

public IpftutorialmodelDomain init(ActorSystem actorSystem, SchedulerPort schedulerAdapter) {

   // All adapters should be added to the domain model
```

```
    return new IpftutorialmodelDomain.Builder(actorSystem)

        .withTutorialDomainFunctionLibraryAdapter(input ->
CompletableFuture.completedStage(new DuplicateCheckResponseInput.Builder(input.getId(),
AcceptOrRejectCodes.Accepted).build()))

        .withAccountingSystemActionAdapter(new SampleAccountingSystemActionAdapter())

        .withFraudSystemActionAdapter(new FraudSystemActionAdapter())

        .withDecisionLibraryAdapter(input ->


input.getCustomerCreditTransfer().getCdtTrfTxInf().get(0).getIntrBkSttlmAmt().getValue().compareT
o(BigDecimal.TEN)>0 ?

                RunFraudCheckDecisionOutcomes.FRAUDREQUIRED :
RunFraudCheckDecisionOutcomes.SKIPFRAUD)

        .withIpftutorialflowAggregateFunctionAdapter(input -> new
ExtractCreditorAccountForFlowIpftutorialflowAggregateFunctionOutput(input.getCustomerCreditTra
nsfer().getCdtTrfTxInf().get(0).getCdtrAcct()))

        .withSchedulerAdapter(schedulerAdapter)

        .build();

}
```

That's us done from a code perspective.

Configuration

Our final job is to configure our scheduler. Configuration is done by using properties, the property string we require is of the format:

ipf.flow.<FLOW_NAME>.<STATE_NAME>.<ACTION_NAME>.timeout-duration=<DURATION>

The first thing to note here is that we need to provide the values of the three optional parameters - our flow name, state name, action name - and provide the value for the duration.

In our case then our property should look like:

ipf.flow.Ipftutorialflow.CheckingFraud.CheckFraud.timeout-duration=2s

The first thing to note in defining this property is that where we have spaces in any of the components, we simply ignore the space. So for example our action name is actually Check Fraud but we simply use CheckFraud. We're also specifying a duration here of 2 seconds.

Let's add this property into the application configuration. To do this open the file ipf-tutorial-app/application.conf and add the line above.

Enabling a Test Setup

We're almost set to test our timeout, the one thing remaining is to actually make the check fraud call have an ability to timeout. Let's update the definition of the fraud check we supplied previously to not use the sample fraud adapter but to allow an optional time out too. Let's create a new

package in our ipf-tutorial-app project for adapter implementations - com.iconsolutions.ipf.tutorial.app.adapters.

Then in our new package we'll add an implementation of the FraudActionPort.

You'll use the idea of "magic values" to say that if a payment of value > 50 USD is received by the fraud call we'll timeout, otherwise we'll return successfully. Try and implement that yourself and the solution when ready is below (Hint - take a look at the generated class example SampleFraudSystemActionAdapter in domain-root/sampleapp):

```
@Slf4j

public class FraudSystemActionAdapter implements FraudSystemActionPort {

   private static final Logger LOG =
LoggerFactory.getLogger(SampleFraudSystemActionAdapter.class);

   private Duration duration = Duration.ofMillis(10);

   @Override

   public CompletionStage<Void> execute(final CheckFraudAction action) {

      LOG.debug("Received an action of type {} for id {}", action.getActionName().name(),
action.getId());

      if
(action.getCustomerCreditTransfer().getCdtTrfTxInf().get(0).getIntrBkSttlmAmt().getValue().compare
To(new BigDecimal("50")) >= 0) {

         return CompletableFuture.supplyAsync(() -> "delaying response",
CompletableFuture.delayedExecutor(duration.toNanos(), TimeUnit.NANOSECONDS))

               .thenAccept(string -> log.debug("Pretending to timeout the fraud call for aggregate {}",
action.getProcessingContext().getAssociationId()));

      } else {

         return CompletableFuture.supplyAsync(() -> "delaying response",
CompletableFuture.delayedExecutor(duration.toNanos(), TimeUnit.NANOSECONDS))

               .thenCompose((String string) -> IpftutorialmodelDomain.fraudSystem().handle(new
FraudCheckResponseInput.Builder(action.getId(),
AcceptOrRejectCodes.Accepted).build()).thenAccept((Done done) -> log.debug("Sent input of type {}
for id {} with result {}", done.getCommandName(), action.getId(), done.getResult().name())));

      }

   }

}
```

(Note - you may have a slightly different implementation for the FraudCheck (built at the end of section 'DSL 4') and your solution may use different classes (i.e. if you didn't start from a fresh clone at the start of this section). For example the above expects AcceptOrRejectCodes, for the FraudCheckResponseInput, but you may have implemented separate response codes for fraud)

Finally we then need to add our new adapter into our config as normal (we are changing the .withFraudSystemActionAdapter to use our newly created Adapter FraudSystemActionAdapter):

```
@Bean
public IpftutorialmodelDomain init(ActorSystem actorSystem, SchedulerPort schedulerAdapter) {
  // All adapters should be added to the domain model
  return new IpftutorialmodelDomain.Builder(actorSystem)
      .withDomainFunctionAdapter(input -> CompletableFuture.completedStage(new DuplicateCheckResponseInput.Builder(input.getId(), AcceptOrRejectCodes.Accepted).build()))
      .withAccountingSystemActionAdapter(new SampleAccountingSystemActionAdapter())
      .withFraudSystemActionAdapter(new FraudSystemActionAdapter())
      .withDecisionAdapter(input ->

input.getCustomerCreditTransfer().getCdtTrfTxInf().get(0).getIntrBkSttlmAmt().getValue().compareTo(BigDecimal.TEN)>0 ?
                  RunFraudCheckDecisionOutcomes.FRAUDREQUIRED : RunFraudCheckDecisionOutcomes.SKIPFRAUD)
      .withIpftutorialflowAggregateFunctionAdapter(input -> new ExtractCreditorAccountForFlowIpftutorialflowAggregateFunctionOutput(input.getCustomerCreditTransfer().getCdtTrfTxInf().get(0).getCdtrAcct()))
      .withSchedulerAdapter(schedulerAdapter)
      .build();
}
```

That's everything complete, time to build and spin up the container environment to check it all works:

Checking our Solution

As normal let's now check that the solution works. Start up the application as previously (instructions are available in Reviewing the initial application if you need a refresher!)

For payments, we need to reconsider our logic we've built:

If a payment is over $50 (but over 10 to make sure the fraud check is required!) then we time out, if it's not we proceed as before.

So let's try both scenarios starting with a payment over $50:

```
curl -X POST localhost:8080/submit -H 'Content-Type: application/json' -d '{"value": "150"}' | jq
```

Let's bring up the payment in the Developer GUI and bring up the domain events view (search by unit of work id, click view, click domain events):

Here we can see that this time we've successfully got the timeout event coming through. For confirmation, if we now repeat the process with a value of say $25 we'll see that the fraud check has happily processed successfully and not timed out so our flow has proceeded onto completion.

Persistent Scheduling

IPF provides another scheduling interface by default. This is the persistent scheduler. It is quartz backed and the major difference is that is is backed by a persistence layer and as such should the application fail, the schedules will be available post jvm shutdown.

To apply the persistent scheduler we simply need to replace the akka scheduler dependency with the one for our persistent scheduler:

<dependency>

<groupId>com.iconsolutions.ipf.core.platform</groupId>

<artifactId>ipf-flo-scheduler-persistent</artifactId>

</dependency>

If you want, go ahead and try this change, you can repeat the testing we did above to prove that our new scheduler is working.

Conclusions

In this section we've learnt how to setup timeouts on actions and work with when they are invoked.

DSL 6 - Using an aggregate function

DSL 8 - Versioning

DSL 8 - Versioning

Getting Started

The tutorial step uses the "handling_timeouts" solution of the project as it's starting point.

If at anytime you want to see the solution to this step, this can be found on the "add_version" solution!

Versioning

Suppose that we have a flow running in production. A requirement then comes in to add a new step to the flow - so we want the in-flight transactions to stay on the current flow but all new transactions to start processing on the new updated flow. We can achieve this by versioning.

For the purpose of this tutorial we're going to insert a step to call out to a CSM service. We're going to do this by creating a new version of our execution flow that contains the extra step. So let's get on and see how this works.

Defining the CSM Service

The first thing we need to do is to call our CSM service. We will:

Send the CSM a pacs.008

Receive a pacs002 from the CSM - this can either contains an accepted or rejected code.

For this we'll need a new external domain and a request defined on it. It's very similar to what we did in DSL 4 - Using an external domain. You'll call our request "Clear and settle".

See if you can set it up yourself, and the solution is below:

Versioning the Flow

Time to set up our versioning, if we look at our current flow we'll see:

Here we can see it says "No Versioning". Let's go ahead and add versioning, to do this we will press ALT + ENTER on the flow and then select "New Version" from the drop down:

If you look in our navigator, we'll now see:

Let's open "Ipftutorialflow (V2)", what we'll find is an exact clone of our original flow, except it now has the versioned specified as V2 (note - the original flow has Version now specified).

Now we're ready to edit our version two flow. So whereas previously we had the following "Event Behaviour":

You'll change this so that instead of moving to "Complete" and raising our additional event, we will move to a new status "Clear And Settling" and then call our Clear and Settle request (defined on the CSM Service).

If the clear and settle request is successful, we'll move to completing and raise our event. If the clear and settle fails, we'll move to Rejected.

Try and add those conditions now and the solution is below, remember we'll need to add the appropriate States, Event definitions, Input Behaviour and lastly Event behaviour!

(If you need a recap, review: DSL 4 - Using an external domain)

First of here's the state definition:

Then we'll need an event for both positive and negative CSM response:

Don't forget when setting up these events, that the CSM service is sending us back the pacs002 and we want to store that as business data!

Next up is our input behaviour, it's very standard:

And finally our event behaviour!

Note here that we've applied the clear and settling logic to both the "Skip Fraud" and "Fraud Required" cases!

That's everything we need to do, we've now successfully created a new flow version and updated it to use our CSM Service. If we look at the flow for version 2 we'll see:

Whereas the flow for version 1 remains without this step.

That's all our DSL work done, so let's now move onto the implementation side.

Java Implementation

As normal, let's start our implementation by regenerating our code base.

mvn clean install

In this case the build should fail:

If we check the IfpTutorialConfig.java we see:

This is failing because the aggregate function is unique to a flow, and now that we've created our new version we need to tell the application how to apply the aggregate function for each flow - as it's perfectly possible the implementation logic may change. However, in our case we just need to do the same thing, so we'll simply define the new functions in the same way. Note that the new versions will be named V1 and V2 respectively.

We've also added the new CSM Service, so we need to remember to include that in our domain configuration. For this we'll just use the sample one provided.

Have a go at updating and the solution is below:

```
@Bean
public IpftutorialmodelDomain init(ActorSystem actorSystem, SchedulerPort schedulerAdapter) {
   // All adapters should be added to the domain model
   return new IpftutorialmodelDomain.Builder(actorSystem)
        .withTutorialDomainFunctionLibraryAdapter(input ->
CompletableFuture.completedStage(new DuplicateCheckResponseInput.Builder(input.getId(),
AcceptOrRejectCodes.Accepted).build()))
        .withAccountingSystemActionAdapter(new SampleAccountingSystemActionAdapter())
        .withFraudSystemActionAdapter(new FraudSystemActionAdapter())
        .withDecisionLibraryAdapter(input ->

input.getCustomerCreditTransfer().getCdtTrfTxInf().get(0).getIntrBkSttlmAmt().getValue().compareT
o(BigDecimal.TEN)>0 ?
                    RunFraudCheckDecisionOutcomes.FRAUDREQUIRED :
RunFraudCheckDecisionOutcomes.SKIPFRAUD)
        .withIpftutorialflowV1AggregateFunctionAdapter(input -> new
ExtractCreditorAccountForFlowIpftutorialflowV1AggregateFunctionOutput(input.getCustomerCredit
Transfer().getCdtTrfTxInf().get(0).getCdtrAcct()))
        .withIpftutorialflowV2AggregateFunctionAdapter(input -> new
ExtractCreditorAccountForFlowIpftutorialflowV2AggregateFunctionOutput(input.getCustomerCredit
Transfer().getCdtTrfTxInf().get(0).getCdtrAcct()))
        .withSchedulerAdapter(schedulerAdapter)
        .withCSMServiceActionAdapter(new SampleCSMServiceActionAdapter())
        .build();
}
```

(note - if you check domain-root/domain/target/classes/com/icon/tutorial/ipftutorialmodel/domain/IpftutorialmodelDomain.class, you can see V1 and V2 versions to be called)

That's everything we have to do to make our code now work, but before continuing let's go back to our initiation controller (ipf-tutorial-app/src/main/java/com/icon/tutorial/ipftutorialsolution/app/controller/InitiationController.java) and consider how it's working:

```
return Mono.fromCompletionStage(IpftutorialmodelDomain.initiation().handle(new
InitiateIpftutorialflowInput.Builder(entityId)
        .withProcessingContext(ProcessingContext.builder()
            .unitOfWorkId(unitOfWorkId)
            .clientRequestId(clientRequestId)
            .build())
        .withCustomerCreditTransfer(dummyPacs008)
        .build()).thenApply(done ->
InitiationResponse.builder().requestId(clientRequestId).uowId(unitOfWorkId).aggregateId(done.getA
ggregateId()).build()));
```

So we're sending it here just a common "InitiateIpftutorialflowInput" - how does the application know which flow to use? The answer is that by default it will always route "New" work to the latest (V2 in our case) flow. However, we can also choose which flow to send data to.

Now, for testing purposes let's recall that we have a version fields on our InitiationRequest object, let's use that to be able to pick and choose which flow to send to. So let's update our initiation logic to be like the below:

```
 return Mono.fromCompletionStage(IpftutorialmodelDomain.initiation().handle(new
InitiateIpftutorialflowInput.Builder(entityId)
        .withProcessingContext(ProcessingContext.builder()
            .unitOfWorkId(unitOfWorkId)
            .clientRequestId(clientRequestId)
            .build())
        .withCustomerCreditTransfer(dummyPacs008)
            .withVersion(request == null || request.getVersion() == null ? null :
"V1".equals(request.getVersion()) ? IpftutorialflowFlowVersions.V1 : IpftutorialflowFlowVersions.V2)
        .build()).thenApply(done ->
InitiationResponse.builder().requestId(clientRequestId).uowId(unitOfWorkId).aggregateId(done.getA
ggregateId()).build()));
```

Here we can see we've added a line to set the version on the request based off the version property we've supplied (note - the use of the IpftutorialflowFlowVersions enum which contains "V1" and "V2").

In addition to specifying the version as we did above, there are explicit input's (ipftutorialmodel/inputs) you can use too. So instead of InitiateIpfTutorialflowInput in the above you could have used InitiateIpfTutorialflowV1Input for example.

Checking Our Solution

As normal let's now check out solution works. Start up the application as previously (instructions are available in Reviewing the initial application if you need a refresher!)

You'll start by sending in a payment without specifying a version

curl -X POST localhost:8080/submit | jq

Now the first thing to note is the response:

{

"requestId": "c16a5c43-1038-4311-9d3f-8bf34efa0c81",

"uowId": "0945fe73-521c-478e-9f62-df4ac6393091",

"aggregateId": "IpftutorialflowV2|239f3e48-8d26-4a2f-8241-0997dc25f1c2"

}

So here we can see that we've hit our V2 flow from the aggregate id "IpftutorialflowV2|….". If we now double check the events (remember to update the aggregate id to match yours!):

Let's bring up the payment in the Developer GUI and bring up the flow events view (search by unit of work id, click view) and we should see:

Here we can see that the process flow being executed is indeed our V2 flow, and looking at the events view shows the same (click domain events):

Now let's try hitting our V1 version:

curl -X POST localhost:8080/submit -H 'Content-Type: application/json' -d '{"version": "V1"}' | jq

And again, looking at the response:

{ "requestId": "492a0177-d9c3-4845-bc81-f54c9aae917d", "uowId": "d8cf8b99-448b-44e7-8207-a015dc41623a", "aggregateId": "IpftutorialflowV1|b0e19b4a-34f6-4584-8109-eef311fd2a13" }

So here we can see that we've hit our V1 flow from the aggregate id "IpftutorialflowV1|….". If we now bring up the payment in the Developer GUI and bring up the flow view (search by unit of work id, click view) and we should see:

Showing us again we're hitting the right process flow. If we check the domain events tab (click domain events) then we see there's no CSM Service invocation:

If you want, you could now send in a V2 flow request specifically and see that working as well.

Additional Notes

Consequence of Flow Versions & Configuration

In the previous tutorial we configured ActionTimeouts with the following configuration;

ipf.flow.Ipftutorialflow.CheckingFraud.CheckFraud.timeout-duration=2s

Here we used the flow name, Ipftutorialflow, and it worked just fine. However with the versioning of our flows this has actually created 2 flows of different names (with different behaviours) and this needs to be reflected in the config. For each flow you wish to retain this configuration, you should set it as follows (note the addition of V1 and V2 to the names):

ipf.flow.IpftutorialflowV1.CheckingFraud.CheckFraud.timeout-duration=2s

ipf.flow.IpftutorialflowV2.CheckingFraud.CheckFraud.timeout-duration=2s

You can thus have different configurations per flow.

Or if you wish to have the same for all flows you can use the 'Any' wildcard (which will apply the configuration for ALL flows for this CheckFraud action);

ipf.flow.Any.CheckingFraud.CheckFraud.timeout-duration=2s

Handling In-Flight Transactions

There could be cases where a transaction was started on V1 of the flow but is paused waiting on a response/instruction to a long-running process external to the flow. During this time you have deployed V2 of the flow but the in-flight transactions are still not in a terminal state.

This scenario is perfectly valid and when the old transactions on V1 flow resume they will continue on the flow which they started on, in this case the V1 flow. Any new transactions that have been initiated since the upgrade would run on the V2 flow.

It’s not possible for a transaction initiated using one version of a flow (e.g. V1) to be resumed using another (e.g. V2). You either need to resume the flow on the original (V1) flow or initiate a new flow on the new version (V2).

Conclusions

In this section we’ve:

versioned our flow

added a CSM Service call to our versioned (V2) flow

showed the default behaviour of versioning

DSL 7 - Handling Timeouts

DSL 9 - Using Subflows

DSL 9 - Using Subflows

Getting Started

The tutorial step uses the "add_version" solution of the project as it’s starting point.

If at anytime you want to see the solution to this step, this can be found on the "add_subflow" solution!

What is a Subflow?

A "Subflow" is a reusable section of a flow. It effectively has all the same features as a flow but is not expected to be standalone but instead is included inside of another flow.

A subflow can therefore be :

Used within many different flows.

Used multiple times within the same flow.

An often seen example of a subflow is a sanctions check. It may be that many different flows all use a sanctions check or indeed that a sanctions check may be needed at different parts of the flow (for example following a resumption on delay).

For this section, we'll use the sanctions example to integrate a subflow into our flow.

Our sanctions subflow, will need to perform the following actions:

When the subflow begins we make a request to a sanctions system.

Then the sanctions system can either return:

Passed - all is good, flow can proceed.

Wait

Then on the wait condition, we should wait until the sanction systems sends us one of three further messages:

Passed

Blocked

Failed

DSL Set Up

Adding the Sanctions Domain

Before we start looking at the subflow itself, we first need to add an external domain that is capable of supporting our flow. So let's go ahead and do that.

Firstly, we'll create a new external domain for our sanctions sytem. This is just like we did in (New v2Flo External Domain) except that the response needs to use an extra capability we've not fully discussed before. So let's start by adding the basics:

A name of "Sanctions Domain"

A description of "A sample sanctions system"

Add a request with details:

A name of "Check Sanctions"

A description of "Sample sanctions call"

Add the customer credit transfer to business data

Now we consider the responses. There are a number of different ways we could model the required responses using combinations of response codes and reason codes like we did in the fraud system. Here however, we will do it by modelling multiple responses to our single sanctions request.

The first response is the "passed" scenario, we'll call this a "no hit" scenario. For this we just add a simple response like we have before:

A name of "Sanctions No Hit"

A description of "The sanctions check has passed"

All other fields we'll leave as the default.

The second response is the one when the initial check does not immediately pass and we are asked to wait. You'll call this a "hit" scenario.

The key difference between the "hit" and "no hit" scenarios are that:

in the no-hit scenario it completes the request - i.e. we're not expecting any further information from the sanctions system.

in the hit scenario it does not complete the request, we're still expecting a final result of our sanctions call.

So this time, for our no-hit scenario we need to set our "Completing" flag to false to tell the system that we will expect subsequent messages from the sanctions system in response to the initial request.

Let's set this response up:

A name of "Sanctions Hit"

A description of "The sanctions system is checking the outcome"

The completing flag unchecked

All other fields we'll leave as the default.

Finally we also need the result response when the sanctions system eventually sends us a response. You'll call this the "Final Sanctions Response". So let's add this:

A name of "Sanctions Final Response"

A description of "The final result from the sanctions system"

For the response codes we'll need to create a new response code set for our "Sanctions Final Response" in the "Response Code Library". This is just as in DSL 4 Using an External Domain. In our case we'll have three response codes: False Hit, Block, Reject.

All other fields we'll leave as the default.

Once we've put all this together our sanctions system definition should look like:

And for reference our new "Sanctions Final Response Codes" will look like:

Note that we can simply add our response codes to the existing library that we used for our account validation codes.

That’s our sanctions system all set up and ready to use.

Adding the Subflow

Now that we have our sanctions domain set up, let’s go and create our subflow. We do this by right clicking on our model and selecting New v2Flo Subflow.

This should create a new subflow page:

The first thing to note is how similar this is to the flow page. That’s because it is effectively a specialised type of flow! So using this page should feel very familiar.

Let’s setup our new subflow:

Let’s give our sanctions flow a name of "Sanctions Subflow" and a description of "An example subflow for sanctions".

The next thing to consider is our states we are going to need. From the requirements we can see that there are three states we will need:

A "Checking Sanctions" state for when we make the initial request to sanctions and are awaiting a response.

A "Waiting" state for when we have received a wait notification from sanctions and are awaiting the final response.

A "Complete" state for when we have successfully completely a sanctions check.

A "Rejected" state for when the sanctions check has failed.

A "Blocked" state for when the sanctions check has resulted in a blocked notification.

Go ahead and set these up now. Consider what values you will need for the terminal flag and global state on each of these states and when complete the solution is below:

It’s really important within a subflow to get the terminal states correct. That’s because those are the states that the subflow will be able to report back to the parent flow. So here we have "Complete", "Rejected" and "Blocked" as terminal states. That’s because the "Checking Sanctions" and "Awaiting Final Result" states are intermediary states during the subflow processing. We can further emphasize that by the setting of the "PENDING" global state.

Now let’s carry on down our subflow and the next thing to consider is the events. Again going back to requirements we can see that we will need 4 events:

"Sanctions Passed" for a successful sanctions check

"Sanctions Rejected" for a failed sanctions check

"Sanctions Blocked" for a blocked sanctions check

"Sanctions Hit" for an wait notification.

Note here there are many different event names we could use. We've chosen to use the same event "Sanctions Passed" for both the direct and indirect (via wait) passing of the sanctions check. We could just have easily created two events to uniquely identify each.

Let's add these Event Definitions now and we should see:

Continuing on down our subflow our next section to consider is the Input Behaviour. Here we have to consider each of our three different responses from the sanctions system and how we want to handle them. Try to do this yourself and the solution is below:

Next up, on initiation we need to call the sanctions system. So we'll need a new state "Checking Sanctions" and an initiation behaviour that moves us to the "Checking Sanctions" state and calls our sanctions system. Try that now and the solution is below:

Finally we need to handle our event behaviour. See if you can work it out, and then the solution is below:

The one interesting point to note here is how we've handled line 4 in the event behaviour. It would have been just as correct to have two lines here, one for the "Checking Sanctions" current state and one for the "Awaiting Final Result" current state. But we've chosen here to use the ability of having multiple states defined in the current state. This is simply a shorthand way to avoid repeating the same logic multiple times if the outcome is no different.

We've now completed all our sections of the flow, but if we look there is still an error showing on the "Perform Action" of the initiation behaviour. Let's investigate this by validating the flow (ALT + ENTER then validate flow). It tells us:

It's telling us that the subflow doesn't have access to a customer credit transfer and hence it can't make the call out to the sanctions system. In our case, our customer credit transfer belongs to the parent flow. So to provide it down to the subflow we need to add it to the initiation data, let's do that and we should now see:

And our error has been resolved. As normal before completing let's have a look at our graph (Tools > Open Flow Viewer) to find:

That's our subflow all setup and ready to use, so the next question is how do we apply this to our parent flow. Let's go back to it, we'll update only in our latest V2 version:

We're going to add our subflow in as a new step after our account validation. Currently we have:

To slide in the extra step, we want the validating account to call our subflow (not run the fraud check decision) and then run the fraud check on successfully passing sanctions instead.

If you remember back to DSL 5 - Using a decision, we introduced the concept of a pseudo state and in that case the specific type of a "Decision State". Here we want to use a different type of state - a "Subflow State". Let's start by adding it after we received the "Account Validation Passed" result.

So instead of moving to the "Run Fraud Check" decision, lets create our "Subflow State".

To do this we first need to delete the existing decision state and then select "Create Subflow State".

You'll enter the name as "Sanctions".

Then in the "Perform Action" box we'll delete the existing call to Fraud and then select "Call Subflow" and then choose the "Sanctions Subflow". Once done we should look like:

If you inspect the error now we'll see:

So here we can see that the flow needs us to define how to handle the fact that the subflow has reached those Complete, Rejected or Blocked states. Why only those 3? Because those are the ones we specified as terminal with the sanctions subflow.

Let's setup our handling of the subflow terminating as "Event Behaviour" in the main V2 flow. To do this we start by adding a new Event Behaviour with current state of "In Sanctions". Then in the "For Event" we select "Subflow Completion" and then we'll start by selecting "Complete".

In our case on receipt of complete from the sanctions flow we'll need to create our "Run Fraud Check" decision and fire the fraud check decision as we had previously (note - until this is done "Run Fraud Check" shows as an 'Unresolved Reference'):

Now let's do the same to handle the outcome from Rejected and Blocked, for now we'll just send both to rejected.

Again, note here how we've used the "Any Of" capability to minimise the entry.

Finally, let's look at how this has impacted our flow diagram (Tools > Open FlowViewer):

Here we can see that our subflow call has been represented by box. The box contain all of the subflow logic to minimise the complexity of the graph. However, if you look at the top of the graph there's a new option "Expand Subflow". Let's click this and apply to see:

And in this view we can see the subflow being expanded out so we can see the inner workings of the flow.

That's all our DSL setup done. Now let's move onto the implementation side.

Java Implementation

Defining the Adapter

Let's switch to Intellij to work with the java side.

Firstly, let's rebuild our project so that we can generate our new flow. To do this open a new terminal and run:

mvn clean install

There is nothing special about the subflow itself from a generation viewpoint, the changes we have to consider are simply that we have defined a new External Domain (Sanctions Domain) that we now have to implement. You'll again here choose to just use the sample app's implementation as we did in DSL 4 - Using an external domain, so try and add that now and solution is below when ready:

```
@Bean
public IpftutorialmodelDomain init(ActorSystem actorSystem, SchedulerPort schedulerAdapter) {
   // All adapters should be added to the domain model
   return new IpftutorialmodelDomain.Builder(actorSystem)
         .withTutorialDomainFunctionLibraryAdapter(input ->
CompletableFuture.completedStage(new DuplicateCheckResponseInput.Builder(input.getId(),
AcceptOrRejectCodes.Accepted).build()))
```

```
        .withAccountingSystemActionAdapter(new SampleAccountingSystemActionAdapter())
        .withFraudSystemActionAdapter(new FraudSystemActionAdapter())
        .withDecisionLibraryAdapter(input ->

input.getCustomerCreditTransfer().getCdtTrfTxInf().get(0).getIntrBkSttlmAmt().getValue().compareTo(BigDecimal.TEN)>0 ?
                RunFraudCheckDecisionOutcomes.FRAUDREQUIRED :
RunFraudCheckDecisionOutcomes.SKIPFRAUD)
        .withIpftutorialflowV1AggregateFunctionAdapter(input -> new
ExtractCreditorAccountForFlowIpftutorialflowV1AggregateFunctionOutput(input.getCustomerCredit
Transfer().getCdtTrfTxInf().get(0).getCdtrAcct()))
        .withIpftutorialflowV2AggregateFunctionAdapter(input -> new
ExtractCreditorAccountForFlowIpftutorialflowV2AggregateFunctionOutput(input.getCustomerCredit
Transfer().getCdtTrfTxInf().get(0).getCdtrAcct()))
        .withSchedulerAdapter(schedulerAdapter)
        .withCSMServiceActionAdapter(new SampleCSMServiceActionAdapter())
        .withSanctionsDomainActionAdapter(new SampleSanctionsDomainActionAdapter())
        .build();
}
```

Here we have added into our config the addition of the sanctions system specification.

Checking our Solution

As normal let's now check out solution works. Start up the application as previously (instructions are available in Reviewing the initial application if you need a refresher!)

For payments, we'll just fire in a standard one:

```
curl -X POST localhost:8080/submit | jq
```

Then as normal, if we now bring up the payment in the Developer GUI and bring up the flow graph (search by unit of work id, click view, click view graph) and we should see:

And we can see that our subflow has been expanded out to be part of the main running flow which has completed successfully. If we look at the events instead (click domain events):

We should note that the event name is the combination of both the prefix we provided in our pseudo state together with the actual name of the event in the subflow. This is important to realise as it is this capability that allows us to use our subflow in multiple places across our flow. You can try this yourself if you want by adding a second sanctions subflow call into the flow!

Conclusions

In this section we've learnt how to create a subflow and invoke it from within our flow.

Having considered subflows and how to use them, let's now turn our attention to other flows and how to call one flow from another in: DSL 10 - Calling other flows (Part One - Within a model)

DSL 8 - Versioning

DSL 10 - Calling other flows (Part One - Within a model)

DSL 10 - Calling Other Flows (Part One - Within a Model)

Getting Started

The tutorial step uses the add_subflow solution of the project as it's starting point.

If at anytime you want to see the solution to this step, this can be found on the add_flow solution

Scenario

So far to date we have only considered a single flow, which takes in a pacs.008 and performs some actions on it. We could consider this to be an "execution" flow. Let's now consider how we may do "initiation" and create a second flow that will:

Receive a pain.001

Convert the pain.001 to a pacs.008

Forward the pacs.008 to the execution flow

Determine the outcome of the execution flow and terminate appropriately.

So the key thing to consider here is how we communicate flow to flow. In this step we will only look at flows that are deployed together - namely when two flows are both defined in the same model. We will look at the alternative scenario in part two of calling other flows.

Whilst this section discusses how to use the DSL itself to setup flow to flow, it is important to realise that in essence flow to flow is effectively calling from one domain to another and as such it would be just as valid (and possible) to simply model the flow interaction using external domains.

DSL Setup

Adding the Initiation Flow

Let's add a new flow by right clicking on the model and selecting New v2Flo Flow. Then in the flow edit we'll set:

A name of "InitiationFlow"

A description of "Initiates payment processing"

A new state to represent flow rejection called "Rejected". This should have a REJECTED global code.

When done we should look like:

Now let's move down to the initiation behaviour, as per our requirement we need to receive a pain001. The first thing we'll do here is use the pre-packaged initiation types. To do these we have to import them into our model.

You'll do this by right clicking on our model and clicking "Model Properties" (or just using the ALT+ENTER shortcut). This will bring up the model properties view:

From here we'll click the "+" and then enter "ISOInitiationTypes" (note that as you type it will remove non matching options so you should be able to select it before entering the whole string) and click ok.

Now if we go to our Initiation Behaviour on the new flow, we should have the option to select "Payment Initiation" so let's do that.

If you can't find a business data element that you are interested in, instead of adding the model like we did above, from the selection box press "CTRL + R". Then click the box for "include stub and non project models" (or press CTRL+R) again and you should be able to search for your business data library that contains the element you want.

Integrating the Flows

Now let's integrate the two flows. To do this, we need to use another pseudo state just like in DSL 5 Using a decision. In this case it's a "Flow State". So in the "Move To State" box, select "Create Flow State".

Let's call our Flow State "Call Execution".

Then in the perform action state, we'll select "Call Flow" and then select our ipftutorial flow. When done it should look like this:

Now we can see that the calling of the flow is underlined as an error. If you validate the flow (ALT + ENTER then Validate Flow) we'll find:

Let's work through each of these, firstly "Called Flow requires missing data: Customer Credit Transfer". This is because our execution flow needs to be given a credit transfer object, but at the moment the initiation flow doesn't have one to give. To fix this, we'll create a new aggregate function to map from the pain.001 to the pacs.008. If you need a reminder of aggregate functions, then review DSL 6 - Using an aggregate function. Try and add this new aggregate function now and when ready, the solution is below:

If we re-validate our flow, then we should no longer see the data error.

The state issues are due to the fact we're not handling the output from Ipftutorialflow. Let's go back to our initiation flow and add our Event Behaviour to fix this.

You'll start by adding the behaviour when we're in our "Call Execution" state. The event is the special bit here we need to select a special "On Flow State" event, and then select the "Complete" state from the execution flow. Finally, we'll move to the "Complete" state from the initiation flow. Putting that together we have:

Now we can do the same for the Rejected and Timed Out state. For now, we'll just send both to Rejected.

Viewing the Flows

That's all our DSL changes complete, but before moving on let's consider the graph for our new initiation flow. As normal, let's open the Flow Viewer (Tools > Open Flow Viewer`):

Here we can see that our child flow is represented by the green box for the call to the ipf tutorial flow. Note that unlike with the subflow, we cannot expand the execution block as that is a separate flow and is not considered an embedded part of this flow.

That's all our DSL work done, and we have completed the setup of the flow.

Java Implementation

Let's now switch back to Intellij and look at how we plug this into our implementation code. As normal we'll start by running a build from a terminal window:

mvn clean install

This will successfully generate all of our DSL related components.

Previously we were firing our requests directly into the ipf tutorial flow, whereas now we want to call the initiation flow. So let's change "InitiateIpftutorialflowInput" to "InitiateInitiationflowInput" (in the ipftutorialsolution/app/controller/InitiationController.java). In doing this, the expect type of data supplied changes from the customer credit transfer to the payment initiation object. So we'll also need to change that line, for now we can use the Pain001Generator from within the ifptutorial-app.

Try and do this now and the solution is below:

```
var samplePain001 = Pain001Generator.generate();

 if (request != null && request.getValue() != null) {


samplePain001.getPmtInf().get(0).getCdtTrfTxInf().get(0).getAmt().getInstdAmt().setValue(request.getValue());

}

return Mono.fromCompletionStage(IpftutorialmodelDomain.initiation().handle(new InitiateInitiationFlowInput.Builder(entityId)

      .withProcessingContext(ProcessingContext.builder()

            .unitOfWorkId(unitOfWorkId)

            .clientRequestId(clientRequestId)

            .build())

      .withPaymentInitiation(samplePain001)

      .build()).thenApply(done -> InitiationResponse.builder().requestId(clientRequestId).uowId(unitOfWorkId).aggregateId(done.getAggregateId()).build()));
```

Make sure you remember to clean up the imports (CTRL+SHIFT+O or/ CTRL+ALT+O on Windows).

Next we have to add the aggregate function for the initiation flow, now let's revisit what we were trying to achieve with this function. We need something to map from the pain.001 to the pacs.008.

Here you could use any approach required to perform the mapping. In our case, we're going to use a pre-built library that provides us with a pain.001 to pacs.008 mapping. So let's add that now.

First we need to add the dependency to the pom.xml within the ipf-tutorial-app module (ipf-tutorial/ipf-tutorial-app/pom.xml).:

```
<dependency>
   <groupId>com.iconsolutions.ipf.core.mapper</groupId>
   <artifactId>mapping-library-spring</artifactId>
</dependency>
```

This will load in a Spring based implementation of various ISO to ISO mappings (). Have a look at the IsoMappingService class that has been brought in and you can see it has a method to apply a mapping from pain.001 to pacs.008:

```
public FIToFICustomerCreditTransfer mapPain001toPacs008(CustomerCreditTransferInitiation initiation) {
   return (FIToFICustomerCreditTransfer)this.transformationService.mapThenEnrichWithDefault(initiation, FIToFICustomerCreditTransfer.class);
}
```

Here we can see that it is calling a transformation service to apply the mapping. This is using Icon's "Mapping Framework" to perform the mapping. The mapping framework can be used to build your own custom mappings, and this will be covered later in this series. Note that this implementation assumes as 1:1 ratio between pain.001 and pacs.008.

As we have chosen to use the spring based implementation, an instance of the IsoMappingService will be automatically injected into the spring context for us, so we simply need to add it to our tutorial class and then use it to provide the implementation of our aggregate function.

Try and add this as now just as we did for the tutorial flow's aggregate function port before. When ready, the solution is below.

```
@Bean
public IpftutorialmodelDomain init(ActorSystem actorSystem, IsoMappingService mappingService, SchedulerPort schedulerAdapter) {
   // All adapters should be added to the domain model
   return new IpftutorialmodelDomain.Builder(actorSystem)
         .withTutorialDomainFunctionLibraryAdapter(input -> CompletableFuture.completedStage(new DuplicateCheckResponseInput.Builder(input.getId(), AcceptOrRejectCodes.Accepted).build()))
         .withAccountingSystemActionAdapter(new SampleAccountingSystemActionAdapter())
         .withFraudSystemActionAdapter(new FraudSystemActionAdapter())
         .withDecisionLibraryAdapter(input ->
```

```
input.getCustomerCreditTransfer().getCdtTrfTxInf().get(0).getIntrBkSttlmAmt().getValue().compareTo(BigDecimal.TEN)>0 ?

                RunFraudCheckDecisionOutcomes.FRAUDREQUIRED :
RunFraudCheckDecisionOutcomes.SKIPFRAUD)

        .withIpftutorialflowV1AggregateFunctionAdapter(input -> new
ExtractCreditorAccountForFlowIpftutorialflowV1AggregateFunctionOutput(input.getCustomerCredit
Transfer().getCdtTrfTxInf().get(0).getCdtrAcct()))

        .withIpftutorialflowV2AggregateFunctionAdapter(input -> new
ExtractCreditorAccountForFlowIpftutorialflowV2AggregateFunctionOutput(input.getCustomerCredit
Transfer().getCdtTrfTxInf().get(0).getCdtrAcct()))

        .withInitiationFlowAggregateFunctionAdapter(parameters -> new
MapPain001ToPacs008ForFlowInitiationFlowAggregateFunctionOutput(mappingService.mapPain00
1toPacs008(parameters.getPaymentInitiation())))

        .withSchedulerAdapter(schedulerAdapter)

        .withCSMServiceActionAdapter(new SampleCSMServiceActionAdapter())

        .withSanctionsSystemActionAdapter(new SampleSanctionsSystemActionAdapter())

        .build();

}
```

That's it, now when our flow calls the aggregate function it will be returned a fully converted pacs.008.

Checking our solution

As normal let's now check out solution works. Start up the application as previously (instructions are available in Reviewing the initial application if you need a refresher!)

And we fire in a payment:

```
curl -X POST localhost:8080/submit | jq
```

And this time we'll notice our aggregate id has changed:

```
{

  "requestId": "1a53c51c-c96e-4786-9f3f-d0d91f80b973",

  "uowId": "ba0b5c6c-855b-41ef-98a7-9b0ee121e6da",

  "aggregateId": "InitiationFlow|176a4e23-6299-49be-89bf-891ca12740de"

}
```

We can see here that it's the initiation flow that's being hit

Then as normal, if we now bring up the payment in the Developer GUI and bring up the flow view (search by unit of work id, click view) and we should see:

This is a really important moment to understand. We've brought back here all the flows associated to our unit of work id. And in doing so we're now seeing two flows, rather than just our one.

Firstly we have the initiation flow, let's look at its graph:

So we can see here that it's calling the execution flow AND receiving the completion response back from it. So our main execution flow must have completed correctly. We can validate that by clicking on the graph for the ipftutorialv2 flow and see:

So as we expected it has completed successfully too.

Its also worth just checking the "Domain Events" where you can observe the Events and the Process Flow from which those events originated.

Conclusions

In this section we have successfully:

Created an initiation flow

Added the integration between the initiation and integration flow

Implemented the new adapters

Deployed and tested the application

Seen that the two flows are linked via the unit of work id.

Note that we have restricted this for now to flow's within the same model, we'll later look at how to do cross-model flows.

DSL 9 - Using Subflows

DSL 11 - Using additional events

DSL 11 - Using additional events

Getting Started

The tutorial step uses the add_flow solution of the project as it's starting point.

If at anytime you want to see the solution to this step, this can be found on the raise_additional_event solution!

What is an Additional Event?

An Additional Event is simply an ability to generate an event from within the IPF processing. This is usually used as a method of adding descriptive information into the event history, but can also be used for performing conditional processing logic. In fact all the configuration we set up in could have been performed by using additional events.

In our scenario, when you look at the event history for an execution flow you can see:

```
[
  "Flow Initiated",
```

"Duplicate Check Passed",

"Account Validation Passed",

"In Sanctions Sanctions Passed",

"Run Fraud Check SKIP_FRAUD",

"Clear and Settle Passed"

]

Here it is not immediately obvious that our flow has completed, it appears that we have only reached a clear and settle passed stage.

So to resolve this we will make the application raise an additional event to clearly show from the event history that we have completed.

DSL Setup

Adding the additional event

Firstly, let's remind ourselves of the DSL logic we currently have for our Clear and Settle Passed event, we see this in the Event Behaviour:

Here instead of immediately completing, we want to raise our additional event. However, once we reach a terminal state the flow has finished and no other actions are allowed. For that reason, we'll need a new State and a new state transition.

Let's add a Completing state like this:

You'll also need to define our additional Event Definition, call this Flow Complete and it gets added to the event definition of our flow just like anything else:

Note here we haven't supplied any business data on the event, but we could do that. That data would then be populated from data contained from other events in our flow.

To actually raise our additional event we need to change our Event Behaviour (for Cleared And Settling) to move to the new Completing state (rather than Complete) and then raising an additional event. So here in the perform action box we need to choose to raise an additional event:

Then we can just select the event we want to raise, so in our case our new Flow Complete event.

When complete the Event Behaviour should look like this:

Finally, we need to transition to the Complete step again. This is simply a case of transition from Completing to Complete on receipt of our new Flow Complete event. So let's add a new event behaviour to do this:

That's it, our DSL work is complete. Let's also look at the graph:

Java Implementation

The good news here is that there is no implementation required when adding additional events, it is all dealt with by the generation.

Checking our solution

As normal let’s now check out solution works. Start up the application as previously (instructions are available in Reviewing the initial application if you need a refresher!)

Then we can fire in a payment

curl -X POST localhost:8080/submit | jq

Now if we now bring up the payment in the Developer GUI and bring up the domain events view (search by unit of work id, click view, click domain events) and we should see:

Here we can see that there is the new Flow Complete event present for the ipftutorialflowv2 process and hence we’ve shown that our additional event is correctly being fired.

If you look at the graph for the ipftutorialv2 flow you can see the extra event and transition:

Conclusions

In this section we’ve learnt how to use additional events.

DSL 10 - Calling other flows (Part One - Within a model)

DSL 12 - Using custom business data

DSL 12 - Using custom business data

Getting Started

The tutorial step uses the "raise_additional_event" solution of the project as it’s starting point.

If at anytime you want to see the solution to this step, this can be found on the "add_custom_types" solution!

What is custom business data?

Here by custom business data we mean the types of data we can create in our business data libraries. If you remember, the current library we set up we had:

So here we were using the predefined "CashAccount" data type that comes from the Icon data model. What if we wanted to use our own bespoke types?

To demonstrate this, we’re going to create a new bean and add this to a new business data element to show how it can be used within the DSL.

DSL Setup

Defining the bean

First let’s create our custom bean - this is going to be placed in the domain-root/external-libraries project. You’ll create a bean as follows:

```
package com.iconsolutions.ipf.tutorial.external;

import lombok.AllArgsConstructor;

import lombok.Builder;

import lombok.Data;
```

```
import lombok.NoArgsConstructor;

import java.io.Serializable;

@Data

@AllArgsConstructor

@NoArgsConstructor

@Builder

public class IpfTutorialBean implements Serializable {

   private String stringField;

   private int intField;

}
```

So that's our simple object, note we're using lombok here as a shorthand to setting up the appropriate class characteristics. You can add the dependency for lombok by adding it to the pom.xml:

```
<dependency>

   <groupId>org.projectlombok</groupId>

   <artifactId>lombok</artifactId>

</dependency>
```

Important!

A key restriction here is that as the data points supplied need to be stored on the events, they MUST be serializable. Any class that isn't won't be available in the dropdown!

Now that we've constructed our class, we need to make sure it's been built and added to the mps setup, to do this we simply need to rebuild the project:

```
mvn clean install
```

Adding the custom data

Let's add a new business data element to our existing Business Data Library. You'll give it a name of "Example Data" and a description of "Simple example data".

Now in the type select try and choose "IpfTutorialBean". You should find it's not present.

To make it available, we simply press CTRL+R whilst on the selection box to bring up the search input. We then tick the checkbox at the top:

Note pressing CTRL+R for the second time will tick the checkbox too!

Top Tip

If you're having trouble seeing your new class, try File Invalidate Caches and then try again.

Search for the "IpfTutorialBean" and double click to add.

Then simply type "IpfTutorialBean" in the Data Type and should now be able to see and select our object. Once done, our business data element should look like:

For the purposes of testing, let's add our sample data point to our the Initiation Behaviour of our InitiationFlow:

Finally it's good to look in the project view, if we scroll to the bottom of our model we can see:

So here it show's us all the classes that have been included in this way.

That's us done, we've added a custom business data element and used it to sent it in on initiation.

Java Implementation

Updating the initiation

To test this, we'll simply add the new object into our initiation call. We will need to add the data point to our initiation controller, first we'll need to rebuild to pick up our new changes:

mvn clean install

And then once built we update our initiation controller to send in a new bean:

```
return Mono.fromCompletionStage(IpftutorialmodelDomain.initiation().handle(new
InitiateInitiationFlowInput.Builder(entityId)
    .withProcessingContext(ProcessingContext.builder()
        .unitOfWorkId(unitOfWorkId)
        .clientRequestId(clientRequestId)
        .build())
    .withPaymentInitiation(samplePain001)
    .withExampleData(IpfTutorialBean.builder().intField(7).stringField("Test").build())
    .build()).thenApply(done ->
InitiationResponse.builder().requestId(clientRequestId).uowId(unitOfWorkId).aggregateId(done.getA
ggregateId()).build()));
```

Checking our solution

As normal let's now check out solution works. Start up the application as previously (instructions are available in Reviewing the initial application if you need a refresher!)

Then as normal we can send in a payment

```
curl -X POST localhost:8080/submit | jq
```

This time let's look at the different events that have been received. If we now bring up the payment in the Developer GUI and bring up the flow view (search by unit of work id, click view, click domain events) and we should see the normal set of Domain Events. However this time if we click to see the body of the initiation flow's "Flow Initiated" event we'll see that it now has the new example data object available at the bottom of the event definition:

Extending beyond simple classes

Adding your own libraries

We can also add in any Maven dependency we want to give access to the beans within it. This can be done simply by adding the dependency into the external-library dependencies pom. Go ahead and give this a go for a library of your choice and see if you can make it appear for you in MPS!

Branching beyond the "external-libraries" module.

The original tutorial project supplied us with a module called "Extended libraries" (under domain-root), this module is responsible for bringing in external dependencies. When you create your own project's later using the archetype this folder will still be built for you. The key here is that this module uses Maven shading to collapse itself and any dependencies, so that MPS can find the files on the class path. We can see this on the pom of our module:

```
<!-- Shade everything, Maven still is the source of truth for the versions, but we can get a
    Smoother MPS experience if it "sees" a single library of dependencies
-->
<build>
 <plugins>
  <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-shade-plugin</artifactId>
   <version>3.2.4</version>
   <configuration>
    <createDependencyReducedPom>false</createDependencyReducedPom>
    <shadedArtifactAttached>true</shadedArtifactAttached>
    <shadedClassifierName>shaded</shadedClassifierName>
   </configuration>
   <executions>
    <execution>
     <phase>package</phase>
     <goals>
      <goal>shade</goal>
     </goals>
    </execution>
   </executions>
  </plugin>
```

</plugins>

</build>

Then the external library is referenced in the MPS model for the project so it can be looked up. We can do this to any module we wish.

Conclusions

In this section we looked at:

Creating our own java types and importing them as custom business data elements.

Reusing other external java types by importing them through Maven dependencies.

Now having configured looked at reuse of java types, let's look at the next stage of reuse - DSL 13 - Using shared concepts (Part One - Within a Solution)

DSL 11 - Using additional events

DSL 13 - Using shared concepts (Part One - Within a Solution)

DSL 13 - Using shared concepts (Part One - Within a Solution)

Getting Started

The tutorial step uses the "add_custom_types" solution of the project as it's starting point.

If at anytime you want to see the solution to this step, this can be found on the "shared_models_one" solution!

What is a shared concept?

In DSL 9 - Using Subflows we looked at setting up a subflow as a way of sharing logic. By creating the subflow, we allowed ourselves to use it in several places within our flow. However, what if we want to go one step further and reuse a subflow not just within the flows within our model, but within other models. This is where shared concepts come into play. It allows us to create modules that are themselves re-usable components that can be pulled into different projects.

You'll do this by extracting the subflow out of our existing model and moving it into it's own sanctions model and then reuse it within the origin tutorial model.

DSL Setup

Extracting the subflow logic

First of all we'll need a brand new model in which to house our sanctions logic. To do this we select the solution in the project navigator and then right click and select New > Model. You'll call our model com.icon.sanctions.

Press ok to create our model.

We should then be prompted to select our dependencies:

For now all we want to add is our actual flo-lang language. So let's click on the "Used Languages" tab, then select the "+" symbol and add our v2Flo language:

After selecting it we should see:

That’s our model all set up so press "OK" to create and you should now see a second model appear in the navigator:

Let’s now think about what makes up our Sanctions logic, we have:

Sanctions Subflow

Sanctions System

Sanctions Response Codes

First let’s move our response codes. For this, we’ll first need to create a new response code library to house the sanctions codes within our new 'sanctions' model. For that, we simply right click on the Sanctions model and then go New v2Flo Response Code Library.

Now we need to move our sanctions response codes into our new library. So let’s open our response code library within the main tutorial model (ipftutorialmodel) and then click on the sanctions response codes and press F6 (or alternatively right click and select Refactor > Move Nodes) and you should get the move nodes popup:

Here it’s showing us out sanctions system is in the ipftutorialmodel. However we now want it in our sanctions model so we simply expand the sanctions model, select our new response code library and click "Refactor", and again "Refactor" on the pop-up.

Then we’ll confirm the refactor on the review screen and we should see the refactoring view appear:

Here we simply tell MPS to do the refactor and we should see our sanctions response codes now sitting in the our new library:

That’s it we’ve refactored our response codes into a different model.

Let’s now repeat the process to move the sanctions subflow into our sanctions model too. For these as they live as top level elements on their own (i.e. they don’t exist within a library) we can simply move them directly. Do this now and then in the navigator we should see:

And we have successfully refactored all of our sanctions components into the new model.

The key thing to understand here is that the sanctions logic now exists in a different model, but that model has been shared with the current model as a dependency. MPS did this for us when we did the refactor, but it is perfectly possible to do this manually too. If we click on the ipftutorialmodel in the navigator and press "ALT+ENTER" (or right click then "Model Properties" we’ll find:

If you note here, the bottom entry shows our sanctions model. If we wanted to we could remove it (using the - sign) and then re-add it (using the +) and searching for our model name. It’s this ability to share a model with a different model that allows us to provide reusable components across many flows and solutions.

Top Tip

If you have a model that you know is only going to provide supporting models (i.e. it does not contain any flows) then you can slightly speed up the generation of your solution by clicking on the model, pressing ALT + ENTER and then going to the "Advanced" tab before checking the "Do Not Generate" box. It won’t cause issue’s if you don’t though!

That’s it from a DSL perspective.

Java Implementation

The good news here is that there is nothing we have to do from an implementation viewpoint to adapt to our newly refactored model. So hence we could just check the flow just like we have in previous steps.

Checking our solution

As normal let’s now check out solution works. Start up the application as previously (instructions are available in Reviewing the initial application if you need a refresher!)

And then we could send in a payment:

curl -X POST localhost:8080/submit | jq

And if we bring up the payment in the Developer GUI and look at the graph of our tutorial flow (search by unit of work id, click view, click ipf tutorial flow, click view graph) then we see:

If we compare this to the graph of DSL 9 - Using Subflows, we can see that everything is the same as it was and we have successfully extracted out our subflow to a different model.

Next up, we’ll take this a step further and look at moving our subflow into a different solution in: DSL 14 - Using shared concepts (Part Two - Across Solutions)

DSL 12 - Using custom business data

DSL 14 - Using shared concepts (Part Two - Across Solutions)

DSL 14 - Using shared concepts (Part Two - Across Solutions)

Getting Started

The tutorial step uses the "shared_models_one" solution of the project as it’s starting point.

If at anytime you want to see the solution to this step, this can be found on the "shared_models_two" solution!

A Quick Recap

In DSL 13 - Using shared concepts (Part One - Within a Solution), we extracted our sanctions logic into a separate model and imported that model so that our flow was still able to work. In doing so we enabled the sanctions logic to be reused across multiple models. However, the limiting factor at the point was that we were doing this within a single overall solution.

This section looks at how we’d do this when we want to use the sanctions code across many different IPF applications (in separate 'Solutions'). We can consider this as per the following diagram:

To do this we’re going to swap out our current sanctions implementation that we’ve created, with a different sanctions implementation.

The sanctions implementation we’re going to use is available here:

git clone bitbucket.iconsolutions.com/scm/ipfv/ipf-tutorial-sanctions.git]

DSL Setup

Creating a New Solution

First of all lets create a new solution, to do this we right click on our project and choose New > Solution

Then we enter the name as "Sanctions".

We should now see the sanctions solution appear alongside our original solution in the navigator.

Next we're going to move our Sanctions model from the main tutorial solution to the new Sanctions solution. To do this, we select the Sanctions Model in the navigator pane and the press "F6" (or right clickt then Refactor Move Model) to see:

And here we simply select our new Sanctions solution and confirm the refactor.

Then we should be able to see our Sanctions model has moved over to the new solution.

This is largely so far just as we did on the previous step when looking at in-model reuse. However there is one key extra step we need to take here to tell the build that it now needs to include both models. To do this, we need to open the build solution ("Ipftutorialsolution"):

And then you should see it highlighting an error:

To resolve this we simply need to tell it to build our new solution as well. To do that selection the end of the 'solution' line, press return and you should see a new entry setup appear (in the Ipftutorialsolution/Ipftutorialsolution.msd):

Select the type as "solution" to see:

Here we can enter the name for the entry, we'll call it "Sanctions" and enter the load path which is the path to the underlying mps solution file. We can do this by entering a / after the "load from ." and then navigating down our path. It will be similar to the original one above and go down to the Sanctions.msd file. Once complete it should look like:

If the red underlines appear, we may need to reload the solutions from disk to clean up. To do that, simply press "ALT+ENTER" on the underlined error and you should see:

And now press "Reload Modules From Disk". Once that's done we can regenerate the build module (right click in project view > regenerate).

That's all our DSL work done, we have now extracted our sanctions logic into a new solution.

Java Implementation

As with the last step, there is no java implementation work to do as MPS has taken care of everything for us.

Checking our Solution

As normal let's now check out solution works. Start up the application as previously (instructions are available in Reviewing the initial application if you need a refresher!)

And then we could send in a payment:

curl -X POST localhost:8080/submit | jq

And if we bring up the payment in the Developer GUI and look at the graph of our tutorial flow (search by unit of work id, click view, click ipf tutorial flow, click view graph) then we see:

If we compare this to the graph of DSL 9 - Using Subflows, we can see that everything is the same as it was and we have successfully extracted out our subflow to a different Solution now.

Conclusions

In this section we have learnt how to extract components into a separate reusable Solution (within the same project) and to include the model (from the new Solution) within our main solution.

DSL 13 - Using shared concepts (Part One - Within a Solution)

DSL 15 - Using shared concepts (Part Three - remote models)

DSL 15 - Using shared concepts (Part Three - remote models)

Getting Started

The tutorial step uses the "shared_models_two" solution of the project as it's starting point.

If at anytime you want to see the solution to this step, this can be found on the "shared_models_three" solution!

A Quick Recap

In DSL 14 - Using shared concepts (Part Two - Across Solutions), we extracted our sanctions logic into a separate solution and imported that so that our flow was still able to work. In doing so we enabled the sanctions logic to be reused across multiple solutions.

In practice however most projects will be set up as a service providing a single solution. Therefore, we now need to look at being able to import models that are contained in separate reusable solutions.

Dependencies

Before we start, we need to clone a new application. This is available here: IPF Tutorial Sanctions, to clone it:

Getting Sanctions Module

git clone ssh://git@bitbucket.iconsolutions.com:7999/ipfv/ipf-tutorial-sanctions.git

Once we've cloned the repository, let's build it:

mvn clean install

Now let's open our sanctions project in Intellij and have a look at what's inside, it's a little different to our main tutorial project.

So here all we have is our domain-root project. That's because all we're going to use this project for is to provide a reusable MPS project, we're not going to build flows or implement any generated interfaces.

Now let's open it in MPS, we'll see that it contains our sanctions model that we created in our previous steps. Now we're going to use it to replace our existing sanctions model and used this shared component.

How the Sanctions Module was Built

This module was built using the IPF Archetype with the following property set:

-DincludeApplication=n

This generates the project as a reusable MPS solution as described above.

Importantly the MPS module pom.xml enables the zipping of the files for import into any downstream projects:

The build script also looks a little different to a non-reusable module but this is all generated for you, and you should not need to modify this if using the IPF Archetype.

Importing the Module

The importing of the module is done within the plugin sections of the pom within our application.

Let's open our main tutorial project in Intellij and then if you open the pom of the mps folder (domain-root/mps/pom.xml) we'll see there already exists an entry for the maven-dependency plugin as follows:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-dependency-plugin</artifactId>
      <version>3.1.2</version>
      <executions>
        <execution>
          <id>copy</id>
....
```

Now all we need to do is add a section to unpack the sanctions mps module into our application. To do this we add a new execution as follows:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-dependency-plugin</artifactId>
```

```
<version>3.1.2</version>
<executions>
  <execution>
    <id>unpack-sanctions-plugin</id>
    <phase>initialize</phase>
    <goals>
      <goal>unpack</goal>
    </goals>
    <configuration>
      <artifactItems>
        <artifactItem>
          <groupId>com.iconsolutions.ipf.tutorial.sanctions.domain</groupId>
          <artifactId>Sanctionsreuse</artifactId>
          <version>1.0.4</version>
          <type>zip</type>
          <overWrite>true</overWrite>
          <outputDirectory>${plugin_home}</outputDirectory>
        </artifactItem>
      </artifactItems>
    </configuration>
  </execution>
 <execution>
  <id>copy</id>
```

The key bits to notice here are the groupId, artifactId and version - it's just like any other Maven dependency (note that the artifactId here is the solution name).

That's all we need to add, if we wanted to add more solutions, we'd simply add more execute blocks ensuring that for each one we have entered the correct groupId, artifactId and version just as we would for a normal Maven dependency.

Now we've added that we're at just the same stage as we would have been if we'd taken the "shared_concepts_three_initial" solution and skipped this part of the tutorial.

Using the Remote Model

Now that we've configured our application to import the model let's use it. First we'll rebuild the application to make sure the module is correctly pulled in.

mvn clean install

Once complete, open MPS. The first thing we'll do is delete the existing sanctions solution. To do this, we simply click on it in the explorer and press delete. A delete confirmation box should appear:

Click the delete files button as we won't need the underlying files any more and then press "Delete".

You should now see errors appear in the navigator like this:

Updating Model Properties

This is because that main solution holds a reference to the deleted solution that it doesn't know what to do with. We will now replace this with a reference to our new solution. To do this this let's start by clicking on the model and pressing "ALT+ENTER" (or right click followed by "Model Properties". This should bring up the model properties:

We can see here the old sanctions model is highlighted in red, to remove it simply click on it and press the "-" symbol. You'll receive a warning:

This is because there are still elements in the flow that are trying to connect to the sanctions model, that's fine we'll sort those in a minute and so it's safe to click delete anyway (if you wanted, you could click "View Usages" and review those).

Now we need to add the new sanctions model so we press the "+" symbol and type sanctions into the box:

We should see our "Sanctionsreuse" model available, so let's select this and press "OK". The new sanctions model will be added and for now it will be red too, press "Apply" and the red will disappear. Instead it will be a feint grey which means it's imported but not currently used - that's because we need to switch out the calls to the old model with the new one. You'll do that next, so click "OK" to finish.

Now we only have an error showing on the solution level, so let's click on our solution (Ipftutorialsolution) and again press ALT+ENTER to see the properties. You'll see here that we now have both sanctions solutions imported:

That's because adding the new one at model level has automatically associated it at solution level, so here we simply have to select our old Sanctions solution and press the "-" symbol to remove. And Apply.

Updating the Flow

Now that we've set it up to be able to reference our new sanctions model, we need to update our flow to use it. This is really simple, just click on the flow to bring it up in the editor and MPS will automatically do the rest for you!

Once done, right click on the solution in the navigator and rebuild it and everything should work fine.

Cleaning the Build Scripts

We also need to do a bit of build clean up, to both remove the old model and provide the references to the new one when we run our build through Maven.

Firstly, in DSL 14 - Using shared concepts (Part Two - Across Solutions) we added our sanctions solution to the build script so we now have to remove that. To do this we simply open the build project:

And then select the sanctions section and press delete to remove it.

You'll see at this point we still have a red error on our Ipftutorialsolution in the build script. So to fix this we need to add the dependency into the build for our new module. So in the dependencies section add a new line by pressing return:

Let's press CTRL+R to import the new sanctions module (this makes it available for us to add as a dependency):

In our new dependency line we can now add Sanctionsresue by selecting the Sanctionsreuse dependency. Note we will have to fill in the location section too, this can be selected and will be $plugin_home/Sanctionsreuse.

Once complete we should look like:

Finally, we simply need to select our red underlined Ipftutorialsolution and press ALT+ENTER then reload all from disk and the red line will disappear.

Once done, right click on the build solution in the navigator and rebuild it and everything should work fine.

Java Implementation

The hard work in this part of our tutorial series was all done above in the MPS dependencies and build set up. From a java implementation viewpoint there is nothing extra for us to do and everything should already have been taken care of for us. So here, we simply need to rebuild our application:

Checking our Solution

As normal let's now check out solution works. Start up the application as previously (instructions are available in Reviewing the initial application if you need a refresher!)

And then we could send in a payment:

curl -X POST localhost:8080/submit | jq

And if we bring up the payment in the Developer GUI and look at the graph of our tutorial flow (search by unit of work id, click view, click ipf tutorial flow, click view graph) then we see:

If we compare this to the graph of DSL 9 - Using Subflows, we can see that everything is the same as it was and we have successfully extracted our Sanctions subflow to a different model which is completely stand alone from the Ipftutorialmodel.

Conclusions

In this section we have learnt how to reuse a common component within our solution.

DSL 14 - Using shared concepts (Part Two - Across Solutions)

DSL 16 - Dynamic Error Text

DSL 16 - Dynamic Error Text

Getting Started

The tutorial step uses the "shared_models_three" solution of the project as it's starting point.

If at anytime you want to see the solution to this step, this can be found on the add_dynamic_text solution.

Purpose

In DSL 4 - Using an external domain, we introduced the concept of a reason code to be able to enrich our events with knowledge as to the underlying reason for the event being raised. At the time, we only used the name and description fields to describe the reason code. In this section, we'll take this one step further and show how we can dynamically setup our reason code text based on the information in the flow. Therefore, instead of just having a generic error message that is used for all payments, we can make our error message unique to our particular payment and hence provide enriched information.

In our case, we're going to create some reason codes for our fraud response and use them to start providing dynamic error messages. Let's get on and see how this is done.

Using the DSL

Placeholders

The first thing we need to do is introduce a new concept, "placeholders". Placeholders are a way of extracting information out of any given piece of business data (or combination of business data!) into a simple string format that we can use in messaging. The placeholder has two properties:

The name of the placeholder

A list of business data.

You can think of a placeholder as a function that takes in a list of business data elements and returns a string. In the simplest case, the business data may just be a string itself but in more complex cases you may for example wish to extract a specific field on the pacs.008.

Let's create a placeholder to pretend to extract some data from our pacs.008. So we start by adding a new placeholder library by right clicking on our model and selecting New Placeholder Library

This will look and feel like all the other libraries, so lets start by clicking "Add placeholder" and then entering:

a name of "Extract from Pacs008"

a business data element of "Customer Credit Transfer"

When complete, it should look like this:

Now let's use our placeholder to create a dynamic text for our reason code.

Dynamic Text

Let's add a new "Reason code set" to our existing "Reason code library" for our fraud system. You'll add two reason codes, the first will just use a standard description whilst the second will also include the dynamic text definition.

So let’s start by adding two reason codes:

One with a name "Simple Reason" and description "A simple reason without dynamic text"

One with a name "Dynamic Reason" and description "A reason with dynamic text".

So far this is just like we did in our previous tutorial (). Now for the dynamic reason lets add a dynamic text message. We start by pressing return in the text block. Now we can type our message just like any normal text input. So let’s start by typing "This is a dynamic message. The special value is " into the box. Now we want to finish our text by grabbing the value of the "Extract from Pacs008" placeholder we setup. To do this we do this simply by pressing CTRL + SPACE and then selecting it from the dropdown of available terms.

When complete, our reason code set and reason codes should look like:

Finally, we need to add our new Fraud Reason codes into our response definition for our fraud check so that we can send back the reason code.

And that’s it from a DSL viewpoint, now lets turn to our implementation.

Java Implementation

Let’s now switch back to Intellij and look at how we plug this into our implementation code. As normal we’ll start by running a build from a terminal window in the root directory of our project:

mvn clean install

In our DSL work we created a placeholder. We described this as a function that takes some business data and returns a string. That’s exactly what it is! Each placeholder will result in a method on the domain’s "Placeholder Port".

Let’s look at the generated code in /domain-root/domain/target and we should now find the port for defining our placeholders like this:

```
package com.iconsolutions.ipf.tutorial;

import
com.iconsolutions.ipf.payments.domain.clearing_and_settlement.pacs008.FIToFICustomerCreditTra
nsfer;

public interface IpftutorialmodelPlaceholderPort {

  String executeExtractFromPacs008(FIToFICustomerCreditTransfer customerCreditTransfer);

}
```

So here we can see that we have generated a function that takes our FIToFICustomerCreditTransfer and returns a String. This is what we’ll need to implement and define on our configuration set up as normal. For now we’ll do this by implementing a very simple constant mapping that returns the string "Test Mapping". You’ll add this to the tutorial config as below:

```
public IpftutorialmodelDomain init(ActorSystem actorSystem, IsoMappingService mappingService,
SchedulerPort schedulerAdapter) {

   // All adapters should be added to the domain model

   return new IpftutorialmodelDomain.Builder(actorSystem)
```

```
        .withTutorialDomainFunctionLibraryAdapter(input ->
CompletableFuture.completedStage(new DuplicateCheckResponseInput.Builder(input.getId(),
AcceptOrRejectCodes.Accepted).build()))

        .withAccountingSystemActionAdapter(new SampleAccountingSystemActionAdapter())

        .withFraudSystemActionAdapter(new FraudSystemActionAdapter())

        .withDecisionLibraryAdapter(input ->


input.getCustomerCreditTransfer().getCdtTrfTxInf().get(0).getIntrBkSttlmAmt().getValue().compareT
o(BigDecimal.TEN)>0 ?

                RunFraudCheckDecisionOutcomes.FRAUDREQUIRED :
RunFraudCheckDecisionOutcomes.SKIPFRAUD)

        .withIpftutorialflowV1AggregateFunctionAdapter(input -> new
ExtractCreditorAccountForFlowIpftutorialflowV1AggregateFunctionOutput(input.getCustomerCredit
Transfer().getCdtTrfTxInf().get(0).getCdtrAcct()))

        .withIpftutorialflowV2AggregateFunctionAdapter(input -> new
ExtractCreditorAccountForFlowIpftutorialflowV2AggregateFunctionOutput(input.getCustomerCredit
Transfer().getCdtTrfTxInf().get(0).getCdtrAcct()))

        .withInitiationFlowAggregateFunctionAdapter(parameters -> new
MapPain001ToPacs008ForFlowInitiationFlowAggregateFunctionOutput(mappingService.mapPain00
1toPacs008(parameters.getPaymentInitiation())))

        .withSchedulerAdapter(schedulerAdapter)

        .withCSMServiceActionAdapter(new SampleCSMServiceActionAdapter())

        .withSanctionsSystemActionAdapter(new SampleSanctionsSystemActionAdapter())

        .withPlaceholderAdapter(customerCreditTransfer -> "Test Mapping")

        .build();

}
```

The final thing we need to do is update our FraudCheckAdapter to provide the reason codes. To enable our testing we'll set it up so that:

if the payment value > 40, we'll reject with our dynamic reason code.

If the payment value > 30 (and below 40!) , we'll reject with our normal description.

Let's look at the code for that:

```
@Slf4j

public class FraudSystemActionAdapter implements FraudSystemActionPort {

   private Duration duration = Duration.ofMillis(10);

   @Override
```

```
  public CompletionStage<Void> execute(final CheckFraudAction action) {

    log.debug("Received an action of type {} for id {}", action.getActionName().name(),
action.getId());

    if
(action.getCustomerCreditTransfer().getCdtTrfTxInf().get(0).getIntrBkSttlmAmt().getValue().compare
To(new BigDecimal("50")) >= 0) {

      return CompletableFuture.supplyAsync(() -> "delaying response",
CompletableFuture.delayedExecutor(duration.toNanos(), TimeUnit.NANOSECONDS))

          .thenAccept(string -> log.debug("Pretending to timeout the fraud call for aggregate {}",
action.getProcessingContext().getAssociationId()));

    } else if
(action.getCustomerCreditTransfer().getCdtTrfTxInf().get(0).getIntrBkSttlmAmt().getValue().compare
To(new BigDecimal("40")) >= 0) {

      return CompletableFuture.supplyAsync(() -> "delaying response",
CompletableFuture.delayedExecutor(duration.toNanos(), TimeUnit.NANOSECONDS))

          .thenCompose((String string) -> IpftutorialmodelDomain.fraudSystem().handle(

              new FraudCheckResponseInput.Builder(action.getId(),
AcceptOrRejectCodes.Rejected)

                  .withReasonCode(FraudReasonCodes.DynamicReason)

                  .build())

              .thenAccept((Done done) -> log.debug("Sent input of type {} for id {} with result {}",
done.getCommandName(), action.getId(), done.getResult().name())));

    } else if
(action.getCustomerCreditTransfer().getCdtTrfTxInf().get(0).getIntrBkSttlmAmt().getValue().compare
To(new BigDecimal("30")) >= 0) {

      return CompletableFuture.supplyAsync(() -> "delaying response",
CompletableFuture.delayedExecutor(duration.toNanos(), TimeUnit.NANOSECONDS))

          .thenCompose((String string) -> IpftutorialmodelDomain.fraudSystem().handle(

              new FraudCheckResponseInput.Builder(action.getId(),
AcceptOrRejectCodes.Rejected)

                  .withReasonCode(FraudReasonCodes.SimpleReason)

                  .build())

              .thenAccept((Done done) -> log.debug("Sent input of type {} for id {} with result {}",
done.getCommandName(), action.getId(), done.getResult().name())));

    } else {

      return CompletableFuture.supplyAsync(() -> "delaying response",
CompletableFuture.delayedExecutor(duration.toNanos(), TimeUnit.NANOSECONDS))
```

```
                .thenCompose((String string) -> IpftutorialmodelDomain.fraudSystem().handle(new
FraudCheckResponseInput.Builder(action.getId(),
AcceptOrRejectCodes.Accepted).build()).thenAccept((Done done) -> log.debug("Sent input of type {}
for id {} with result {}", done.getCommandName(), action.getId(), done.getResult().name())));

      }

   }

}
```

Checking our solution

As normal let's now check out solution works. Start up the application as previously (instructions are available in Reviewing the initial application if you need a refresher!)

Now let's test our application. You'll start by sending a payment through of 45 USD - this should give us our dynamic error text.

```
curl -X POST localhost:8080/submit -H 'Content-Type: application/json' -d '{"value": "45"}' | jq
```

Then lets bring up the payment in the Developer GUI and look at the domain events (search by unit of work id, click view, click domain events) then we see:

Now if we click to see the body of our Fraud Check Failed event, we'll see:

And we can see that our reason code has been generated, pulling in the value from our placeholder function.

However, if we repeat the process for a payment of say 35USD, instead of using the dynamic text we'll revert to just using the description provided:

That's everything working.

This is obviously a very simple implementation with some hardcoded text, but you can provide an implementation of the port (i.e. the IpftutorialmodelPlaceholderPort interface) which accesses any of the payment data;

```
public class PlaceHolderAdapter implements IpftutorialmodelPlaceholderPort {

   @Override

   public String executeExtractFromPacs008(FIToFICustomerCreditTransfer customerCreditTransfer) {

      return "failed for amount " +
customerCreditTransfer.getCdtTrfTxInf().get(0).getIntrBkSttlmAmt().getValue().toString();

   }

}
```

Conclusions

In this section we learnt how we can use placeholders to provide enrichment information within error text.

DSL 15 - Using shared concepts (Part Three - remote models)

CON1 - Adding payment initiation

CON1 - Adding Payment Initiation

Getting Started

The tutorial step uses the add_dynamic_text solution of the project as it's starting point.

If at anytime you want to see the solution to this step, this can be found on the add_payment_init solution.

In the DSL tutorial, we built an application that uses the Icon Payment DSL to build up a flow. To initiate this flow, our sample application had a simple rest controller that allowed us to initiate the flow. We were able to send in certain key values such as the payment value to help us test various different conditions within our flow. However in the real world, these payment instructions would be coming from an external source.

In this section, we're going to use an existing test module - the "Sample Payment Initiator" as the external source. This is a simple application we can use to test with it has a few key properties:

It provides a simple simulator that allows us to generate pain001's, providing an interface to set some of the key values.

The application can be used with different protocols (Kafka, JMS)

The applications comes with a set of pre-packaged "connectors". These are the client components, built using Icon's connector framework that allow quick and easy integration between the main IPF app and the payment initiation sim.

In this tutorial, we're going to use the Kafka version of the payment simulator.

Let's get going and set everything up so that we can start sending messages to our IPF application from the Payment Initiation Simulator.

A Quick Recap

Let's do a quick recap on the existing flow, the key one here is the initiation flow and it's initiation behaviour:

So the key thing to note here is that we're sending in a payment initiation (pain001) into start the flow. We won't worry about the example data object here as it was just used as a method of illustrating custom types.

Now if we remember, when generated this will create a new method on the domain's initiation controller that allows us to make flow initiation requests. In the current flow, we do this within the controller of the main ipf-tutorial-app application. Let's remind ourselves of that code (taken from InitiationController):

```
Mono.fromCompletionStage(IpftutorialmodelDomain.initiation().handle(new
InitiateInitiationFlowInput.Builder(entityId)
    .withProcessingContext(ProcessingContext.builder()
        .unitOfWorkId(unitOfWorkId)
```

```
            .clientRequestId(clientRequestId)
            .build())
        .withPaymentInitiation(samplePain001)
        .withExampleData(IpfTutorialBean.builder().intField(7).stringField("Test").build())
        .build()).thenApply(done ->
InitiationResponse.builder().requestId(clientRequestId).uowId(unitOfWorkId).aggregateId(done.getAggregateId()).build()));
```

So the key here is that we are using the sample pain001 we generated as our payment initiation data when we construct the initiation input. You'll do something very similar when setting up to use the payment initiation simulator.

Adding the Connector

First we need to add the dependency to the sample simulator we will use for payment initiator (add this to the "ipf-tutorial-app" pom.xml). This is here:

```
<dependency>
  <groupId>com.iconsolutions.ipf.sample.samplesystems</groupId>
  <artifactId>payment-initiation-connector-kafka</artifactId>
</dependency>
```

At the time of writing, the sample systems version is 2.0.41, and we need to add this to both the dependency management section of the root pom (adding the version), and the actual dependencies of the "ipf-tutorial-app" project.

Note also that we've chosen the Kafka implementation here as our protocol.

The Client Adapter

When the payment initiation simulator runs, it will post out messages to the relevant Kafka topic. We're going to use the pre-packaged Icon connector in our application and that will read off that topic and processes messages. For this, the connector provides an interface, the PaymentInitiationClientAdapter that we'll need to provide an implementation of this in ipf-tutorial-app. Our implementation will need to provide the same core logic that the current initiation controller does, i.e. take the pain001 and forward it onto the domain's initiation methods.

Let's start by looking at the definition of this interface:

```
public interface PaymentInitiationClientAdapter {
   ProcessingContext determineContextFor(PaymentInitiationRequest request);
   CompletionStage<Void> handle(ReceivingContext context, PaymentInitiationRequest request);
}
```

As you can see there are two methods it provides:

determineContextFor - This is the chance to provide upfront processing context, for example the unitOfWorkId which represents an IPF reference used to track all activity against.

handle - This is where we handle inbound messages, and for us we need to pass the message onto the flow.

Let's consider what we want to do in our implementation.

determineContextFor - in our case we're not too worried about id's. But if there was a specific client request id or unit of work id that you required your messages to use this would be where to set it. However, we'll use a generated unit of work id.

handle - this is where we need to take the pain001 out of the request object and pass it into the domain's initiation methods.

See if you can work out how this code would work, once done the solution (to add to the ipf-tutorial-app) is below:

```
@Slf4j

@Service

public class SamplePaymentInitiationAdapter implements PaymentInitiationClientAdapter  {

   @Override

   public ProcessingContext determineContextFor(PaymentInitiationRequest
paymentInitiationRequest) {

      return ProcessingContext.builder().unitOfWorkId(UUID.randomUUID().toString()).build();

   }

   @Override

   public CompletionStage<Void> handle(ReceivingContext receivingContext,
PaymentInitiationRequest paymentInitiationRequest) {

      return IpftutorialmodelDomain.initiation().handle(

            new InitiateInitiationFlowInput.Builder(UUID.randomUUID().toString())

               .withProcessingContext(receivingContext.getProcessingContext())

               .withPaymentInitiation(paymentInitiationRequest.getPayload().getContent())

            .build())

            .thenAccept(done -> log.debug("Completed request"));

   }

}
```

Note here that we use springs "@Service" annotation to wire in the dependency.

That's all our code done, lets build the application:

```
mvn clean install -rf :ipf-tutorial-app
```

Note that we have only changed implementation code and not our flow so there is no need to rebuild the MPS components at this time.

Configuring the Application

Now it's time to set up the configuration, So there's two things not currently in here that we'll need:

Default no-op crypto bean in ipf-tutorial-app

The configuration for our payment initiation connector to talk to kafka.

First, we need to add a default crypto bean to our ipf-tutorial-app (in IpfTutorialConfig), since we are not encrypting the data we will be configuring a NoOpCrypto bean

```
  @Bean
  public Crypto crypto() {
    return new NoopCrypto();
  }
```

Now we need to look at the configuration for the payment initiation connector.

The standard config like the topic's it's going to read from is provided out of the box. It's going to read messages from the PAYMENT_INITIATION_REQUEST topic. So all we need to tell it how to connect to Kafka. We can do this in two ways.

Explicit Configuration

The first approach we can use is to configure the kafka setup explicitly for this connector. To do this, we need to note that the config root for the payment initiation connector is paymentinitiation. Therefore we need to supply our kafka configuration from this root as follows:

```
paymentinitiation.kafka {
 producer.kafka-clients {
  bootstrap.servers = "kafka:9092"
  client.id = "ipf-tutorial-app"
 }
 consumer.kafka-clients {
  bootstrap.servers = "kafka:9092"
  group.id = "ipf-tutorial-app"
 }
}
```

Note that here we are using "kafka:9092" as our bootstrap URL, this is what is expected for the container solution, but if using your own kafka you may need to change it.

Default Configuration

The alternative approach is to use a default configuration. All the IPF connectors will default to using a config root of akka. This means that if we have multiple services all using the same kafka we can define the configuration once and let all our services use that. We're going to assume for this

tutorial that is the case so we'll set up our service like that. You'll make a few tweaks now to give us extra functionality.

```
// default settings for kafka
common-kafka-client-settings {  (1)
  bootstrap.servers = "kafka:9092" (2)
}
akka.kafka {
  producer {
    kafka-clients = ${common-kafka-client-settings} (3)
  }
  consumer {
    kafka-clients = ${common-kafka-client-settings} (4)
  }
}
// end default kafka settings
```

Let's review a few bits of this:

1

We extract our the raw kafka settings (group id, bootstrap servers) into a separate section. This means we can reuse this across different property sets. You can see that in points 3 and 4.

2

As per above this is the kafka bootstrap url and may be different for your environment.

3

Here we reuse our common kafka settings for the producer.

4

Here we reuse our common kafka settings for the consumer.

Running the Application

Docker

Kafka

Firstly, we need to setup a Kafka container. The below provides a sample entry to add to the application.yml for the Kafka setup for use in this tutorial.

```
zookeeper:
  image: zookeeper:latest
```

```
  container_name: zookeeper
  ports:
    - "2181:2181"
kafka:
  image: wurstmeister/kafka:2.13-2.7.1
  container_name: kafka
  depends_on:
    - zookeeper
  ports:
    - "9092:9092"
  environment:
    - KAFKA_BROKER_ID=0
    - KAFKA_AUTO_CREATE_TOPICS_ENABLE=true
    - KAFKA_ZOOKEEPER_CONNECT=zookeeper:2181
    - KAFKA_LOG_RETENTION_MINUTES=10
    - KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR=1
    - KAFKA_OFFSETS_TOPIC_NUM_PARTITIONS=1
    - KAFKA_LISTENERS=PLAINTEXT://:9092
    - KAFKA_ADVERTISED_LISTENERS=PLAINTEXT://kafka:9092
```

So here we're providing a simple Kafka server which will be running on port 9092 and the configuration we supplied to the application in the previous section will connect to this.

Payment Initiation Simulator

Next lets add the payment initiation simulator to our deployment. We can add the following into the application.yml

```
payment-initiation-simulator-kafka:
  image: registry.ipf.iconsolutions.com/sample-systems-payment-initiation-simulator-kafka:2.0.41
  container_name: payment-initiation-simulator-kafka
  ports:
    - "8082:55555"
  volumes:
    - ./config/payment-initiation-simulator-kafka:/payment-initiation-simulator-kafka/conf
  user: "1000:1000"
```

```
  depends_on:
    - kafka
```

Here we'll need to supply our config for the payment initiation sim itself. To do this, we'll add a new directory in the docker/config folder called payment-initiation-simulator-kafka. It requires an application.conf file that contains:

```
common-kafka-client-settings {
  bootstrap.servers = "kafka:9092"
}
connector {
  default-receive-connector {
    manual-start: true
  }
  default-send-connector {
    manual-start = false
    call-timeout = 30s
    queue-size = 50
    max-concurrent-offers = 500
    resiliency-settings {
      minimum-number-of-calls = 1
      max-attempts = 1
      reset-timeout = 1s
      initial-retry-wait-duration = 1s
      backoff-multiplier = 2
    }
  }
  validator-error-handler-dispatcher {
    type = Dispatcher
    executor = "thread-pool-executor"
    thread-pool-executor {
      fixed-pool-size = 4
    }
  }
```

```
}
akka.kafka {
  producer {
    kafka-clients = ${common-kafka-client-settings}
  }
  consumer {
    kafka-clients = ${common-kafka-client-settings}
  }
}
```

Non Docker

Kafka

When not running in docker, we'll need to use your own kafka environment. This should have two topics available:

PAYMENT_INITIATION_RESPONSE

PAYMENT_INITIATION_REQUEST

Next, we need to tell the IPF tutorial application how to connect to Kafka. You'll then need to update the bootstrap config we set up in the previous section to reach your kafka environment.

Payment Initiation Simulator

Details for how to run the payment initiation simulator can be found here: Using the payment initiation simulator.

Testing the Application

As normal let's now check out solution works. Start up the application as previously (instructions are available in Reviewing the initial application if you need a refresher!)

Now the payment initiation simulator will be available (along with the Zookeeper and Kafka setup).

You can access the simulator page here

When brought up, it should look like:

There's a couple of key functions to note here - that we can both:

set load rate - this allows us to send a consistent rate of transactions, e.g. 10TPS, through to our application.

initiate single payment - this allows us to send a single payment with provided values.

The current activity tab we won't use for now but allows the simulator to track responses to payments.

You'll start by calling an individual payment. Click the initiate single payment button and then we'll leave everything as default and click initiate payment. This will fire in a generated payment. A pop up box will appear that will continue to spin, this is because the simulator by default expects that there will be a response to it's message. For now we don't have this so the spinning is the expected behaviour.

Let's bring up the Developer GUI ("IPF Transaction Explorer" used previously) and run a search. We should see a new record having been created a few seconds ago:

Let's view this record and then go to the "messages tab":

Here we can see that IPF has "received" a PaymentInitiationRequest. You could view the body of the message if you want.

Feel free at this stage to try sending some different messages through with different values, or to send a fixed load through.

Conclusions

In this section we've learnt how to setup to connect to an external system that has a pre-packaged connector available.

Next up we'll look in a little more detail at connectors in: CON2 - Writing your own connector

DSL 16 - Dynamic Error Text

CON2 - Writing your own connector (Kafka)

CON2 - Writing your own connector (Kafka)

Getting Started

The tutorial step uses the "add_payment_init" solution of the project as it's starting point.

If at anytime you want to see the solution to this step, this can be found on the "add_kafka" solution!

In CON1 - Adding payment initiation, we connected our sample application with a payment initiator to allow us to receive payment instructions from a remote service. To do this, we used a pre-packaged connector so that all we had to do was implement the logic for processing upon receipt of the message. This time we're going to take a step further back and actually write the connector ourselves.

You'll do this by integrating in with a test sanctions system. This system:

Expects to receive a custom "SanctionsRequest" object.

Will return a custom "SampleEvent" object.

The system can work over either Kakfa or JMS. You'll use kafka in this example!

A few basics

Let's start with a few basics about the connector framework.

In this tutorial we'll consider two types of connectors - "Sending" connectors and "Receiving" connectors. You'll use these to put a message onto a kafka topic (send) and then also to process the response (receive).

As the request / response will be asynchronous, we will need some way of determining which response lines up to which request. We do this using "correlation".

Connectors in themselves are backed by Lightbend's Alpakka framework and use streams to process. To work the require a "transport". A connector transport tells the connector how to communicate with the underlying protocol it is working with. It is therefore perfectly possible to use the same connector and then supply different transports to be able to communicate over different protocols. In this way, we can keep our logic of the processing of the message separated from the details of the underlying protocol.

There is a third type of connector, the "RequestReply" type, but this will be considered in a later tutorial.

Supporting Classes

The first thing we'll do is import the domain definition for the sanctions system. To do this we need to add a dependency into our "ipf-tutorial-app" applications pom.xml:

```
<dependency>
   <artifactId>sanctions-domain</artifactId>
   <groupId>com.iconsolutions.ipf.sample.samplesystems</groupId>
</dependency>
```

Let's look at the key classes we receive from this module. The first is the request object we send to the Sanctions system.

```
@Data
public class SanctionsRequest extends SampleEvent<SanctionsRequestPayload> {
   private Map<String, String> headers;
}
```

So here we have an object that is taking both a set of headers and a request payload. Have a look down the object hierarchy to get a feel for the request we'll need to send, pay particular attention to the filtering request.

If we consider the response side, we can see similarly there is a SanctionsResponse object. Again checking the hierarchy we'll see the core of the response is:

```
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Data
@JsonInclude(JsonInclude.Include.NON_NULL)
```

```
public class FilteringResponse {
   private String status;
   private String additionalInf;
}
```

So the key element here is we will receive back a status.

Next we'll pull in another dependency,

```
<dependency>
   <artifactId>sanctions-mapping</artifactId>
   <groupId>com.iconsolutions.ipf.sample.samplesystems</groupId>
</dependency>
```

This dependency, provides a prepackaged set of mappers (using Icon's mapping framework) that provides a mapping from a pacs008 to a SanctionsRequest object. The key class to look at here is the SanctionsMapper class which provides this method:

```
public SanctionsRequest map(FIToFICustomerCreditTransfer fiToFICustomerCreditTransfer) {
   var filteringRequest =
transformationService.mapThenEnrichWithDefault(fiToFICustomerCreditTransfer,
FilteringRequest.class);
   SanctionsRequest sanctionsRequest = new SanctionsRequest();
   sanctionsRequest.setHeader(HeaderUtils.makeHeader("Sanctions",
fiToFICustomerCreditTransfer.getCdtTrfTxInf().get(0).getPmtId().getTxId()));
   sanctionsRequest.setPayload(new SanctionsRequestPayload(filteringRequest));
   return sanctionsRequest;
}
```

Here we can see we are mapping from the pacs.008 (FIToFICustomerCreditTransfer) to return the SanctionsRequest.

Now we'll use spring to provide to us an instance of the sanctions mapper that we can use. To do this we'll create a new bean within the IpfTutorialConfig class as:

```
@Bean
public SanctionsMapper sanctionsMapper(ObjectMapper objectMapper) {
   return new SanctionsMapper(objectMapper);
}
```

Those are the building blocks we're going to use, so let's get on and start writing our connector.

The Send Connector

You'll start by looking at the send connector, this is the one that will post a Sanctions Request message onto the appropriate topic for the external system to consume.

Firstly, let's add the dependency for the connector framework's kafka implementation.

```
<dependency>
   <groupId>com.iconsolutions.ipf.core.connector</groupId>
   <artifactId>connector-kafka</artifactId>
</dependency>
```

Now we'll create a new configuration class, called "SanctionsConnectorConfiguration" and we'll place it into a new "connectors" package under the existing config package

(Remember to mark your class with the spring @Configuration annotation! You'll also use lombok's @Slf4j annotation to provide access to a logging implementation)

Now let's write our send connector, we'll need to make a few decisions first:

Types - The definition of a send connector is SendConnector<D, T>. In this instance, the D represents the source (domain) object type and the T the target object type. In our connector will take in a FIToFICustomerCreditTransfer and send out a SanctionsRequest object.

Logging - We can provide a logging implementation to a connector. A message logger is any class that implements this simple functional interface:

```
public interface MessageLogger {
   void logMessage(MessageLogEntry var1);
}
```

In our instance, the application comes with a message logger that is part of the data processing. This has everything we need so we'll just reuse it.

Correlation - For correlation we're going to use another Icon provided utility. You'll use Icon's mongo correlation service. So we'll need to bring in that dependency too:

```
<dependency>
   <groupId>com.iconsolutions.ipf.core.connector</groupId>
   <artifactId>connector-correlation-starter-mongodb</artifactId>
</dependency>
```

Mapping - As discussed above, for mapping we'll use the SampleMapper that we've pulled in.

Let's start by thinking at the class level. You'll provide for class variables to support our connectors:

```
private final SanctionsMapper sanctionsMapper;
private final ObjectMapper objectMapper;
private final ClassicActorSystemProvider actorSystem;
private final CorrelationService correlationService;
```

```
private final MessageLogger messageLogger;
```

These are as discussed above, together with the actor system itself.

You'll use spring's dependency injection to provide these for us by using the @AllArgsConstructor annotation.

Let's then create a new method to create the send connector:

```
@Bean(name = "sanctionsSendConnector")

public SendConnector<FIToFICustomerCreditTransfer, SanctionsRequest>
sanctionsSendConnector(ConnectorTransport<SanctionsRequest>
sanctionsSendConnectorTransport) {

   return SendConnectorBuilderHelper.<FIToFICustomerCreditTransfer,
SanctionsRequest>builder("Sanctions", "sanctions.send-connector", actorSystem)

        .withConnectorTransport(sanctionsSendConnectorTransport)

        .withCorrelationIdExtractor(event ->
CorrelationId.of(event.getHeader().getTechnical().getEventId()))

        .withCorrelationService(correlationService)

        .withSendTransportMessageConverter(this::convertToTransport)

        .withDomainToTargetTypeConverter(sanctionsMapper::map)

        .withMessageLogger(messageLogger)

        .build();

}
```

This is important enough to walk through each part in turn.

Firstly we pass into the method a ConnectorTransport<SanctionsRequest> sanctionsSendConnectorTransport. This is an implementation of a connector transport, if you remember our discussion above, the connector transport is used to provide the low level protocol information. For now we're not going to worry about that, hence we'll just pass it into our method.

Now let's take each line in turn and explain what's going on.

The builder construction - it takes three parameters:

The name of the connector - this is particularly useful later when we look at metrics.

The config-root for the connector. This allows us to pass the root path for the connector's properties. All connector properties will start with this variable. This allows us to then build out connector level properties.

The actor system itself.

the transport - obviously here we just use the connector transport we have passed to the method.

the correlation extractor function - this is a function that will provide the connector with a unique id that is used for correlation. The id must be unique and be obtainable from the response message too. Here we use the event id on the header.

the correlation service - as per above, we'll use Icon's mongo backed implementation.

the transport message converter - this is a function that takes the sanctions request and converts it into a transport message for passing down the wire. In our case, we'll make a simple implementation that creates a new TransportMessage with a string representation of our request as the payload.

```
private TransportMessage convertToTransport(SanctionsRequest request) {
   try {
      return new TransportMessage(new MessageHeaders(CryptoHelper.messageHeaders()), objectMapper.writeValueAsString(request));
   } catch (JsonProcessingException e) {
      throw new IconRuntimeException(e);
   }
}
```

the domain to target type converter - this is the function that will map from our domain type (pacs.008) to our target type (SanctionsRequest). As per above, will use the SanctionsMapper's map method to do this.

the message logger - as discussed we'll just use existing logger here.

That's it, that's our very first send connector built from scratch.

The Receive Connector

Now that we've written the code to setup our connector to send a message out to the sanctions system, we'll need one to do the reverse and receive the response when it's ready. So let's again add a new method to our configuration, this time to construct our receive connector:

```
@Bean(name = "sanctionsReceiveConnector")
public ReceiveConnector<SampleEvent> sanctionsReceiveConnector(ReceiveConnectorTransport sanctionsReceiveConnectorTransport) {
   return ReceiveConnectorBuilderHelper.<SampleEvent>builder("SanctionsReceive", "sanctions.receive-connector", actorSystem)
         .withConnectorTransport(sanctionsReceiveConnectorTransport)
         .withCorrelationIdExtractor(event -> CorrelationId.of(event.getHeader().getTechnical().getOriginalEventId()))
         .withCorrelationService(correlationService)
         .withReceiveTransportMessageConverter(message -> sanctionsMapper.convertResponse(message.getPayload().toString()))
         .withReceiveHandler(this::sanctionsReceiveConnector)
```

```
        .withMessageLogger(messageLogger)
        .build();
}
```

As we did last time, let's walk through the key points of this setup.

The builder construction - just as with send connectors we'll pass in the name of the connector, the root config path and the actor system.

the transport - we'll again need the protocol setup, this time for receiving a message. You'll pass this into our method to allow us to deal with protocol details elsewhere.

the correlation extractor function - just like with the send, we need a function that will provide the same id as we had in the send function, but this time extracted from the response event. This time we'll take the "original event id" from the header.

the correlation service - again we'll use Icon's mongo backed one.

the receive transport converter - here we need a function that converts from the raw message recieved into the expected response target type (a sample event). You'll take that from our sanctions mapper functions.

the receive handler - this is the key function. This is a function that takes in the response message together with the context for it (taken from correlation in our case) and requires us to determine what to do with it. In our case, we want to construct a new domain sanctions input and send it into our domain. See if you can write this and when ready compare with our function below:

```
private CompletionStage<Void> sanctionsReceiveConnector(ReceivingContext receivingContext,
SampleEvent sampleEvent) {
   return IpftutorialmodelDomain.sanctionsSystem().handle(new
SanctionsNoHitInput.Builder(receivingContext.getProcessingContext().getAssociationId().getValue()).
build())
        .thenAccept(done -> log.info("Completed {}", done));
}
```

the message logger - and again we'll just use the existing logger.

That's it, that's our entire receive connector written and ready to go.

The Connector Transports

In the definitions for both our receive and send connectors, we passed in a transport definition. Now we need to set these up. For the moment, we're just going to use Kafka, so we'll need those versions. To do this will create a new spring configuration class called "SanctionsTransportConfiguration" which we'll put in a new "transports" package under our connectors package.

The Sending Connector Transport

You'll start with the sending side:

```
@Bean
```

```
public ConnectorTransport<SanctionsRequest>
sanctionSendConnectorTransport(ClassicActorSystemProvider actorSystem) {

   Config config = AlpakkaConfigProvider.getProducerConfig(actorSystem.classicSystem(),
"sanctions");

   StringSerializerProducerConfig producerConfig = new StringSerializerProducerConfig(config);

   return new KafkaConnectorTransport<>(

         "SanctionsSendKAFKA",

         producerConfig.topic(),

         producerConfig.producerSettings(),

         new StringProducerRecordKeyValueProvider(),

         producerConfig.restartSettings(),

         actorSystem

   );

}
```

At this stage we move into specific low level kafka configuration, the details of which lie outside of this tutorial. The key things to note however are:

We give the connector transport a name, again for use in metrics.

We provide the topic which the transport should use.

We provide the "producer" settings, these are standard kafka connection details for sending (producing) messages to a topic.

We provide a restart policy - this is how the transport should behave in the event of a failure.

We provide the actor system.

The Receive Connector Transport

This is very similar to the send:

```
@Bean

public ReceiveConnectorTransport sanctionsReceiveConnectorTransport(ClassicActorSystemProvider
actorSystem) {

   Config config = AlpakkaConfigProvider.getConsumerConfig(actorSystem.classicSystem(),
"sanctions");

   StringDeserializerConsumerConfig consumerConfig = new
StringDeserializerConsumerConfig(config);

   return KafkaAckReceiveConnectorTransport.<String, String>builder()

         .withName("KafkaSanctionsReceiveConnectorTransport")
```

```
        .withTopics(Set.copyOf(consumerConfig.topics()))
        .withConsumerSettings(consumerConfig.consumerSettings())
        .withRestartSettings(consumerConfig.restartSettings())
        .withPartitions(consumerConfig.maxPartitions())
        .withActorSystem(actorSystem)
        .build();
}
```

The main differences are this time we use the kafka configuration for consuming messages from a topic (or potentially multiple topics!). We also have to decide how many partitions we want to apply to our topic. This will help with performance but for now is outside of the scope of this tutorial.

That's our transport defined, all that remains now is setting up the actual configuration to use it.

Using the connector

Now we need to plug our connections into our flow for usage. The first thing to note here is that the receive connector part (method sanctionsReceiveConnector) is already done for us as the handler is sending on the response back to the domain.

So here we only need to worry about the sending side. For this, we simply need to swap out our sample sanctions adapter (SampleSanctionsSystemActionAdapter) with a real one that calls our new send connector's send method. The action we supply to the adapter has all the information we'll need to do that too! See if you can set that up now and when ready the solution is below.

```
@Bean
public IpftutorialmodelDomain init(ActorSystem actorSystem, IsoMappingService mappingService,
SendConnector<FIToFICustomerCreditTransfer, SanctionsRequest> sanctionsConnector) {
  ...
        .withSanctionsSystemActionAdapter(checkSanctions ->
sanctionsConnector.send(checkSanctions.getProcessingContext(),
checkSanctions.getCustomerCreditTransfer()).thenAccept(done -> log.info("Completed sanctions
call: {}", done)))
  ...
```

So here we can see we are injecting in our new send connector and then simply calling the send and logging the result. That's all we need to do.

Configuration

You'll add our configuration into our application configuration file (ipf-tutorial-app/application.conf).

For our connector to work, we need to add a number of things:

The restart settings

The number of partitions to use on send

The group and client ids for kafka to use

We're going to add these into the common akka configuration that we created in the previous tutorial. We could just as easily add all these into the bespoke sanctions block, but then they wouldn't be available to reuse for other connectors.

Restart Settings

Firstly, we need restart settings, ie to tell the connector what to do on failover. Here we'll define the standard set that we're going to use for all our connectors.

```
default-restart-settings {
  min-backoff = 1s
  max-backoff = 5s
  random-factor = 0.25
  max-restarts = 5
  max-restarts-within = 10m
}
```

We are telling our connector that in the event of transport failure, we will attempt to restart upto 5 times with an increasing time between each restart.

To use this, we're going to add it to the default akka block. You'll do this by adding a line under both the consumer and producer settings:

```
restart-settings = ${default-restart-settings}
```

Note the syntax here, how we are able to refer to a complex block from elsewhere in our hocon structure by using the ${…} setup.

Partitions

Partitions allow us to define how many kafka partitions we should setup. In our case we're just going to setup 10 partitions, we do that by adding it to the akka block we need to add a line to the consumer with:

```
max-partitions = 10
```

Client and Group Ids

Finally, we'll add the client and group ids.

For the client id we need to add an entry to the producer config:

```
kafka-clients {
   client.id = ipf-tutorial-client
}
```

And for the group id we need to add an entry to the consumer config:

```
kafka-clients {
```

```
    group.id = ipf-tutorial-group
}
```

Sanctions Configuration

Now we've defined our common parts, because they are defined with the default akka block they are immediately available to us for our sanctions configuration. That means all we now have to do on a sanctions level is provide the explicit custom configuration for the sanctions setup itself. In this case, the only thing is the actual topics which need to be used.

If we remember back to our transport definition (SanctionsTransportConfiguration) we said:

```
Config config = getConsumerConfig(actorSystem.classicSystem(), "sanctions");
```

The "sanctions" parameter tells the connector to look for elements under the "sanctions" prefix in the configuration. So all we need to do is add our topics under this root as follows:

```
sanctions {
  kafka {
    producer {
      topic = SANCTIONS_REQUEST
    }
    consumer {
      topic = SANCTIONS_RESPONSE
    }
  }
}
```

So the first thing to note if we look at our transport definition (SanctionsTransportConfiguration) we said:

Summary

That's everything from our application configuration side complete, just as a recap the full config for the connector should now look like:

```
sanctions {
  kafka {
    producer {
      topic = SANCTIONS_REQUEST
    }
    consumer {
      topic = SANCTIONS_RESPONSE
```

```
    }
  }
}
// default settings for kafka
default-restart-settings {
  min-backoff = 1s
  max-backoff = 5s
  random-factor = 0.25
  max-restarts = 5
  max-restarts-within = 10m
}
common-kafka-client-settings {
  bootstrap.servers = "kafka:9092"
}
akka.kafka {
  producer {
    kafka-clients = ${common-kafka-client-settings}
    restart-settings = ${default-restart-settings}
    kafka-clients {
      client.id = ipf-tutorial-client
    }
  }
  consumer {
    kafka-clients = ${common-kafka-client-settings}
    restart-settings = ${default-restart-settings}
    max-partitions = 10
    kafka-clients {
      group.id = ipf-tutorial-group
    }
  }
}
```

// end default kafka settings

Running the application

To run the application, the first thing we'll need to do is setup the actual sanctions service that we will be talking to.

Docker Setup

If using docker, here's the new entry for our application.yml (docker/application.yml)

```
  sanctions:
    image: registry.ipf.iconsolutions.com/sample-systems-sanctions-simulator-kafka:2.0.21
    container_name: sanctions-sim
    ports:
      - 5010:5005
      - 8088:55555
    environment:
      - SANCTIONS_MODE=normal
      - SANCTIONS_TRANSPORT=kafka
      - SANCTIONS_SIM_ENCRYPTION_ENABLED=FALSE
      - SANCTIONS_SIM_ENCRYPTION_KEYSTORE_PATH=file:///tmp/keys/connector/keystore-pkcs12-aes128.jks
    volumes:
      - ./config/keys:/tmp/keys:ro
      - ./config/sanctions:/sanctions-simulator-kafka/conf
      - ./logs/sanctions:/ipf/logs
    depends_on:
      - kafka
```

Note that the "2.0.21" version provided here is the latest version at the time of writing of this document.

Also note we are not setting any kafka configuration here, this is because the sample set provided above works with the docker environment.

To make things easier we'll also add a logback.xml file for sanctions:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <appender name="FILE"
```

```
        class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>/ipf/logs/sanctions-sim.log</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.FixedWindowRollingPolicy">
      <fileNamePattern>/ipf/logs/sanctions-sim.log.%i</fileNamePattern>
      <minIndex>1</minIndex>
      <maxIndex>20</maxIndex>
    </rollingPolicy>
    <triggeringPolicy
        class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">
      <maxFileSize>50MB</maxFileSize>
    </triggeringPolicy>
    <encoder>
      <pattern>%date{yyyy-MM-dd} %d{HH:mm:ss.SSS} %-5level %X{traceId} %logger{36}
%X{sourceThread} %X{akkaSource} - %msg%n</pattern>
    </encoder>
  </appender>
  <root level="INFO">
    <appender-ref ref="FILE" />
  </root>
</configuration>
```

Non Docker Setup

Details for how to run the the sanctions simulator can be found here: Using the sanctions simulator

If required, also please ensure the config described above has the correct kafka setup for your environment.

Testing it all works

Now's the time to check everything works, so let's rebuild our application:

```
mvn clean install -rf :ipf-tutorial-app
```

And then we could send in a payment:

```
curl -X POST localhost:8080/submit | jq
```

And if we bring up the payment in the Developer GUI ('IPF Transaction Explorer') we can have a look at a couple of interesting things.

Firstly, if we go to the messages tab (search from the main page by unit of work id (uowId), click view on the transaction, click messages tab) we'll see:

Here we can see that we now are recording the messages going both to (SENT) and from (RECEIVED) the sanctions system. If you "Click to view body" you can view the details of the transformed sanctions messages.

Secondly, just to confirm nothing else has changed we can look at the graph of our tutorial flow (click flows, click IptutorialFlowV2, click view graph) then we see:

And here we can see that our flow is working correctly!

Conclusions

In this tutorial we've achieved quite a lot! We've built our own kafka based connectors from scratch and used them to send messages to and from our remote service.

CON1 - Adding payment initiation

CON3 - Writing your own connector (Http)

CON3 - Writing your own Connector (HTTP)

Getting Started

The tutorial step uses the "add_kafka" solution of the project as it's starting point.

If at anytime you want to see the solution to this step, this can be found on the "add_http" solution!

In CON2 - Writing your own connector (Kafka), we connected our application with an external sanctions system to make requests. To do this, we created our own Kafka connector and used it for communication to send requests and receive the responses over Kafka . In this tutorial, we're going to change our protocols and look at using HTTP.

One important difference with the previous example, is that whereas when using Kafka we were using the connector in an asynchronous way, this time we need to use a synchronous HTTP call. This means that we need to use a different style of connector - the "RequestReplyConnector"

You'll do this by integrating in with a test fraud system. This system:

Expects to receive a custom OlafRequest object.

Will return a custom OlafResponse object.

Supporting Classes

The first thing we'll do is import the domain definition for the fraud system. To do this we need to add a dependency into our applications pom.xml:

```
<dependency>
   <artifactId>fraud-domain</artifactId>
   <groupId>com.iconsolutions.ipf.sample.samplesystems</groupId>
</dependency>
```

Let’s look at the key classes we receive from this module. The first is the request object we send to the fraud system.

```
@Data
public class OlafRequest extends SampleEvent<OlafRequestPayload> {
}
```

If we dig a little further we’ll find that the key element of the payload is the FraudRequest:

```
@Builder
@Data
@AllArgsConstructor
@NoArgsConstructor
public class FraudRequest {
   ObjectNode fiToFICstmrCdtTrf;
}
```

Here we can see that it is expecting to be provided with a pacs.008 in an object node format. This is just a jackson representation of the pacs008.

Mapping

In our Kafka example we used a pre-packaged mapper to convert from our pacs.008 to the OlafRequest. Here as an example of the different approaches available we’ll write our own mapper.

So let’s start by creating a mapping class that can take our pacs.008 and create our Fraud request for us. You’ll do this by creating a new "mappers" package and then adding a class for the "FraudMapper".

To create the FraudRequest object we’ll need to use the jackson mapper’s valueToTree method. We can then use this to construct our FraudRequest. Once we have this we wrap it in the OlafRequest object.

The olafRequest object itself expects a header which must contain a technical event id and a functional component (you can see this digging into the SampleEvent class and the Header class from samplesystems.shared.model). See if you can create this mapping method now, and when ready the solution is below:

```
@AllArgsConstructor
public class FraudMapper {
   private final ObjectMapper objectMapper;
   public OlafRequest mapToRequest(FIToFICustomerCreditTransfer fiToFICustomerCreditTransfer) {
      FraudRequest fraudRequest = new FraudRequest();
      fraudRequest.setFiToFICstmrCdtTrf(objectMapper.valueToTree(fiToFICustomerCreditTransfer));
```

```
        OlafRequest olafRequest = new OlafRequest();
        olafRequest.setHeader(Header.builder()
                .technical(Technical.builder().eventId("EventId").build())
                .functional(Functional.builder().build())
                .build());
        olafRequest.setPayload(new OlafRequestPayload(fraudRequest));
        return olafRequest;
    }
}
```

Note here that we're not using any special frameworks to do our mapping as in this case it's just a simple java mapping that is easiest.

Now let's think about the response side. Firstly we'll need a method to map from the body of the HTTP response to a FraudResponse object. For this again we'll just use the object mapper:

```
public OlafResponse convertResponse(String messageText) {
    try {
        return objectMapper.readValue(messageText, OlafResponse.class);
    } catch (JsonProcessingException e) {
        throw new IconRuntimeException(e);
    }
}
```

Let's add this method to our FraudMapper.

Finally, we'll also need a method to map the OlafRequest object to a TransportMessage. In this case our transport message needs to include two things:

The olaf request itself as a string payload

A set of headers containing:

An "httpUrl" header containing the endpoint url to target

An "httpMethod" header, in this case just a constant "POST"

A "Content-Type" header, in this case just a constant "application/json"

For the endpoint URL we'll retrieve it's value from an injected HttpConnectorTransportConfiguration object. You'll deal with the details of this when discussing the transport shortly, but for now all we need is to know we'll use it's getEndpointURL method ot retrieve the endpoint URL.

See if you can add the method to our FraudMapper, and then when ready the solution is below:

```
public TransportMessage mapToTransport(OlafRequest olafRequest) {
```

```
    try {
        MessageHeaders messageHeaders = new MessageHeaders(CryptoHelper.messageHeaders())
                .putHeader("httpUrl", fraudHttpConnectorTransportConfiguration.getEndpointUrl())
                .putHeader("httpMethod", "POST")
                .putHeader("Content-Type", "application/json");
        return new TransportMessage(messageHeaders,
objectMapper.writeValueAsString(olafRequest));
    } catch (JsonProcessingException e) {
        throw new IconRuntimeException(e);
    }
}
```

We do need to use spring to provide to us an instance of the FraudMapper that we can use. To do this we'll create a new bean within the IpfTutorialConfig class as:

```
@Bean
public FraudMapper fraudMapper(ObjectMapper objectMapper,
HttpConnectorTransportConfiguration connectorTransportConfiguration) {
    return new FraudMapper(objectMapper, connectorTransportConfiguration);
}
```

That's all of our mappings prepared, let's go ahead and start creating our connector.

The Connector

Now that we have our mapper ready, we're good to start building our connector.

Firstly, let's add the dependency for the connector framework's HTTP implementation.

```
<dependency>
    <groupId>com.iconsolutions.ipf.core.connector</groupId>
    <artifactId>connector-http</artifactId>
</dependency>
```

Now let's write our connector, again you'll need to make a few decisions first:

Types - The definition of a request-reply connector is RequestReplySendConnector<REQ_D, REQ_T, REP_D, REP_T>. In this instance:

the REQ_D represents the source (domain) object type

the REQ_T the target object type (which is sent to the HTTP service).

the REP_D represents the response (domain) object type (received back from the HTTP service)

the REP_T the target object type (returned from the connector's send method).

In our connector will take in a FIToFICustomerCreditTransfer, transform it to an OlafRequest to send to the Fraud system and then return a OlafResponse. When we receive that OlafResponse we'll leave it in that format (but we could map it to something else if we wanted!).

Logging - As with our previous example, we'll make it as simple as possible and use the logging implementation that has been provided for us.

Mapping - Here we'll use our mapper we discussed above!

Let's start by thinking at the class level. You'll create a new class within our connectors package for the FraudConnectorConfiguration.

Your class will need to access the actor system, the fraud mapper and the message logger implementation to support the creation of the request reply connector.

```
@Slf4j

@Configuration

@AllArgsConstructor

public class FraudConnectorConfiguration {

   private final ClassicActorSystemProvider actorSystem;

   private final MessageLogger messageLogger;

   private final FraudMapper fraudMapper;

   @Bean

   public RequestReplySendConnector<FIToFICustomerCreditTransfer, OlafRequest, OlafResponse,
OlafResponse> fraudSendConnector(HttpConnectorTransport<OlafRequest>
fraudHttpConnectorTransport) {

      return new RequestReplySendConnector.Builder<FIToFICustomerCreditTransfer, OlafRequest,
OlafResponse, OlafResponse>("Fraud")

            .withActorSystem(actorSystem)

            .withMessageLogger(messageLogger)

            .withDomainToTargetTypeConverter(fraudMapper::mapToRequest)

            .withSendTransportMessageConverter(fraudMapper::mapToTransport)

            .withReceiveTransportMessageConverter(message ->
fraudMapper.convertResponse(message.getPayload().toString()))

            .withConnectorTransport(fraudHttpConnectorTransport)

            .build();

   }

}
```

This is important enough to walk through each part in turn.

Firstly we pass into the method a HttpConnectorTransport<OlafRequest> fraudHttpConnectorTransport. This is an implementation of our HTTP connector transport. You'll discuss creating that in a minute!

The builder construction - this takes a simple string of the name we want to provide of connector.

The actor system

the message logger - as discussed we'll just use standard logging approaches here for now.

the domain to target type converter - here we'll use our mapToRequest method we built on the FraudMapper, to map FIToFICustomerCreditTransfer to an OlafRequest.

the transport message converter - here we'll use our mapToTransport method we built on the FraudMapper, to map the OlafRequest to a TransportMessage.

the receive transport type converter - here we'll just supply a function that converts the message payload to a string and then uses our fraud mapper service to change into our OlafResponse.

the transport - obviously here we just use the connector transport we have passed to the method.

That's it, that's our very first send connector built from scratch.

You may wonder where the correlation controls are. In the request reply world, because everything is synchronous, the correlation features are a complexity we don't have to worry about as everything is handled within the same thread!

That's our connector defined, let's now move onto the transports.

The Connector Transport

Now's the time to set up the connector transport. In this case, it's a HttpConnectorTransport you need.

```
@Configuration
@AllArgsConstructor
public class FraudTransportConfiguration {
   private ClassicActorSystemProvider actorSystem;
   @Bean
   public HttpConnectorTransportConfiguration fraudHttpConnectorTransportConfiguration() {
      return
HttpConnectorTransportConfiguration.create(actorSystem.classicSystem().settings().config(),
"fraud");
   }
   @Bean
   @SneakyThrows
```

```
   public HttpConnectorTransport
fraudHttpConnectorTransport(HttpConnectorTransportConfiguration
fraudHttpConnectorTransportConfiguration) {

      return new HttpConnectorTransport.Builder()

            .withName("OlafRequestReplyHttpConnectorTransport")

            .withActorSystem(actorSystem)

            .withTransportConfiguration(fraudHttpConnectorTransportConfiguration)

            .build();

   }

}
```

Using the connector

Now let's look back at our FraudSystemActionAdapter. To make things simple in our current use case, you'll simply adapt the existing success case (else) to send to the FraudSystem using the newly built HTTP connector, sending your pacs.008 to the fraud service. You'll assume for brevity that any response is successful.

See if you can update the adapter now, and when ready the solution is below (the other else if conditions remain as before).

```
@Slf4j

public class FraudSystemActionAdapter implements FraudSystemActionPort {

   private RequestReplySendConnector<FIToFICustomerCreditTransfer, OlafRequest, OlafResponse,
OlafResponse> fraudConnector;

   @Autowired

   public FraudSystemActionAdapter(RequestReplySendConnector<FIToFICustomerCreditTransfer,
OlafRequest, OlafResponse, OlafResponse> fraudConnector) {

      this.fraudConnector = fraudConnector;

   }

   ...

      } else {

         return
fraudConnector.send(action.getProcessingContext(),action.getCustomerCreditTransfer())

               .thenCompose(response -> IpftutorialmodelDomain.fraudSystem().handle(new
FraudCheckResponseInput.Builder(action.getId(), AcceptOrRejectCodes.Accepted).build())

               .thenAccept((Done done) -> log.debug("Sent input of type {} for id {} with result {}",
done.getCommandName(), action.getId(), done.getResult().name())));

      }
```

```
  }
}
```

You also need to make a modification to use the RequestReplySendConnector fraudConnector and the changed FraudSystemActionAdapter now that we added the FraudConnector to its constructor.

```
@Bean
 public IpftutorialmodelDomain init(ActorSystem actorSystem,
                IsoMappingService mappingService,
                SendConnector<FIToFICustomerCreditTransfer, SanctionsRequest> sanctionsConnector,
                RequestReplySendConnector<FIToFICustomerCreditTransfer, OlafRequest, OlafResponse,
OlafResponse> fraudConnector) {
  ...
          .withFraudSystemActionAdapter(new FraudSystemActionAdapter(fraudConnector))
  ...
```

That's all our code done, next up let's look at configuration.

Configuration

You'll add our configuration into our application configuration file (ipf-tutorial-app/application.conf). Firstly, as per the discussion above we need to tell the connector what to do on failover. For this you will create a common restart configuration:

```
fraud {
 transport = http
 http {
  client {
   host = "localhost"
   port = "8084"
   endpoint-url = "/v1"
  }
 }
}
```

This configuration assumes that the fraud simulator will be running on the localhost and port 8084. This is what it will be when running from the command line (see non-docker setup). If we're running from within docker, we'll need to update only the host and port.

That's everything from our application side complete.

Running the application

To run the application, the first thing you'll need to do is set up the actual fraud service that you will be talking to. For this you need a new entry in the application.yml (docker/application.yml)

Docker Setup

```
 fraud-sim:
   image: registry.ipf.iconsolutions.com/sample-systems-fraud-simulator-http:2.0.40
   container_name: fraud-sim
   environment:
    - FRAUD_SIM_ENCRYPTION_ENABLED=FALSE
   ports:
    - 8089:8080
    - 8090:55555
   volumes:
    - ./config/fraud:/fraud-simulator-http/conf
    - ./logs:/ipf/logs
   user: "1000:1000"
```

Note that the "2.0.21" version provided here is the latest version at the time of writing of this document.

To make things easier you'll also add a logback.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
   <appender name="FILE"
          class="ch.qos.logback.core.rolling.RollingFileAppender">
      <file>/ipf/logs/fraud-sim.log</file>
      <rollingPolicy class="ch.qos.logback.core.rolling.FixedWindowRollingPolicy">
         <fileNamePattern>/ipf/logs/fraud-sim.log.%i</fileNamePattern>
         <minIndex>1</minIndex>
         <maxIndex>20</maxIndex>
      </rollingPolicy>
      <triggeringPolicy
           class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">
         <maxFileSize>50MB</maxFileSize>
      </triggeringPolicy>
```

```
      <encoder>
        <pattern>%date{yyyy-MM-dd} %d{HH:mm:ss.SSS} %-5level %X{traceId} %logger{36} %X{sourceThread} %X{akkaSource} - %msg%n</pattern>
      </encoder>
    </appender>
    <root level="INFO">
      <appender-ref ref="FILE" />
    </root>
</configuration>
```

You'll also need to update the application.conf in the docker setup to tell it the hostname of the new fraud simulator. To do this we simply add:

```
fraud {
  http {
    client {
      host = "fraud-sim"
      port = 8080
    }
  }
}
```

Non Docker Setup

Details for how to run the the fraud simulator can be found here: Using the fraud simulator

Testing it all works

Now's the time to check everything works, so let's rebuild the application:

```
mvn clean install -rf :ipf-tutorial-app
```

And then you can send in a payment:

```
curl -X POST localhost:8080/submit -H 'Content-Type: application/json' -d '{"value": "25"}' | jq
```

Note here you are sending in a value of 25. That's to ensure you reach our happy path and call out to the Fraud Simulator.

And if you bring up the payment in the Developer GUI and look at the messages of the tutorial flow (search by unit of work ID, click view, click ipf tutorial flow, click messages) then you will see:

Here you can see that we are now also sending out OlafRequest fraud messages. If you look at the message data (Click to view body), you should see the Request message contains the full fiToFICstmrCdtTrf (pacs.008) and the response has the fraudFeedback payload.

Conclusions

In this section, we've established a new HTTP connection to an existing fraud server.

CON2 - Writing your own connector (Kafka)

TEST1 - Adding tests

TEST1 - Adding Tests

Getting Started

The tutorial step uses the add_http solution of the project as it's starting point.

If at anytime you want to see the solution to this step, this can be found on the add_tests solution.

Upto now we've been focusing on experimenting with some of the different capabilities of the IPF Framework and how we can use them to quickly build up an application.

During that time, we haven't looked at how we can test our application.

Here we'll introduce Icon's Test Framework and show how we can use it to test the application you've built.

We'll assume in this tutorial a basic awareness of what the Test Framework is, and an understanding of both BDD and the Gherkin syntax.

The Icon Test Framework

Concepts

You'll start our intro into the Test Framework by summarising some key concepts.

Message: An abstraction model for any 'message' that is handled by the framework implementation (request, response, payload etc).

A message is typed against a known Java type that represent the contents de-serialised form, also referred to as Document Type.

MessageType: A representation of the messages type that can be referred to through the BDD, there should be a one to one mapping between MessageType instance and a Messages associated Document type.

MessageDefinition: A contextual structure that provides functionality for handling messages of the configured type, serving as a point of Inversion of Control with the test-framework.

There should be a one-one mapping between the MessageDefinition instance and configured Message Type, and it is common to see both Message and MessageDefinition as arguments to core methods.

MessageBucket: A glorified collection that any messages received by the test-framework (either directly from Consumers, or secondary such as HTTP responses) are appended to.

The internal collection is encapsulated and a predicate-based accessor methods are provided in order to "fish" correlated messages from the bucket.

A successfully "fished" message is typically removed from the bucket and added to the test own Context object.

Transporter: An abstraction of a protocol on which a message may be sent to the target system e.g. HTTP, JMS etc

Context: A scenario context that holds test information and is accessible from any step, the internal data structure is thread local to facilitate parallelisation and the is cleared down between scenarios by JBehave lifecycle hooks.

Extensions

For this tutorial we're going to use an extension on the Test Framework that is explicitly designed to make testing easier using the IPF product suite.

```
<dependency>
  <groupId>com.iconsolutions.ipf.core.test</groupId>
  <artifactId>ipf-test-fw-core</artifactId>
  <scope>test</scope>
</dependency>
```

The ipf-test-fw provides a number of useful things:

A set of pre-built steps that utilise the system events structure of an IPF application to provide rich processing steps that can be used for validation.

A set of pre-built steps that utilise the model operations capability to interrogate the actual aggregate of any given flow.

A set of common steps (scenario start / end)

A set of transporter utilities to allow easy set up of stubbed HTTP, Kafka and JMS services.

You'll use these features throughout this tutorial.

Project Set Up

You'll begin by putting the basics of a new project in place for the tests.

This will be a new Maven module which you'll call ipf-tutorial-application-tests.

If you're using IntelliJ you can do this by right-clicking on the ipf-tutorial project in the project view and selecting New Module.

Then you should be prompted to add a new Maven module:

Then you should be prompted to add a new Maven module:

Press "Finish" to complete the project setup.

Once complete if you expand the module in the navigator, you can delete the ipf-tutorial-application-tests/src/main directory as we will only be working in the test folder here.

You'll also a new directory "resources" under the ipf-tutorial-application-tests/src/test directory.

You'll make this as a test resouces root (right click the folder > Mark Directory As > Test Resources Root).

Under the new resources directory we'll add one more directory called "stories".

When complete our project structure should look like:

A First BDD Test

Now we have a project, let's get on and start writing our first BDD test case.

To do this we need to create a "story" file.

Let's create a new file called HappyPath.story and add it to the new stories directory.

There are some great plugins available within IntelliJ to help support the development of BDD test cases.

We recommend for example this one: IntelliJBehave.

When installed, it will provide ability to see which steps have already been implemented and provide click through capability to see the code.

Let's now populate your story file:

Meta:

Narrative:

This test covers the basic tutorial happy path flow.

Scenario: Basic Happy Path

Given the IPF service is healthy

This is the basis of all the stories we'll write for IPF.

The first line of the scenario "Given the IPF service is healthy" is one of the steps we'll use for the ipf-test-fw capabilities to check that IPF has come up and is ready to process, this will ensure that all the connectors in the application are up and running before we start a test.

When running a test, if this step fails, always check the logs as it will tell you which connectors have failed.

This is normally down to a configuration error in your test!

Having confirmed our application is up and running, we need to start thinking about the different steps of our payment lifecycle.

You'll do this in as minimal way as possible over this tutorial and then build out on that later.

Hence the summary below, is not an exhaustive list of all the options and abilities of the test-fw, nor a full test but simply and introduction to get us started!

1. A payment is sent in (we'll assume via the HTTP controller for now)

When the channel sends a 'initiation request' with values:

| requestId | fraud-happy-request-id |

| value | 25 |

This is an important line to digest and understand.

Firstly, we use the term "channel" here, but we use this term to represent the invoker of the HTTP initiation call.

We then define the type of request we want to send in "initiation request" and we provide two values: the requestId (containing value 'fraud-happy-request-id') and the value (containing 25).

The value is easy to understanding, we are sending in a value < 30 to ensure we hit the right scenario.

The request ID will be used by our test to track the specific scenario we are running here.

The test framework is capable of running many tests in paralell, so we need to be able to uniquely identify our tests and it is this value that we will do that with.

It'll become more obvious how this is used later!

2. An initiation flow is created.

Then a new InitiationFlow flow is started

3. The HTTP response is returned to the initiating call.

And the channel receives a 'initiation response'

4. The initiation flow creates a new ipftutorialv2 flow.

And a new IpftutorialflowV2 flow is started

5. The ipftutorialv2 flow process a couple of no-ops (duplicate check, account validation) and then calls the sanctions system.

And Sanctions receives a 'sanctions request'

So here we're telling our test that the sanctions system must receive a new message type "sanctions request"

6. The sanctions system returns a response.

When Sanctions sends a 'sanctions response'

Again here we can see we're using another definition of a message the "sanctions response".

7. If the payment value < 30, a call is made to the fraud system

Then Fraud receives a 'fraud request'

8. The fraud system returns a response.

When Fraud sends a 'fraud response'

9. The ipftutorialv2 flow uses sample no-op adapters to complete (clear and settle)

Then the 'InitiationFlow' flow is in state 'Complete'

10. The execution flow returns control to the initiation flow which also completes.

And the 'IpftutorialflowV2' flow is in state 'Complete'

Putting this all together we have our first full BDD test!

Meta:

Narrative:

E2E test to demonstrate testing a flow

Scenario: Execute Fraud Happy

Given the IPF service is healthy

When the channel sends a 'initiation request' with values:

| requestId | fraud-happy-request-id |

| value | 25 |

Then a new InitiationFlow flow is started

And the channel receives a 'initiation response'

And a new IpftutorialflowV2 flow is started

And Sanctions receives a 'sanctions request'

When Sanctions sends a 'sanctions response'

Then Fraud receives a 'fraud request'

When Fraud sends a 'fraud response'

Then the 'InitiationFlow' flow is in state 'Complete'

And the 'IpftutorialflowV2' flow is in state 'Complete'

It’s quite a simple test for now, and we’ll look to add some complexity and different elements of testing within it during the course of this tutorial, but for now this will be our starting test that we’re going to run.

Test Implementation

The first thing we need to do is to add our dependency to the ipf-test-fw as discussed.

You’ll also need to tell the code where the application code itself is, so we’ll need to add a dependency to the ipf-tutorial-app itself too.

```
<dependencies>
  <dependency>
    <groupId>com.iconsolutions.ipf.tutorial</groupId>
    <artifactId>ipf-tutorial-app</artifactId>
    <version>${project.version}</version>
    <scope>test</scope>
```

```
    </dependency>
    <dependency>
      <groupId>com.iconsolutions.ipf.core.test</groupId>
      <artifactId>ipf-test-fw-core</artifactId>
       <scope>test</scope>
    </dependency>
</dependencies>
```

There are a number of key things we now need to build in order to get our tests to run:

A "runner" - this is a class that will provide the spring boot test runner that will execute all our story files.

"Config classes" for the different external services - we'll need to tell the test framework how we want to stub out the real services that are used as part of our flow.

So in our case, we're going to have to provide config for Fraud, Sanctions and payment initialisation.

"Config" files - just like we did in our docker environment, we'll need to supply

Let's look at each of these now.

The Runner

The runner actually runs our tests, it's responsibility is to determine all the available story files and execute the scenarios within them.

So let's set up a new class like the below:

```
@SpringBootTest(classes = Application.class, webEnvironment =
SpringBootTest.WebEnvironment.DEFINED_PORT)
@Import({AllTestConfig.class})
public class FeatureTestRunner extends IPFFeatureTestRunner {
}
```

You'll add this into a new package com.iconsolutions.ipf.tutorial.test

The runner we're using here extends the out of the box IPF Feature Test runner.

We're telling it that we're going to use our Application.class (from the ipf-tutorial-app project) as the basis of our spring boot test.

Note that we're providing it the AllTestConfig import, this will enable a number of features we'll use in setting up the running of our test, and we'll discuss these later.

For our test, we're going to use mongo as our database and Kafka as the transport layer for sanctions, so we need to consider how we will use those.

In this tutorial we're going to use test containers to supply us with the docker implementations of both that the test will bootstrap as part of it's execution.

To add these is really simple, firstly we need to change our class definition for the runner:

public class FeatureTestRunner extends IPFFeatureTestRunner implements KafkaIntegrationTestSupport, MongoIntegrationTestSupport {

So here you can see we've added test support for both Kafka and Mongo.

Now we also need to tell Kafka which topics we want to create and then start the containers.

You'll do this within a static block within our runner:

```
static {
    kafkaContainer.withEnv("KAFKA_CREATE_TOPICS",
"SANCTIONS_RESPONSE:1:1,SANCTIONS_REQUEST:1:1,PAYMENT_INITIATION_REQUEST:1:1,PAYME
NT_INITIATION_RESPONSE:1:1");
    SingletonContainerPattern.startAllInParallel(mongoContainer, kafkaContainer);
}
```

So here we're telling our tests which topics to create (for sanctions and the payment initiation over Kafka) and then we're simply starting the two containers.

That's everything on our runner for now, we'll make add in a couple of extra config files later to complete it.

Configuration Classes

Payment Initiation

As discussed above, for now we're going to use the HTTP use case for payment initiation.

In our BDD, we defined to messages the "initiation request" and the "initiation response".

You'll need to create types and definitions for how the test framework should handle these requests.

Let's create a new package called config.

In here we will create a class called InitiationConfig.

The first thing we need to do in our initiation config is to define the message types.

These message types need to implement the MessageType interface.

So let's construct a new enum within our config file for the message types:

```
public enum InitiationTypes implements MessageType {
    INITIATION_REQUEST("initiate request"),
    INITIATION_RESPONSE("initiation response");
    private final String name;
    InitiationTypes(String aName) {
        name = aName;
```

```
    }
    @Override
    public String getName() {
        return name();
    }
    @Override
    public Set<String> getAliases() {
        return Sets.newHashSet(name);
    }
}
```

Here we can see we're defining two message types.

The key bit to note here is that the names provided in the constructor for the types must match the names provided in your BDD story file.

Having set up a message type, we now need to define the messages.

For each message type we have setup (so both our initiationrequest and response in this case) we need to create a new "MessageDefinition" for that type.

You'll start with the initiation response as it's slightly easier:

```
@Bean
MessageDefinition<InitiationResponse> initiationResponseMessageDefinition() {
    return new DefaultMessageDefinition.Builder<InitiationResponse>()
            .withType(InitiationTypes.INITIATION_RESPONSE) (1)
            .withCausedByType(InitiationTypes.INITIATION_REQUEST) (2)
            .withDocumentTypeClass(InitiationResponse.class) (3)
            .withCorrelatingIdGet(doc -> Optional.ofNullable(doc.getDocument().getRequestId())) (4)
            .build();
}
```

Let's walk through the key points here:

1

Here we are defining the type that this definition applies too, so that's simple it's the initiation response one.

2

As this is a response, we define the type that this message has been caused by - i.e. the request message.

This allows correlation to take place automatically over http.

3

Here we are defininig the actual java type of the class, so again a simple one that the response that comes back from the initiation controller is a InitiationResponse.

4

Here we need to define a function that tells us how to correlate this response with the initial request.

So how will the test framework be able to know that a response and request are linked.

We'll simply do this by using the request ID field - so here we are saying that the "request id" on the "InitiationResponse" object will contain the same value that we will define on the initiation request definition next.

Let's now look at the request definition:

```
@Bean
MessageDefinition<InitiationRequest>
initiationRequestMessageDefinition(@Value("${application.base.url}") String baseUrl) {
   return new DefaultMessageDefinition.Builder<InitiationRequest>()
         .withType(InitiationTypes.INITIATION_REQUEST) (1)
         .withDocumentTypeClass(InitiationRequest.class) (2)
         .withGenerator(props -> new InitiationRequest()) (3)
         .withCorrelatingIdGet(doc -> Optional.ofNullable(doc.getDocument().getRequestId())) (4)
         .withDestination(baseUrl + "/submit") (5)
         .withPreSend(message -> {
            ContextUtils.setClientRequestId(message.getDocument().getRequestId()); (6)
         })
         .build();
}
```

Again lets walk through each part of this definition:

1

This time our definition is for the request message type.

2

This time our definition is for actually sending the request so we're using the InitiationRequest object.

3

We need to supply a generator.

This generator tells the test framework that when it needs to send an initiation request it will construct it using this generator method.

In our case, we just want a simple brand new InitiationRequest object.

You'll use a more complicated generator definition later when looking at fraud.

4

Similarly to the response definition, we need to supply the correlation approach.

So here again we're just going to use the request id.

So the two correlation functions (on the request and response) together mean that the request ID on the request object must match the request ID on the response object for the test framework to know they are the related pair.

5

For the destination, this is the HTTP address that the initiation request will be sent to.

Note here we are injecting the url path by properties so we'll need to add a property for this into our application.conf file.

You'll do this in the configuration file section below.

6

In the pre-send we can set any extra things that need to be done to help the test framework before the message is sent out.

In our case, we're going to set the local request ID definition on the test to the one on our message.

This request id will then be available for use throughout our scenario - but will be unique to our scenario.

This is key in giving us the ability to process multiple scenario's concurrently.

That's our definitions done, so the final part of our initiation setup is to set the transport up, i.e. we need to provide the test framework with a mechanism to make a call to the initiation controller.

To do this we're going to use another test-fw utility, the HttpSenderTestTransporter.

Let's create another bean for it as follows:

```
@Bean

public MessageTransport initiationTransport(MessageDefinition<InitiationRequest>
initiationRequestMessageDefinition, MessageDefinition<InitiationResponse>
initiationResponseMessageDefinition) {
```

```
    return new HttpSenderTestTransporter.Builder<InitiationRequest, InitiationResponse>()
            .withIdentifier("initiation")
            .withRequestMessageDefinition(initiationRequestMessageDefinition)
            .withResponseMessageDefinition(initiationResponseMessageDefinition)
            .build();
}
```

So this is really simple, we construct a new instance and provide a unique identifier (if we have multiple sender transports each one will need a unique id, you could leave this blank but for tracing any issues a known name is better!).

We also provide access to the request and response definition we set up.

The transport will extract all the other information it needs from those message definitions!

That's everything done, from an initiation viewpoint we're all complete and ready to start testing.

Let's now look at sanctions.

The Sanctions Definition

Now we move onto sanctions, the steps are essentially the same:

Create the message types

Create the definitions for the request and response

Define the transport.

The only difference here is this time rather than using the HttpSenderTransport, we'll use a KafkaMessageTransport.

So let's create a new class for our "SanctionsConfig".

Firstly let's create our message type enum.

Remember from our BDD we have defined the "sanctions request" and "sanctions response" messages.

See if you can create our enum now and when ready the solution is below:

```
public enum SanctionsTypes implements MessageType {
    SANCTIONS_REQUEST("sanctions request"),
    SANCTIONS_RESPONSE("sanctions response");
    private final String name;
    SanctionsTypes(String aName) {
        name = aName;
    }
```

```
    @Override
    public String getName() {
        return name();
    }
    @Override
    public Set<String> getAliases() {
        return Sets.newHashSet(name);
    }
}
```

Now let's think about our request definition.

The first thing to realise is that from the test framework perspective it's going to "receive" the sanctions request.

So things are effectively reversed when considering the test framework - i.e. when we use a send connector to send out the message the test framework will receive it.

It's also going to receive it from a Kafka topic, so we'll need to provide a mechanism to convert from the serialised string on the topic to the actual Sanctions Request object.

Let's see how that all works:

```
@Bean
MessageDefinition<SanctionsRequest> receiveSanctionsRequest() {
    return new DefaultMessageDefinition.Builder<SanctionsRequest>()
            .withType(SanctionsTypes.SANCTIONS_REQUEST) (1)
            .withDocumentTypeClass(SanctionsRequest.class) (2)
            .withSource("SANCTIONS_REQUEST") (3)
            .withFromStringMapper(s -> SerializationHelper.stringToObject(s, SanctionsRequest.class)) (4)
            .withCorrelatingIdGet(doc -> Optional.ofNullable(ContextUtils.getCorrelatingId())) (5)
            .build();
}
```

1

Here again we define our type.

2

And the java object.

3

The source field represents the Kafka topic we're going to read from, note that it's called source here as message definitions are protocol independent, so this would be the same definition in a jms world (replacing the topic idea for a queue name)

4

The fromStringMapper defines how we're going to convert from the serialized string version of our message to our java class.

In our case, we're just going to use a pre-defined stringToObject function (this is available from Icon's SerializationHelper) which will do a simple jackson mapping.

5

This function tells the test framework where to get the correlating ID from which is used to match the response to the request message

Next is our response definition, like with payment initiation the key here is that we will also need to provide a generator function to create the new sanctions response.

```
@Bean
MessageDefinition<SanctionsResponse> sendSanctionsResponse() {
   return new DefaultMessageDefinition.Builder<SanctionsResponse>()
         .withType(SanctionsTypes.SANCTIONS_RESPONSE)
         .withDocumentTypeClass(SanctionsResponse.class)
         .withDestination("SANCTIONS_RESPONSE")
         .withGenerator(props -> {
            SanctionsResponse sanctionsResponse = new SanctionsResponse();
            sanctionsResponse.setHeader(HeaderUtils.makeHeader("Sanctions",
ContextUtils.getClientRequestId()));
            sanctionsResponse.setPayload(new SanctionsResponsePayload());
            sanctionsResponse.getHeader().getTechnical().setOriginalEventId(((SanctionsRequest)
PreviousMessages.getLastMessage(SanctionsTypes.SANCTIONS_REQUEST,
false).getDocument()).getHeader().getTechnical().getEventId());
            return sanctionsResponse;
         })
         .withCorrelatingIdGet(doc -> Optional.ofNullable(ContextUtils.getClientRequestId()))
         .build();
}
```

Here we can see we create a new response object and add a default header to it.

The most interesting part is that we set the original event Id… WHY?

Finally we need to consider the transport again.

This time we'll just use a KafkaMessageTransport implementation.

Let's look at this:

```
@Bean
public MessageTransport sanctionsKafkaTransport(MessageDefinition<SanctionsRequest>
sanctionsRequestMessageDefinition,
MessageDefinition<SanctionsResponse> sanctionsResponseMessageDefinition,
ClassicActorSystemProvider actorSystem) {
   return new KafkaTestTransporter.Builder<SanctionsRequest,SanctionsResponse>()
         .withIdentifier("sanctions") (1)
         .withPropertiesPath("sanctions") (2)
         .withRequestMessageDefinition(sanctionsRequestMessageDefinition) (3)
         .withResponseMessageDefinition(sanctionsResponseMessageDefinition) (4)
         .withActorSystem(actorSystem) (5)
         .build();
}
```

Let's take a look at the key parts of this:

1

Again we provide a unique identifier for the transport.

2

For Kafka we're going to retrieve our configuration from a properties file, we'll look at the details of the configuration in the next section, here all we need to do is specify the path within the config file that our sanctions properties will be.

3

Here is simply our request message definition

4

Here is simply our response message definition

5

And finally we pass the actor system

That's all our sanctions setup done, so let's move onto Fraud.

The Fraud Definition

As normal, we'll create a FraudConfig class and then start by adding the fraud message types and definitions.

T

Just like with Sanctions, the main complexity is the generation of the response object.

The most important point on the generation of the olaf response is that we need to ensure that the originalEventId field contains the original id from incoming request message.

This is required in order to be able to correlate.

```
private OlafResponse createAndEnrichResponseFrom(OlafRequest olafRequest) {
   String originalId = olafRequest.getHeader().getTechnical().getEventId();
   com.iconsolutions.samplesystems.shared.model.header.Header header =
Header.copy(olafRequest.getHeader());
   header.getTechnical().setEventId(UUID.randomUUID().toString());
   header.getTechnical().setOriginalEventId(originalId);
   header.getTechnical().setEventType("FraudFeedback");
   OlafResponse response = new OlafResponse();
   response.setHeader(header);
   response.setPayload(new FraudFeedbackPayload(new FraudFeedback(new
FraudFeedback.PaymentStatusChanged("FraudFeedbackOK", "0"), null)));
   return response;
}
```

The next key point to consider is how we find the olaf request to send to our message.

For this we're going to use the "PreviousMessages" capability of the test framework which allows us to retrieve the last fraud request message.

That's all the information we need to create our definitions.

See if you can do this and when ready the solution is below:

```
@Bean
MessageDefinition<OlafResponse> fraudResponseMessageDefinition() {
   return new DefaultMessageDefinition.Builder<OlafResponse>()
         .withType(FraudTypes.FRAUD_RESPONSE)
         .withCausedByType(FraudTypes.FRAUD_REQUEST)
         .withDocumentTypeClass(OlafResponse.class)
         .withCorrelatingIdGet(response -> Optional.of(ContextUtils.getClientRequestId()))
```

```
        .withGenerator((param) -> createAndEnrichResponseFrom((OlafRequest)
PreviousMessages.getLastDocumentReceived(FraudTypes.FRAUD_REQUEST).getDocument()))
        .build();
}
@Bean
MessageDefinition<OlafRequest> fraudRequestMessageDefinition() {
   return new DefaultMessageDefinition.Builder<OlafRequest>()
        .withType(FraudTypes.FRAUD_REQUEST)
        .withDocumentTypeClass(OlafRequest.class)
        .withFromStringMapper(s -> SerializationHelper.stringToObject(s, OlafRequest.class))
        .withCorrelatingIdGet(fraudRequest -> Optional.of(ContextUtils.getClientRequestId()))
        .build();
}
public enum FraudTypes implements MessageType {
   FRAUD_REQUEST("fraud request"),
   FRAUD_RESPONSE("fraud response");
   private final String name;
   FraudTypes(String aName) {
      name = aName;
   }
   @Override
   public String getName() {
      return name();
   }
   @Override
   public Set<String> getAliases() {
      return Sets.newHashSet(name);
   }
}
```

Now for transports, we'll use the FraudConsumerTransport.

Note that this is effectively the opposite way round to the way we were working in the payment initiation space.

There we were sending a message to IPF and reading the response.

Here IPF is sending us a message and reading the response.

Hence it is a consumer transport and not a sender one as in the payment initiation case.

```
@Bean
   public MessageTransport fraudTransport(FlowIdentifiersFinder finder,
                                          @Value("${fraud.http.client.port}") String port,
                                          MessageDefinition<OlafRequest> fraudRequestMessageDefinition,
                                          MessageDefinition<OlafResponse> fraudResponseMessageDefinition,
                                          ClassicActorSystemProvider actorSystem) {
      return new HttpConsumerTestTransporter.Builder()
            .withIdentifier("fraud") (1)
            .withPort(Integer.parseInt(port)) (2)
            .withOperation(new HttpOperation.Builder<>("v1", fraudRequestMessageDefinition,
fraudResponseMessageDefinition).withHttpMethod(HttpMethod.POST).build()) (3)
            .withActorSystem(actorSystem)
            .build();
   }
```

The transport here is very similar to the other test transports we've done to date.

The key differences are:

1

We need to define the identifier

2

We need to define the port which it will be on.

3

We need to define the function that needs to be performed, i.e. the 'HttpOperation'.

To construct this we use the builder and specify these four parameters:

The path of the function

The method type it will accept (POST, GET or PUT)

The request message type

The response message type

That's all our fraud setup done.

Processing Data

The last bit of configuration we have to do is to provide something that will listen to the processing data.

You'll do this by adding a simple wiremock consumer.

```
<dependency>
   <groupId>com.github.tomakehurst</groupId>
   <artifactId>wiremock</artifactId>
   <version>2.27.1</version>
</dependency>
<dependency>
   <groupId>com.github.tomakehurst</groupId>
   <artifactId>wiremock-jre8-standalone</artifactId>
   <version>2.27.1</version>
</dependency>
@Configuration
public class DummyODSConsumer {
   @Bean
   public WireMockServer odsMock(ObjectMapper objectMapper) throws JsonProcessingException {
      WireMockServer wireMockServer = new WireMockServer(
            new WireMockConfiguration()
                  .port(8093)
                  .needClientAuth(true)
      );
      wireMockServer.start();
      wireMockServer.stubFor(WireMock.post(WireMock.urlEqualTo("/ipf-processing-data"))
            .willReturn(WireMock.aResponse()
                  .withStatus(200)));
      return wireMockServer;
   }
}
```

Here we are simply ignoring and returning a success response for the ODS call.

Updating the Runner

Now we have all our configuration files, we need to add them to our runner:

@SpringBootTest(classes = Application.class, webEnvironment = SpringBootTest.WebEnvironment.DEFINED_PORT)

@Import({AllTestConfig.class, FraudConfig.class, SanctionsConfig.class, InitiationConfig.class, DummyODSConsumer.class})

public class FeatureTestRunner extends IPFFeatureTestRunner implements KafkaIntegrationTestSupport, MongoIntegrationTestSupport {

Executing our Test

Now we should just be able to run our test by right clicking on the FeatureTestRunner class and clicking run.

Once run we should see something like:

Running in Maven

Finally, if we want to run in Maven we need to add an extra build plugin to ensure they are executed.

For this, we simply add the Maven failsafe plugin to our pom.xml as follows:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-failsafe-plugin</artifactId>
      <version>2.22.2</version>
      <executions>
        <execution>
          <goals>
            <goal>integration-test</goal>
            <goal>verify</goal>
          </goals>
        </execution>
      </executions>
      <configuration>
        <includes>
          <include>**/*Runner.java</include>
```

```
            </includes>
            <excludes>
              <exclude>**/*Test.java</exclude>
              <exclude>**/*Tests.java</exclude>
              <exclude>**/*InProgressRunner.java</exclude>
              <exclude>**/*RepeatRunner.java</exclude>
            </excludes>
          </configuration>
        </plugin>
      </plugins>
    </build>
```

Conclusions

In this section, we've set up and run a basic test fw test.

CON3 - Writing your own connector (Http)

RES1 - Resiliency and retry settings (HTTP)

RES1 - Resiliency and Retry Settings (HTTP)

Getting Started

The tutorial step uses the add_http solution of the project as it's starting point.

If at anytime you want to see the solution to this step, this can be found on the resiliency solution.

In CON3 - Writing your own connector (HTTP), we connected our application with an external test fraud systems. This gave us an synchronous connection to an external system which is inherently less stable than using Kafka or JMS. And our landscape at this point in the tutorials looks like;

In this tutorial we are going to look at how we can control the resiliency and retry settings in a best effort to allow the HTTP call to be successful. We will do this by simulating failures of the fraud-sim such that HTTP calls to that service will fail.

Starting the Application

If the environment is not running, we need to start up our docker environment. To do this from the docker directory of the project, run the following command:

docker-compose -f application.yml up -d

This should start all applications and simulators. We can check whether the containers are started and healthy using the command:

docker-compose -f application.yml ps

Validate BAU Processing

Lets check everything is working BAU first will all simulator end points up and functioning, send in a payment:

curl -X POST localhost:8080/submit -H 'Content-Type: application/json' -d '{"value": "25"}' | jq

Checking the payment in the Developer GUI we can see the messages being sent and spot the OlafRequest & OlafResponse messages to the fraud-sim (search by unit of work id, click view, click ipf tutorial flow, click messages) then we see:

Failure Scenario Test

Assuming all is well with the BAU processing, lets test the scenario where the fraud-sim is down and OlafResponses are not coming back. The easiest way to do this is to stop the fraud-sim container:

docker stop fraud-sim

Once the container is down we can send in another payment request:

curl -X POST localhost:8080/submit -H 'Content-Type: application/json' -d '{"value": "24"}' | jq

Checking the payment in GUI again you should see the OlafRequest being sent but not OlafResponse coming back and the status of the transaction itself shows as PENDING:

Finally from the GUI we can see the system event which has been generated for this failure:

Its also worth check the container logs to see the exception and the specific errors (this will become important as we configure the service to retry the HTTP call). You will note there are no more errors, processing is effectively stalled with our current configuration:

2022-10-27 09:43:17.245 ERROR c.i.i.c.c.RequestReplySendConnector - Fraud Send connector returned a response without associated response, response: DeliveryOutcome(deliveryReport=DeliveryReport(outcome=FAILURE, deliveryException=java.util.concurrent.CompletionException: java.lang.IllegalStateException: No closed routees for connector: Fraud. Calls are failing fast), response=null)

2022-10-27 09:43:17.246 ERROR c.i.i.c.c.RequestReplySendConnector - Fraud Received a failure for UnitOfWorkId(value=5f97e12d-8bdd-4fbe-92c3-cbe9e2be58a6) with reason java.lang.IllegalStateException: java.util.concurrent.CompletionException: java.lang.IllegalStateException: No closed routees for connector: Fraud. Calls are failing fast

java.util.concurrent.CompletionException: java.lang.IllegalStateException: java.util.concurrent.CompletionException: java.lang.IllegalStateException: No closed routees for connector: Fraud. Calls are failing fast

Configure Timeout and Resiliency Settings

As things stand with the tutorial application it is not proactively configured for retry and has not set the resiliency settings to protect against intermittent errors on the HTTP synchronous connection. It is possible however to define resiliency settings to retry the HTTP call within a defined period and at configurable intervals. The default configuration is shown below, including both the connector settings and the resiliency settings.

Now we'll update the max-attempts to be 6 which is intended to give sufficient retries of the HTTP call to allow the fraud-sim service to recover (attempts of 6, together with the backoff-multiplier of 2 seconds should give 5 attempts before the call-timeout of 30 seconds)

You'll add our configuration into our application configuration file (ipf-tutorial-app/application.conf):

```
fraud {
  transport = http
  http {
    client {
      host = "fraud-sim"
      port = "8080"
      endpoint-url = "/v1"
    }
  }
  resiliency-settings {
    max-attempts = 6
  }
}
```

And we have to update the FraudConnectorConfiguration to provide the resiliency settings from the application config when building the RequestReplySendConnector.

```
...
      return new RequestReplySendConnector.Builder<FIToFICustomerCreditTransfer, OlafRequest, OlafResponse, OlafResponse>(
            "Fraud",
<1>             "fraud",
<2>             actorSystem
          ).withMessageLogger(messageLogger)
    }
```

There are two changes we have made here, both to add extra arguments to the constructor:

1

- We have defined the root of the properties we want to apply to our connector. So here we use "fraud".

2

- We supply the actor system

Note that this also contains retryOnFailureWhen to effectively retry all failures as configured. We can do other things here to retry based on certain Exceptions, for example;

...

.retryOnFailureWhen(this::isRetryableFailure)

Where isRetryableFailure is the result of a test against a defined list of exceptions you want to retry.

Failure Scenario Test 2

Now we can apply this configuration by rebuilding the ipf-tutorial-app container (mvn clean install -rf :ipf-tutorial-app) and starting it, then running through the following test steps:

GIVEN the fraud-sim is stopped && ipf-tutorial-app has resiliency settings to retry HTTP calls

WHEN a payment is initiated && the fraud-sim recovered within the 30 second connector timeout

THEN we the payment will complete processing with delay and retries evident in the logs

docker stop fraud-sim

curl -X POST localhost:8080/submit -H 'Content-Type: application/json' -d '{"value": "23"}' | jq

Wait 5 seconds (this will allow the Connector to retry).

docker start fraud-sim

If you are observing the ipf-tutorial-app logs (change the logback.xml for ipf-tutorial-app to have <logger name="com.iconsolutions.ipf" level="DEBUG"/>

) and you should see retry entries like (note - this is the decision to retry the actual retry happens once the backoff period has expired):

2022-10-31 14:30:50.941 WARN c.i.i.c.c.t.HttpConnectorTransport - Failure reply for association ID [UnitOfWorkId(value=5931c9c4-66f0-4a87-964b-b00ff11954ee)] with exception [UnknownHostException: fraud-sim] and message [TransportMessage(,)]

2022-10-31 14:30:50.949 DEBUG c.i.i.c.c.r.*ResiliencySettings* - *retryOnResult decided to retry this attempt since it was a failure*: DeliveryReport(outcome=FAILURE, deliveryException=akka.stream.StreamTcpException: Tcp command [Connect(fraud-sim:8080,None,List(),Some(10 seconds),true)] failed because of java.net.UnknownHostException: fraud-sim)

(strictly speaking an UnknownHostException shouldn't be retried but its a quick way to demonstrate retry processing and the resiliency settings).

Once the backoff period has passed the actual retry will take place:

2022-10-31 14:30:52.950 DEBUG c.i.i.c.c.r.ResiliencyPassthrough - Calling UNKNOWN : using OlafRequestReplyHttpConnectorTransport

Checking the payment in GUI again you should see the OlafRequest being sent, but the success response in the Messages tab appears after the delay (approximately 15 seconds).

A few things to note:

You can flexibly configure the retries by thinking about the backoff-multiplier & the initial-retry-wait-duration. For example

initialRetryWaitDuration

backoffMultiplier

First 5 attempt intervals

1

2

1, 2, 4, 8, 16

5

2

5, 10, 20, 40, 80

1

5

1, 5, 25, 125, 625

This retry happened within the 30 seconds connector timeout. Thus you should also be considering the call-timeout in conjunction with the resiliency settings.

As the tutorial is currently written, if the retry is not a success within that 30 seconds this will return to the flow and the fraud check won't have been completed.

This is a good example of something which is short term transient and resolves itself quickly. Where that is not the case we have a number of options to configure additional transport end points, to "retry" from the flow by defining appropriate business logic in the IPF DSL.

We also have the options to react differently to actual business responses (using retryOnResultWhen), to retry on certain business error codes returned from the called application. But this should be balanced with how much logic you want at the connector level versus within the flow logic.

The resiliency component is implemented with resilience4j. See docs on the Resilience4j framework for more information on these settings and behaviours.

Conclusions

In this section, we've established potential options for configuring retries on the HTTP Connector. Next steps might be to explore Kafka connectors and other options to configure retry when the call-timeout is exceeded.

TEST1 - Adding tests

RUN1 - Running your application on Kubernetes

RUN 1 - Running on Kubernetes

Prerequisites

In order to run this tutorial you will need to run a kubernetes cluster locally:

Below are some possible options you could use:

Docker Desktop

minikube

Rancher desktop

Install and startup your desired solution for running a Kubernetes cluster locally

Kubernetes (K8s) is an open-source system for automating deployment, scaling and management of containerised applications.

Deploy IPF Tutorial on Kubernetes

For running the IPF tutorial as a clustered application on Kubernetes, we will do the following:

Step 1 - Create a Namespace

Step 2 - Create Service Account

Step 3 - Create Configmap

Step 4 - Create imagePullSecret

Step 5 - Create MongoDB

Step 6 - Create Developer App

Step 7 - Create a Deployment

Requirements

Step 1 - Create a Namespace

kubectl create namespace ipf-tutorial

Step 2 - Create Service Account

Create a 'serviceAccount.yaml' file and copy the following admin service account manifest.

```
---
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: ipf-tutorial
  namespace: ipf-tutorial
rules:
  - apiGroups: [""]
```

```
    resources: ["pods"]
    verbs: ["get", "watch", "list"]
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: ipf-tutorial
  namespace: ipf-tutorial
---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: ipf-tutorial
  namespace: ipf-tutorial
subjects:
  - kind: ServiceAccount
    name: ipf-tutorial
    namespace: ipf-tutorial
roleRef:
  kind: Role
  name: ipf-tutorial
  apiGroup: rbac.authorization.k8s.io
```

The 'serviceAccount.yaml' creates a ipf-tutorial role, 'ipf-tutorial' ServiceAccount and binds the 'Role' to the service account.

The 'ipf-tutorial' role has all the permissions to query the API for IPF pods running within the namespace.

Now create the service account using kubectl.

```
kubectl apply -f serviceAccount.yaml
```

Step 3 - Create Configmap

Create a Configmap file named 'configmap.yaml' and copy the following config manifest.

```
apiVersion: v1
kind: ConfigMap
```

```
metadata:
  name: ipf-tutorial-service-cm
  namespace: ipf-tutorial
data:
  application.conf: |
    akka {
      loglevel = "INFO"
      cluster {
        # undefine seed nodes to allow for Kubernetes to describe cluster topography
        seed-nodes = []
        sharding {
          distributed-data.majority-min-cap = 2
        }
      }
      # Use Kubernetes API to discover the cluster
      discovery {
        kubernetes-api {
          pod-label-selector = "app=%s"
        }
      }
      actor.provider = cluster
    }
    akka.remote.artery.canonical.hostname = ${POD_IP}
    akka.management {
      # available from Akka management >= 1.0.0
      health-checks {
        readiness-path  = "health/ready"
        liveness-path   = "health/alive"
      }
      # use the Kubernetes API to create the cluster
      cluster.bootstrap {
```

```
      contact-point-discovery {
        service-name           = "ipf-tutorial-service"
        discovery-method       = kubernetes-api
        required-contact-point-nr = 2
      }
    }
  }
  management.endpoints.web.exposure.include = "*"
  flow-restart-settings {
    min-backoff = 1s
    max-backoff = 5s
    random-factor = 0.25
    max-restarts = 5000
    max-restarts-within = 3h
  }
  ipf.behaviour.retries.initial-timeout = 3s
  ipf.behaviour.config.action-recovery-delay = 3s
  # CHANGEME
  ipf.mongodb.url = "mongodb://location_of_mongodb_cluster:27017/ipf_tutorial"
 logback.xml: |
  <?xml version="1.0" encoding="UTF-8"?>
  <configuration>
    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
      <target>System.out</target>
      <encoder>
        <pattern>[%date{ISO8601}] [%level] [%logger] [%marker] [%thread] - %msg MDC:
{%mdc}%n</pattern>
      </encoder>
    </appender>
    <appender name="ASYNC" class="ch.qos.logback.classic.AsyncAppender">
      <queueSize>8192</queueSize>
```

```
      <neverBlock>true</neverBlock>
      <appender-ref ref="CONSOLE" />
    </appender>
    <logger name="akka" level="WARN" />
    <root level="INFO">
      <appender-ref ref="ASYNC"/>
    </root>
  </configuration>
```

Now create the config map using kubectl.

```
kubectl apply -f configmap.yaml
```

Step 4 - Create imagePullSecret

Substitute docker-server, docker-username and docker-password for appropriate values

```
kubectl create secret docker-registry registrysecret --docker-server=**** --docker-username=********* --docker-password=******* --namespace ipf-tutorial
```

Step 5 - Create MongoDB

Create a statefulset file named 'infrastructure.yaml' and copy the following manifest. This creates MongoDB which is required by the tutorial application

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mongo
  namespace: ipf-tutorial
spec:
  selector:
    matchLabels:
      role: mongo
  serviceName: "mongo"
  replicas: 1
  template:
    metadata:
      labels:
        role: mongo
```

```
  spec:
    imagePullSecrets:
      - name: "registrysecret"
    terminationGracePeriodSeconds: 10
    containers:
      - name: mongo
        image: https://hub.docker.com/_/mongo/tags/mongo:latest
        imagePullPolicy: Always
        ports:
          - containerPort: 27017
      - name: mongo-exporter
        image: bitnami/mongodb-exporter:0.11.2
        imagePullPolicy: IfNotPresent
        ports:
          - name: mongo-exporter
            containerPort: 9216
            protocol: TCP
        env:
          - name: MONGODB_URI
            value: "mongodb://localhost:27017"
```

```
kubectl apply -f infrastructure.yaml
```

Step 6 - Create Developer App

Create a deployment file named developerApp.yaml and copy the following manifest. This creates the developer application which is required by the tutorial application to view flow events

${registry_service} - substitute for the location of the docker registry

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ipf-developer-service
  namespace: ipf-tutorial
  annotations:
```

```
    prometheus.io/scrape: "true"
    prometheus.io/path: "/"
    prometheus.io/port: "9001"
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ipf-developer-service
      product: ipfv2
  template:
    metadata:
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/path: "/"
        prometheus.io/port: "9001"
      labels:
        app: ipf-developer-service
        product: ipfv2
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: app
                    operator: In
                    values:
                      - ipf-developer-service
              topologyKey: kubernetes.io/hostname
      securityContext:
        fsGroup: 1000
```

```
      runAsUser: 1000
    serviceAccountName: ipf-tutorial
    imagePullSecrets:
      - name: "registrysecret"
    containers:
      - name: ipf-developer-service
        image: $\{registry_service}/ipf-developer-app:latest
        imagePullPolicy: Always
        ports:
          - name: actuator
            containerPort: 8081
        env:
          - name: "POD_NAME"
            valueFrom:
              fieldRef:
                fieldPath: metadata.name
          - name: "POD_IP"
            valueFrom:
              fieldRef:
                fieldPath: status.podIP
          - name: "KUBERNETES_NAMESPACE"
            valueFrom:
              fieldRef:
                fieldPath: metadata.namespace
          - name: "IPF_JAVA_ARGS"
            value: "-Dma.glasnost.orika.writeClassFiles=false -
Dma.glasnost.orika.writeSourceFiles=false"
        resources:
          limits:
            memory: "2Gi"
          requests:
```

```
            memory: "2Gi"
            cpu: "1000m"
          volumeMounts:
           - mountPath: /ipf-developer-app/conf/logback.xml
             name: config-volume
             subPath: logback.xml
           - mountPath: /ipf-developer-app/conf/application.conf
             name: config-volume
             subPath: application.conf
      volumes:
       - name: config-volume
         configMap:
           name: ipf-developer-service-cm
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: ipf-developer-service-cm
  namespace: ipf-tutorial
data:
  application.conf: |
    flow-restart-settings {
     min-backoff = 1s
     max-backoff = 5s
     random-factor = 0.25
     max-restarts = 5
     max-restarts-within = 10m
    }
    spring.data.mongodb.uri = ${?ipf.mongodb.url}
    actor-system-name = ipf-developer
    ipf.mongodb.url = "mongodb://mongo:27017/ipf"
```

```
    ods.security.oauth.enabled = false
    application.write.url="http://ipf-tutorial-app:8080"
    ipf.processing-data.ingress.transport=http
  logback.xml: |
    <?xml version="1.0" encoding="UTF-8"?>
    <configuration>
      <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <target>System.out</target>
        <encoder>
          <pattern>[%date{ISO8601}] [%level] [%logger] [%marker] [%thread] - %msg MDC:
{%mdc}%n</pattern>
        </encoder>
      </appender>
      <appender name="ASYNC" class="ch.qos.logback.classic.AsyncAppender">
        <queueSize>8192</queueSize>
        <neverBlock>true</neverBlock>
        <appender-ref ref="CONSOLE" />
      </appender>
      <logger name="akka" level="WARN" />
      <root level="INFO">
        <appender-ref ref="ASYNC"/>
      </root>
    </configuration>
---
apiVersion: v1
kind: Service
metadata:
  name: ipf-developer-service
  namespace: ipf-tutorial
  labels:
    name: ipf-developer-service
```

```
spec:
  ports:
    - protocol: TCP
      port: 8081
      targetPort: 8081
      name: ipf-developer-service
  selector:
    app: ipf-developer-service
```

Step 7 - Create a Deployment

Create a Deployment file named 'deployment.yaml' and copy the following deployment manifest.

${registry_service} - substitute for the location of the docker registry

${tutorial-service-version} - version of tutorial app

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ipf-tutorial-service
  namespace: ipf-tutorial
  annotations:
    prometheus.io/scrape: "true"
    prometheus.io/path: "/"
    prometheus.io/port: "9001"
spec:
  replicas: 3
  selector:
    matchLabels:
      app: ipf-tutorial-service
      product: ipfv2
  template:
    metadata:
      annotations:
        prometheus.io/scrape: "true"
```

```
    prometheus.io/path: "/"
    prometheus.io/port: "9001"
  labels:
    app: ipf-tutorial-service
    product: ipfv2
spec:
  #     affinity:
  #       podAntiAffinity:
  #         requiredDuringSchedulingIgnoredDuringExecution:
  #           - labelSelector:
  #               matchExpressions:
  #                 - key: app
  #                   operator: In
  #                   values:
  #                     - ipf-tutorial-service
  #             topologyKey: kubernetes.io/hostname
  securityContext:
    fsGroup: 1000
    runAsUser: 1000
  serviceAccountName: ipf-tutorial
  imagePullSecrets:
    - name: "registrysecret"
  containers:
    - name: ipf-tutorial-service
      image: ${registry_service}/ipf-tutorial-app:${tutorial-service-version}
      imagePullPolicy: Always
      ports:
        - name: actuator
          containerPort: 8080
        - name: akka-artery
          containerPort: 55001
```

```
  - name: akka-management
    containerPort: 8558
  - name: akka-metrics
    containerPort: 9001
livenessProbe:
  failureThreshold: 5
  httpGet:
    path: /health/alive
    port: akka-management
    scheme: HTTP
  initialDelaySeconds: 30
  periodSeconds: 10
  successThreshold: 1
  timeoutSeconds: 1
readinessProbe:
  failureThreshold: 3
  httpGet:
    path: /health/ready
    port: akka-management
    scheme: HTTP
  initialDelaySeconds: 30
  periodSeconds: 10
  successThreshold: 10
  timeoutSeconds: 1
env:
  - name: "POD_NAME"
    valueFrom:
      fieldRef:
        fieldPath: metadata.name
  - name: "POD_IP"
    valueFrom:
```

```
            fieldRef:
              fieldPath: status.podIP
          - name: "KUBERNETES_NAMESPACE"
            valueFrom:
              fieldRef:
                fieldPath: metadata.namespace
          - name: "IPF_JAVA_ARGS"
            value: "-Dma.glasnost.orika.writeClassFiles=false -
Dma.glasnost.orika.writeSourceFiles=false"
        resources:
          limits:
            memory: "2Gi"
          requests:
            memory: "2Gi"
            cpu: "1000m"
        volumeMounts:
          - mountPath: /ipf-tutorial-app/conf/logback.xml
            name: config-volume
            subPath: logback.xml
          - mountPath: /ipf-tutorial-app/conf/application.conf
            name: config-volume
            subPath: application.conf
      volumes:
        - name: config-volume
          configMap:
            name: ipf-tutorial-service-cm
```

the affinity rules (commented out), if used are a way of specifying where/on which nodes pods should be scheduled. This can be useful for ensuring that pods are scheduled on different nodes for example.

In this IPF Kubernetes deployment we have used the following:

securityContext for IPF pod

Liveness and readiness probe to monitor the health of the IPF pod.

Application and logging configuration files stored as Kubernetes ConfigMap items.

Create the deployment using kubectl.

kubectl apply -f deployment.yaml

Check the deployment status.

kubectl get deployments -n ipf-tutorial

Now, you can get the deployment details using the following command.

kubectl describe deployments --namespace=ipf-tutorial

RES1 - Resiliency and retry settings (HTTP)

Create your own project

BUILD1 - Creating a New Project

Throughout the tutorials so far, we have taken the example IPF tutorial project and worked through our example based of that. The key point here however is that we started from a pre-existing IPF project that was all setup and ready to use. Now we consider the question - "How do we create a brand new IPF project?".

For this we introduce the IPF Archetype.

Introducing the Archetype

The archetype is a maven archetype project that will bootstrap a brand-new project. It will create a new project structure that should be very familiar from the tutorial namely:

A domain-root module that contains all the MPS setup. A brand-new flow is generated as part of this for you, the same one we saw in the initial solution of the tutorial!

A local-e2-test module that contains a basic testing setup for the project.

An <project>-app module that contains a spring boot based starter project that is hooked into the application.

This should be very familiar from the tutorials.

Running the Archetype

Let’s run the archetype.

The first thing to note is that when run, the archetype will generate a new folder from the directory you are currently in, this folder must not exist. You’ll also need to make sure your current directory does not include any pom files.

When ready we run this command to start the archetype:

mvn archetype:generate -DarchetypeGroupId=com.iconsolutions.ipf.core.archetype -DarchetypeArtifactId=icon-archetype -DarchetypeVersion=1.*.*

You must supply the target version of both the archetype (archetypeVersion) and use the compatible IPF version (ipfVersion).

The versions must be compatible, please check and replace the numbers based on IPF's release documentation and/or the details for your target environment!

After a few seconds you should be prompted to provide the solution name.

Define value for property 'solutionName' (should match expression '^[A-Z][a-z0-9]+$'):

Here we note that it is telling us that the solution name must be alpha numeric and start with a capital letter, the rest being lower case. Let's enter a name of "Mytest" and then press return. Then we'll be prompted to:

Define value for property 'modelName' (should match expression '^[a-z]+$'):

Here we're being asked for the mps model name, this time it can only contain a lower case alphabetic expression. Let's enter a name of "mymodel" and then press return. Then we'll be prompted to:

Define value for property 'flowName' (should match expression '^[a-zA-Z]+$'):

Here we're being asked for the mps flow name, this time it can be any upper or lower case alphabetic expression. Let's enter a name of "Myflow" and then press return. Then we'll be prompted to:

Define value for property 'artifactId' (should match expression '^[a-z]+$'):

Here we're being asked to provide the Maven artifact id for the new project. Note that this will be the name of your overall project and it will also be the name of the directory into which all the application code is generated. As it's a maven artifact id, it needs to follow all the same rules for it. Let's enter a name of "mytest" and then press return. You'll be prompted to:

Define value for property 'groupId' (should match expression '^[a-z]+$'):

Here we're being asked to provide the Maven group id of the project. Let's enter a name of "mygroup" and then press return. Then we'll be prompted to:

Define value for property 'ipfVersion':

Here we're being asked for the core IPF version you want to use. The version must be compatible with the Archetype version, please check and use the correct versions in IPF's release documentation and/or the details for your target environment!

Let's enter that and press return. Then we'll be prompted to:

Define value for property 'version' 1.0-SNAPSHOT: :

Here we can provide the version for our new project, you can just press return to use the default provided (1.0-SNAPSHOT) but you can enter any valid Maven version number here. For now let's just use the default and press return. You'll then be prompted to:

Define value for property 'package' mygroup:

Here we can define the base package name for our application, by default we use the group id here but again if required you can overwrite it and provide your own value. Let's just use the default again and press return.

That's all the properties we need to enter, so at this point the archetype will print a list of all the properties it has been provided:

solutionName: Mytest

modelName: mymodel

flowName: Myflow

artifactId: mytest

groupId: mygroup

ipfVersion: 2023.1.0

includeApplication: y

includeMappings: n

includeSystemConnectors: n

includeMonitoring: y

groupId: mytest

artifactId: mytest

version: 1.0-SNAPSHOT

package: mygroup

Y: :

If these are correct, we can simply press return here to confirm. If they are not correct, we can enter an "N" and then press return which will allow us to review and update the properties. Let's confirm it by pressing return.

Now the archetype will go off and start generating the project for you, when it's complete you should see:

[INFO] Project created from Archetype in dir: /build/mytest

[INFO] --

[INFO] BUILD SUCCESS

[INFO] --

[INFO] Total time: 10.505 s

[INFO] Finished at: 2022-09-12T08:37:21+01:00

[INFO] --

So here we can see that I ran the archetype from the /build directory and as my artifactid was mytest it's told you that it's created the project in /build/mytest. Note that the time here show's the total time taken to run the archetype that includes all the time entering the properties above. The actual generation of the archetype code itself should take no more than a few seconds.

Once your familiar with the archetype and it's properties, you can use it in "batch" mode. By specifying the -B parameter and then entering all the required properties on the initial command, the archetype will run without any further manual intervention. For example, to do this for the project we've just created we could run (see Note above about choosing compatible versions):

mvn archetype:generate -B -DarchetypeGroupId=com.iconsolutions.ipf.core.archetype -DarchetypeArtifactId=icon-archetype -DarchetypeVersion=1.*.* -DgroupId=mytest -DartifactId=mytest -Dversion=1.0-SNAPSHOT -DipfVersion=202*.*.* -DsolutionName=Mytest -DmodelName=mymodel -DflowName=Myflow

Let's now look at the /build/mytest directory and we'll see:

➜ mytest ls -ltr

total 44

drwxrwxr-x 2 bensavage bensavage 4096 Sep 12 08:39 bin

drwxrwxr-x 3 bensavage bensavage 4096 Sep 12 08:39 mytest-app

drwxrwxr-x 8 bensavage bensavage 4096 Sep 12 08:39 domain-root

-rw-rw-r-- 1 bensavage bensavage 10284 Sep 12 08:39 mvnw

-rw-rw-r-- 1 bensavage bensavage 2412 Sep 12 08:39 Readme.md

-rw-rw-r-- 1 bensavage bensavage 2769 Sep 12 08:39 pom.xml

-rw-rw-r-- 1 bensavage bensavage 6734 Sep 12 08:39 mvnw.cmd

drwxrwxr-x 3 bensavage bensavage 4096 Sep 12 08:39 local-e2e-test

So here we can see we have the core folders we discussed above, the domain-root, local-e2e-test and mytest-app.

There is also a Readme.md file the contains information on how to run the generated project, have a look at that now.

Having generated our project, let's build it. Move to the base directory of the project and run:

mvn clean install

As with the main tutorial projects, this will generate all of our MPS code together with the docker images for the application itself. It will then run tests against the configured container environment to make sure everything is up and running ok. After a few minutes, we should get a successful build result.

That's it, we've successfully generated an IPF project and are ready to get going. The normal next step would be to open up the flow in MPS and start building it out. To do this we simply open the domain-root/mps folder in our MPS install.

Conclusions

In this section we've learnt how to create a brand-new project.

RUN1 - Running your application on Kubernetes

Sanctions Simulator

UTILS3 - Sanctions Simulator

This document covers how to use the sanctions simulator. There are two recommended approaches for the tutorial, both docker and non-docker. Let's look at these:

Using the Application Through Docker

Below provides a simple docker file entry for the developer application:

```
  sanctions-sim:
    image: registry.ipf.iconsolutions.com/sample-systems-sanctions-simulator-kafka:2.0.41
    container_name: sanctions-sim
    ports:
      - 5010:5005
      - 8088:55555
    environment:
      - SANCTIONS_MODE=normal
      - SANCTIONS_TRANSPORT=kafka
      - SANCTIONS_SIM_ENCRYPTION_ENABLED=FALSE
    volumes:
      - ./config/sanctions-sim:/sanctions-simulator-kafka/conf
      - ./logs:/ipf/logs
    user: "1000:1000"
    depends_on:
      - kafka
```

In addition, the application will need configuration which is placed in the application.conf file within a config directory relative to the above docker entry. The configuration file contains:

```
common-kafka-client-settings {
  bootstrap.servers = "kafka:9092"
}
akka.kafka {
  producer {
```

```
    kafka-clients = ${common-kafka-client-settings}
  }
  consumer {
    kafka-clients = ${common-kafka-client-settings}
  }
}
```

When executed the sanctions sim will listen for data on the SANCTIONS_REQUEST topic and post back to the SANCTIONS_RESPONSE. The logs will be exported to logs/ipf-developer.app relative to the docker file.

Using the Application Without Docker

Prerequisites

To run this simulator requires:

A Java 11 runtime

A Kafka environment that either has:

Auto topic creation enabled

Topics created for SANCTIONS_REQUEST and SANCTIONS_RESPONSE

Configuration Options

Property

Description

Default Value

common-kafka-client-settings.bootstrap.servers

Defines the server bootstrap settings for kafka.

localhost:9092

server.port

Defines the port the application will run on

8082

simulator.http.port

Defines the port upon which to listen for http requests.

Download

The sanctions simulator is available here:

nexus.ipf.iconsolutions.com/repository/ipf-releases-temp/com/iconsolutions/ipf/sample/samplesystems/sanctions-simulator-kafka/2023.1.0/sanctions-simulator-kafka-2023.1.0-runnable.jar

Running

Once downloaded, place it into this directory and run:

java -cp "sanctions-simulator-kafka-2023.1.0-runnable.jar:config" -D"loader.main"="com.iconsolutions.samplesystems.sanctions.simulator.SanctionsSimulatorKafkaApplication" "org.springframework.boot.loader.PropertiesLauncher"

The application will then start and be available from:

localhost:55555/index.html

Note that on Windows, the colon in the run command should be replaced by a semicolon.

Create your own project

Payment Initiation Simulator

UTILS2 - Payment Initiation Simulator

This document covers how to use the ipf-developer-app. There are two recommended approaches for the tutorial, both docker and non-docker. Let’s look at these:

Using the Application Through Docker

Below provides a simple docker file entry for the developer application:

```
  payment-initiation-simulator-kafka:
    image: registry.ipf.iconsolutions.com/sample-systems-payment-initiation-simulator-kafka:2.0.41
    container_name: payment-initiation-simulator-kafka
    ports:
     - "8082:55555"
    volumes:
     - ./config/payment-initiation-simulator-kafka:/payment-initiation-simulator-kafka/conf
    user: "1000:1000"
    depends_on:
     - kafka
```

In addition, the application will need configuration which is placed in the "application.conf" file within a config directory relative to the above docker entry. The conf file contains:

```
common-kafka-client-settings {
  bootstrap.servers = "kafka:9092"
```

```
}
connector {
default-receive-connector {
manual-start: true
}
  default-send-connector {
    manual-start = false
    call-timeout = 30s
    queue-size = 50
    max-concurrent-offers = 500
    resiliency-settings {
      minimum-number-of-calls = 1
      max-attempts = 1
      reset-timeout = 1s
      initial-retry-wait-duration = 1s
      backoff-multiplier = 2
    }
  }
  validator-error-handler-dispatcher {
    type = Dispatcher
    executor = "thread-pool-executor"
    thread-pool-executor {
      fixed-pool-size = 4
    }
  }
}
akka.kafka {
producer {
kafka-clients = ${common-kafka-client-settings}
}
consumer {
```

kafka-clients = ${common-kafka-client-settings}

}

}

When executed the ipf-developer-app will listen for data on port 8082. This port will also be exposed to the host machine. The logs will be exported to logs/ipf-developer.app-log relative to the docker file.

Using the Application Without Docker

Prerequisites

To run this simulator requires:

A Java 11 runtime

A Kafka environment that either has:

Auto topic creation enabled

Topics created for PAYMENT_INITIATION_REQUEST and PAYMENT_INITIATION_RESPONSE

Configuration Options

Property

Description

Default Value

common-kafka-client-settings.bootstrap.servers

Defines the server bootstrap settings for kafka.

localhost:9092

server.port

Defines the port the application will run on

8082

Download

The payment initiation simulator is available here:

nexus.ipf.iconsolutions.com/repository/ipf-releases-temp/com/iconsolutions/ipf/sample/samplesystems/payment-initiation-simulator-kafka/2023.1.0/payment-initiation-simulator-kafka-2023.1.0-runnable.jar

Running

Once downloaded, place it into this directory and run:

java -cp "payment-initiation-simulator-kafka-2023.1.0-runnable.jar:config" -D"loader.main"="com.iconsolutions.samplesystems.paymentinitiation.simulator.PaymentInitiationSimulatorKafkaApplication" "org.springframework.boot.loader.PropertiesLauncher"

The application will then start and be available from:

localhost:55555/index.html

Note that on Windows, the colon in the run command should be replaced by a semicolon.

Sanctions Simulator

Fraud Simulator

UTILS4 - Fraud Simulator

This document covers how to use the fraud simulator. There are two recommended approaches for the tutorial, both docker and non-docker. Let's look at these:

Using the Application Through Docker

Below provides a simple docker file entry for the developer application:

```
 fraud-sim:
   image: registry.ipf.iconsolutions.com/sample-systems-fraud-simulator-http:2.0.40
   container_name: fraud-sim
   environment:
    - FRAUD_SIM_ENCRYPTION_ENABLED=FALSE
   ports:
    - 8089:8080
    - 8090:55555
   volumes:
    - ./config/fraud-sim:/fraud-simulator-http/conf
    - ./logs:/ipf/logs
   user: "1000:1000"
```

The fraud simulator requires no specialised configuration beyond the defaults. When executed the fraud-simulator will listen for data on port 55555 on the docker network. This is mapped to port 8090 on the host machine. The logs will be exported to logs/fraud-sim.app-log relative to the docker file.

Using the Application without Docker

Prerequisites

To run this simulator requires:

A java 11 runtime

Configuration

Property

Description

Default Value

simulator.http.port

Defines the port upon which to listen for http requests.

55555

server.port

Defines the port the application will run on

8080

The fraud, payment-initiation and sanctions simulator will all listen on port 55555, so when running together on one host it is important to override the properties to give them unique ports.

Download

The fraud simulator is available here:

nexus.ipf.iconsolutions.com/repository/ipf-releases-temp/com/iconsolutions/ipf/sample/samplesystems/fraud-simulator-http/2023.1.0/fraud-simulator-http-2023.1.0-runnable.jar

Running

Once downloaded, place it into this directory and run:

java -cp "fraud-simulator-http-2023.1.0-runnable.jar:config" -D"loader.main"="com.iconsolutions.samplesystems.fraud.simulator.FraudSimulatorHttpApplication" "org.springframework.boot.loader.PropertiesLauncher"

The application will then start and be available from:

localhost:55555/index.html

Note that on Windows, the colon in the run command should be replaced by a semicolon.

Payment Initiation Simulator

Developer App

UTILS1 - IPF Developer App

This document covers how to use the ipf-developer-app. There are two recommended approaches for the tutorial, both docker and non-docker. Let’s look at these:

Using the Application Through Docker

Below provides a simple docker file entry for the developer application:

```
  ipf-developer-app:
```

```
  image: registry.ipf.iconsolutions.com/ipf-developer-app:1.0.29
  container_name: ipf-developer-app
  ports:
   - 8081:8081
  volumes:
   - ./config/ipf-developer-app:/ipf-developer-app/conf
   - ./logs:/ipf/logs
  environment:
   - IPF_JAVA_ARGS=-Dma.glasnost.orika.writeClassFiles=false -
Dma.glasnost.orika.writeSourceFiles=false -Dconfig.override_with_env_vars=true
  user: "1000:1000"
  depends_on:
   - ipf-mongo
  healthcheck:
   test: [ "CMD", "curl", "http://localhost:8081/actuator/health" ]
```

In addition, the application will need configuration which is placed in the application.conf file within a config directory relative to the above docker entry. The conf file contains:

```
flow-restart-settings {
 min-backoff = 1s
 max-backoff = 5s
 random-factor = 0.25
 max-restarts = 5
 max-restarts-within = 10m
}
spring.data.mongodb.uri = ${?ipf.mongodb.url}
actor-system-name = ipf-developer
ipf.mongodb.url = "mongodb://ipf-mongo:27017/ipf"
ods.security.oauth.enabled = false
application.write.url = "http://ipf-tutorial-app:8080"
ipf.processing-data.ingress.transport = http
```

When executed the ipf-developer-app will listen for data on port 8081. This port will also be exposed to the host machine. The logs will be exported to logs/ipf-developer.app-log relative to the docker file.

Using the Application Without Docker

Prerequisites

To run this simulator requires:

A Java 11 runtime

A MongoDB database

Configuration

Configuration is through the application.conf file located in the config directory.

Property

Description

Example Value

ipf.mongodb.url

Defines the mongodb URL the developer application should use.

"mongodb://ipf-mongo:27017/ipf"

application.write.url

Defines the host and port that the main ipf application will expose it’s query service on

ipf-tutorial-app:8080

server.port

Defines the port the application will run on

8081

Download

The ipf-developer-app is available in the ipf-releases Nexus repository, for example version 1.0.29 is:

nexus.ipf.iconsolutions.com/repository/ipf-releases/com/iconsolutions/ipf/developer/ipf-developer-app/1.0.29/ipf-developer-app-1.0.29-runnable.jar

You must choose and download the correct version of the ipf-developer-app for the IPF Release you are using.

The versions must be compatible, please check and replace the numbers based on IPF’s release documentation and/or the details for your target environment!

Running

Once downloaded, place it into this directory and run (replacing the version you are using):

java -cp "ipf-developer-app-1.*.**-runnable.jar:config" -D"ma.glasnost.orika.writeClassFiles"=false -D"ma.glasnost.orika.writeSourceFiles"=false -D"config.override_with_env_vars"=true -D"loader.main"="com.iconsolutions.ipf.developer.app.IpfDeveloperApplication" "org.springframework.boot.loader.PropertiesLauncher"

The application will then start and be available from:

localhost:8081/explorer.html

Note that on Windows, the colon in the run command should be replaced by a semicolon.

Fraud Simulator

Using Debulker

Using Debulker

What is a Debulker?

A Debulker is a application module responsible for splitting large files into components which will be used for processing. Debulker expects to receive a notification about the file that needs to be debulked, together with the configuration name that is going to be used to split the file into components.

Debulker in your application

First step is to add the debulker starter maven modules to your IPF application pom:

<!--Responsible for debulking-->

<dependency>

<artifactId>com.iconsolutions.ipf.debulk</artifactId>

<groupId>ipf-debulker-starter</groupId>

</dependency>

<!--Responsible for cleaning up after bulk components are processed by interested party-->

<dependency>

<artifactId>com.iconsolutions.ipf.debulk</artifactId>

<groupId>ipf-debulker-housekeeping-starter</groupId>

</dependency>

This modules depends on some additional modules:

component-store - responsible for storing components produced by debulker.

ipf-debulker-new-file-notification - responsible for consuming the file notification which should trigger debulking of the file.

ipf-debulker-archiver - responsible for archiving the bulk file after it was successfully processed.

ipf-debulker-client-processing - notifies an external system that debulking is finished and produced components can be processed. This also consumes notification of successful processing of those components by the external system, so that it can start performing housekeeping.

Adding dependencies needed for the starter modules

<!-- ipf-component-store implementation which uses mongodb to store and read components -->

<dependency>

<groupId>com.iconsolutions.ipf.componentstore</groupId>

<artifactId>ipf-component-store-mongo</artifactId>

</dependency>

<!-- Kafka receive connector implementation of ipf-debulker-new-file-notification which consumes FileNotification message which tells debulker to process bulk file -->

<dependency>

<groupId>com.iconsolutions.ipf.debulk</groupId>

<artifactId>ipf-debulker-new-file-notification-connector-kafka</artifactId>

</dependency>

<!-- ipf-debulker-archiver implementation which archives processed bulk file to local file system -->

<dependency>

<groupId>com.iconsolutions.ipf.debulk</groupId>

<artifactId>ipf-debulker-archiver-local</artifactId>

</dependency>

<!-- Kafka connector implementation of ipf-debulker-client-processing -->

<dependency>

<groupId>com.iconsolutions.ipf.debulk</groupId>

<artifactId>ipf-debulker-client-processing-connector-kafka</artifactId>

</dependency>

Configuring Debulker in your application

In order to split the file of specific type (xml, json…) and structure, configuration for splitting needs to be provided. This can be done by via ipf.debulker.configurations property. It expects an array of configuration objects, each one containing:

name (string) - used to uniquely identify the configuration. File notification will contain the configuration name which will be used to debulk the file.

splitter (string) - splitter type which will be used for extracting the components. Currently there are xml and json.

component-hierarchy (object) - tree structure representing the hierarchy of the components which will be extracted from the bulk file. Each node can have configured child nodes which will be extracted as separate components. The content of the child components will be ommited from the parent component.

Configuration example for debulking pain.001.001.09 XML file.

```
ipf.debulker {
  configurations = [
    {
      name = "pain.001.001.09"
      splitter = "xml"
      component-hierarchy {
        marker = "Document"
        children = [
          {
            marker = "CstmrCdtTrfInitn.PmtInf"
            children = [
              {
                marker = "CdtTrfTxInf"
              }
            ]
          }
        ]
      }
    }
  ]
}
```

This configuration tells us that the pain.001 XML file will be debulked into single Document component, which contains all child elements except CstmrCdtTrfInitn.PmtInf elements, which will be extracted as separate components. Each child PmtInf component will contain all child elements except CdtTrfTxInf elements, which will be extracted as separate child components of each PmtInf.

In case of a pain.001 XML file which has 3 PmtInf elements, each one containing 3 CdtTrfTxInf elements, debulker will produce 12 components:

1 Document component

3 PmtInf components

9 CdtTrfTxInf components

Configuring Archiver

Since we are using ipf-debulker-archiver-local for archiving, only thing that we need to configure is a path where bulk files will be archived:

ipf.debulker.archiving.path="/tmp/bulk_archive"

Docker Setup for ipf-debulker-tutorial-app

```
  ipf-debulker-tutorial-app:
    image: registry.ipf.iconsolutions.com/ipf-debulker-tutorial-app:latest
    container_name: ipf-debulker-tutorial-app
    ports:
      - 8080:8080
      - 8559:8558
      - 5006:5005
      - 55002:55001
      - 9002:9001
    volumes:
      - ./config/ipf-debulker-tutorial-app:/ipf-debulker-tutorial-app/conf
      - ./logs:/ipf/logs
    environment:
      - IPF_JAVA_ARGS=-Dma.glasnost.orika.writeClassFiles=false -
Dma.glasnost.orika.writeSourceFiles=false -Dconfig.override_with_env_vars=true
    depends_on:
      - ipf-mongo
      - kafka
    healthcheck:
      test: [ "CMD", "curl", "http://localhost:8080/actuator/health" ]
```

In order to work ipf-debulker-tutorial-app requires MongoDB and Kafka.

Running the application

You can start application.yml using next command:

docker-compose -f application.yml up -d

Testing the application

Now the application is started we can test it, this is done by:

Providing a source data file in the expected location.

Sending a FileNotification to the ipf-debulk, via Kafka, to notify the application a file is ready for processing.

Validating that the file is debulked.

Sending a ComponentProcessingCompleteCommand, via Kafka, to trigger housekeeping which will delete components from component store and remove the bulk file.

Validating that the housekeeping is performed.

Step 1 - Creation of pain.001 file which will be debulked

Since we have configuration for debulking pain.001 XML file, we will use that one for testing.

Pain.001 file sample.

```
<Document xmlns="urn:iso:std:iso:20022:tech:xsd:pain.001.001.09">
  <CstmrCdtTrfInitn>
    <GrpHdr>
      <MsgId>abc</MsgId>
    </GrpHdr>
    <PmtInf>
      <PmtInfId>1</PmtInfId>
      <NbOfTxs>2</NbOfTxs>
      <CdtTrfTxInf>
        <PmtId>
          <EndToEndId>1</EndToEndId>
        </PmtId>
        <Amt>
          <InstdAmt Ccy="GBP">500.00</InstdAmt>
        </Amt>
      </CdtTrfTxInf>
      <CdtTrfTxInf>
        <PmtId>
          <EndToEndId>2</EndToEndId>
        </PmtId>
      </CdtTrfTxInf>
```

```
    </PmtInf>
    <PmtInf>
      <PmtInfId>2</PmtInfId>
      <NbOfTxs>2</NbOfTxs>
      <CdtTrfTxInf>
        <PmtId>
          <EndToEndId>3</EndToEndId>
        </PmtId>
      </CdtTrfTxInf>
      <CdtTrfTxInf>
        <PmtId>
          <EndToEndId>4</EndToEndId>
        </PmtId>
      </CdtTrfTxInf>
    </PmtInf>
    <SplmtryData>
      <Envlp/>
    </SplmtryData>
  </CstmrCdtTrfInitn>
</Document>
```

File like this is already created and it is located in the solutions/add-debulker/docker/bulk_files/ directory.

Step 2 - Sending FileNotification to Kafka

We have configured the application to take FileNotifications from Kafka, that notification has several properties which must be provided:

configName - name of the configuration which will be used by debulker to debulk the bulk file into components.

bulkId - to correlate produced components by debulker.

fileProvider - name of the provider which will be used to retreive the bulk file for processing.

filePath - path to the file.

FileNotification message to send:

```
{
```

```
 "configName": "pain.001.001.09",
 "bulkId": "pain.001.12345",
 "fileProvider": "local",
 "filePath": "/tmp/bulk_files",
 "fileName": "pain_001_test.xml"
}
```

we can push the fileNotification to Kafka using Kafka console producer:

./kafka-console-producer.sh --topic FILE_NOTIFICATION_REQUEST --bootstrap-server localhost:9092

Message we are sending should be in one line:

{"configName": "pain.001.001.09", "bulkId": "pain.001.12345", "fileProvider": "local", "filePath": "/tmp/bulk_files/", "fileName": "pain_001_test.xml"}

Step 3 Validating that the file is debulked

At this point the debulker should have received the notification, accessed the file and debulked it. There are a couple of things we should check:

pain_001_test.xml file should exist in bulk_archive directory

components should be present in the component store

InitiateComponentProcessingCommand (notifies interested party that bulk components are ready for processing) is sent to kafka. Default topic is CLIENT_PROCESSING_REQUEST.

Easiest way to check components are via component store REST API.

You will need to add ipf-component-store-service maven dependency:

```
<dependency>
   <groupId>com.iconsolutions.ipf.componentstore</groupId>
   <artifactId>ipf-component-store-service</artifactId>
</dependency>
```

We should run next command from command line to get all components related to our bulk:

curl http://localhost:8080/v1/components/pain.001.12345 | json_pp

This should be the expected output for debulked pain_001_test.xml file (component number and their content should be the same):

```
[
  {
    "bulkId" : {
      "value" : "pain.001.12345"
```

```
    },
    "content" : "<CdtTrfTxInf><PmtId><EndToEndId>1</EndToEndId></PmtId><Amt><InstdAmt
Ccy=\"GBP\">500.00</InstdAmt></Amt></CdtTrfTxInf>",
    "creationTime" : "2023-02-27T14:15:52.788Z",
    "custom" : null,
    "id" : {
      "value" : "a78d25e5-3625-4fe6-86dd-b220b92d9e14"
    },
    "index" : 2,
    "marker" : "Document.CstmrCdtTrfInitn.PmtInf.CdtTrfTxInf",
    "parentId" : {
      "value" : "a78d25e5-3625-4fe6-86dd-b220b92d9e14"
    }
  },
  {
    "bulkId" : {
      "value" : "pain.001.12345"
    },
    "content" : "<CdtTrfTxInf><PmtId><EndToEndId>2</EndToEndId></PmtId></CdtTrfTxInf>",
    "creationTime" : "2023-02-27T14:15:52.845Z",
    "custom" : null,
    "id" : {
      "value" : "cc1a2d7c-4f94-4c94-a755-242c99881162"
    },
    "index" : 3,
    "marker" : "Document.CstmrCdtTrfInitn.PmtInf.CdtTrfTxInf",
    "parentId" : {
      "value" : "cc1a2d7c-4f94-4c94-a755-242c99881162"
    }
  },
  {
```

```
  "bulkId" : {
    "value" : "pain.001.12345"
  },
  "content" : "<PmtInf><PmtInfId>1</PmtInfId><NbOfTxs>2</NbOfTxs></PmtInf>",
  "creationTime" : "2023-02-27T14:15:52.851Z",
  "custom" : null,
  "id" : {
    "value" : "bd8f8a58-e9d8-4e16-805e-5b7b1579584c"
  },
  "index" : 1,
  "marker" : "Document.CstmrCdtTrfInitn.PmtInf",
  "parentId" : {
    "value" : "bd8f8a58-e9d8-4e16-805e-5b7b1579584c"
  }
},
{
  "bulkId" : {
    "value" : "pain.001.12345"
  },
  "content" : "<CdtTrfTxInf><PmtId><EndToEndId>3</EndToEndId></PmtId></CdtTrfTxInf>",
  "creationTime" : "2023-02-27T14:15:52.857Z",
  "custom" : null,
  "id" : {
    "value" : "ab312e01-9385-4846-93c4-75bdd0d94c66"
  },
  "index" : 5,
  "marker" : "Document.CstmrCdtTrfInitn.PmtInf.CdtTrfTxInf",
  "parentId" : {
    "value" : "ab312e01-9385-4846-93c4-75bdd0d94c66"
  }
},
```

```
{
  "bulkId" : {
    "value" : "pain.001.12345"
  },
  "content" : "<CdtTrfTxInf><PmtId><EndToEndId>4</EndToEndId></PmtId></CdtTrfTxInf>",
  "creationTime" : "2023-02-27T14:15:52.863Z",
  "custom" : null,
  "id" : {
    "value" : "fd9a777d-e012-44a5-8560-903cdafe65f6"
  },
  "index" : 6,
  "marker" : "Document.CstmrCdtTrfInitn.PmtInf.CdtTrfTxInf",
  "parentId" : {
    "value" : "fd9a777d-e012-44a5-8560-903cdafe65f6"
  }
},
{
  "bulkId" : {
    "value" : "pain.001.12345"
  },
  "content" : "<PmtInf><PmtInfId>2</PmtInfId><NbOfTxs>2</NbOfTxs></PmtInf>",
  "creationTime" : "2023-02-27T14:15:52.868Z",
  "custom" : null,
  "id" : {
    "value" : "f9086dbb-5f8e-43ee-bf83-5ee92f0255bb"
  },
  "index" : 4,
  "marker" : "Document.CstmrCdtTrfInitn.PmtInf",
  "parentId" : {
    "value" : "f9086dbb-5f8e-43ee-bf83-5ee92f0255bb"
  }
```

```
  },
  {
    "bulkId" : {
      "value" : "pain.001.12345"
    },
    "content" : "<Document xmlns=\"urn:iso:std:iso:20022:tech:xsd:pain.001.001.09\"><CstmrCdtTrfInitn><GrpHdr><MsgId>abc</MsgId></GrpHdr><SplmtryData><Envlp></Envlp></SplmtryData></CstmrCdtTrfInitn></Document>",
    "creationTime" : "2023-02-27T14:15:52.873Z",
    "custom" : null,
    "id" : {
      "value" : "c35a269e-4e58-4b16-83b5-32c2bb5000f4"
    },
    "index" : 0,
    "marker" : "Document",
    "parentId" : {
      "value" : "c35a269e-4e58-4b16-83b5-32c2bb5000f4"
    }
  }
]
```


Checking whether InitiateComponentProcessingCommand is sent to kafka by starting console consumer(this is the command which is sent to the client application/flow to inform it a bulk has been received, debulked and the components are ready for processing):

./kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic CLIENT_PROCESSING_REQUEST --from-beginning

There should be a single message which should look like this:


```
{"bulkId":"pain.001.12345"}
```


Step 4: Triggering debulker housekeeping

Next step would be to send the ComponentProcessingCompleteCommand to notify debulker that components are processed and debulker can perform housekeeping (delete the file, remove components from component store). Default topic for sending this message is CLIENT_PROCESSING_RESPONSE

Starting kafka console producer:

./kafka-console-producer.sh --topic CLIENT_PROCESSING_RESPONSE --bootstrap-server localhost:9092

Sending the message:

{"bulkId": "pain.001.12345"}

Step 5: Validating that housekeeping is performed

There are a couple of things we should check:

pain_001_test.xml file should be removed from bulk_files directory

all components related to bulkId="pain.001.12345" should be removed from component store

Conclusions

In this section we:

Successfully configured debulker in our IPF application

Processed a pain.001 XML file and validated that components are produced and file is archived.

Triggered housekeeping of bulk by deleting a bulk file and deleting components from component store.

Developer App

Releases

Release notes for recent IPF releases are found here. Please explore each release for detailed notes and links to other relevant sections within the documentation.

IPF-2023.1.0 - Release Notes

Migration Steps for IPF-2023.1.0

IPF-2023.1.0.1 - Fix Release Notes

Migration Steps for IPF-2023.1.0.1

IPF-2022.3.4-2 - Fix Release Notes

Migration Steps for IPF-2022.3.4.2

IPF-2022.3.4 - Release Notes

Migration Steps for IPF-2022.3.4

Reference

IPF-2023.1.0 - Release Notes
