Core Overview IPF is a collection of modules designed to work together to produce an overall application capable to supporting high throughput, low latency payment transactions. These modules aim to let the engineer focus on developing the business needs rather than complex technical details.

The following diagram shows a high level view of the overall IPF solution.

ipf developer view The 'Core' in terms of IPF is considered to be:

1.Orchestration - this is at the heart of the IPF Product. Orchestration is provided through Icon's flo-lang module.

2.Data Processing & Persistence - being able to manage data is crucial to the success of any IPF project. IPF provides it's own canoncial data model based on the ISO standards, but it's able to support any other types required too.

3.System Connectivity - being able to quickly and reliably integrate to other platforms is crucial for any payments platform. The connector framework provides a fluent approach to utilise the power of the alpakka framework.

4.Testing - testing of reactive message based systems can be hard. With this in mind, IPF provides the Icon Test Framework together with specific extensions and libraries for common IPF scenarios to simplify this complex area.

-------- Flo-Lang Orchestration is one of the key areas where IPF sets itself apart from competitors. We have built our own unique payments domain specific language (DSL) leveraging JetBrains MPS which enables clients to use it with minimal dependency on Icon.

Besides being a low-code language to quickly and flexibly model complex payment orchestration flows, one of the key benefits is that the code itself is generated automatically, along with test scenarios, graphical visualisation and documentation, meaning it is continually up to date and in sync.

The use of a payments domain specific language (DSL) simplifies and accelerates process definitions empowering the business whilst de-risking code delivery through alignment with documentation and testing.

Flo Life Cycle The key point to understand here is that by using flo-lang, developers and business analysts can come together to talk in one common language and define the required process in a way that both sides can be confident the right process is in place.

Once the process has been defined, flo-lang then takes care of both:

1.Automated generation of a self-contained Akka powered Java domain.

2.Automated generation of BDD testing scripts, graphical views and supporting documentation (always in sync)

Flo-lang provides a number of key features to support payment processing, including:

1.Support for the Icon Payments Model, a pre-built ISO20022 type library.

2.Support for custom business types.

3.'Fan-out/Fan-in' processing, for asynchronous processing

4.Customisable response & reason codes

5.Retry, Timeout and Exception handling

6.Process aborting and resuming

7.Process flow chaining

8.Ability to define sub-flows that are reusable multiple times within a single flow and across multiple flows

All these features and more are discussed at length in the following documentation.

We recommend starting at Concepts to understand key concepts of the DSL.

Once these are understood, you can get instructions on how to set up a DSL environment in Getting Started.

---------- Flo-Lang Concepts 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:

1.A name

2.A description

3.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.

4.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:

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

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

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

4.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.

NOTE: 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:

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

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

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

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

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

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

7.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:

1.A name

2.A description

3.A version

4.A global state set

5.A list of "States"

6.A list of "Events"

7.An "Initiation Behaviour"

8.A list of "Input Behaviours"

9.A list of "Event Behaviours"

10.A list of "Aggregate Functions"

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

NOTE:

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.

NOTE: 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:

1.A name

2.A description

3.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:

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

2.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")

3.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")

NOTE: 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).

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

NOTE: 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:

concepts 1

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:

concepts 2

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:

concepts 3

NOTE: 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:

concepts 4

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:

concepts 5

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:

concepts 6

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.

concepts 7

---------- Getting Started Flo-Lang is a DSL built with Jetbrains MPS that was designed to model orchestration workflows and generate various software components that can be used in a Java application. Currently, these are:

1.Akka Event-Sourced Domain model + Finite State Machine

2.Graphiz graphs for visualising the flow

3.Test-FW complaint BDD Test scenarios for the possible permutations through the flow

4.Embeddable documentation from the model, in the form of Asciidoc

MPS itself is traditionally IDE driven, so we have built a set of modules and configurations that allow transparent integration into downstream projects uses, providing that the downstream project is Maven based, and adheres to a set of conventions.

Note that whilst the MPS IDE (Or Intellij with MPS Plugins) is needed to create and modify solutions, it is not needed to compile existing solutions via the Maven build. The Maven build process actually downloads an MPS binary and compiles the solution in a headless mode.

Users have several options on workflow with regard to obtaining MPS.

1. Installing MPS Current working MPS version for command line builds and IDE is 2021.3.1.

The most common way to install MPS is via Jetbrains ToolBox as this manages installation of different versions.

getting started 1 An alternative is to use the MPS binary that gets downloaded and compiled against as part of the build process.

There is a shell script to start the IDE in the following equivalent folder:

<your_project_root>/<your_domain_root>/mps/target/mps/bin/mps.sh

Project Setup It is always advised that to create a new project setup you use the IPF archetype. This provides a simple mechanism to bootstrap a new project and gives you the ideal launch pad to create your payment flows.

Once your new project has been set up using the archetype, we can simply open the project in MPS by selecting File/Open and then navigating to <generated_archetype_root>/domain-root/mps.

---------- Code Reviewing How do I review code? MPS generates xml file which are not easy to understand especially if you are trying to review changes from them. If you view diffs in your VCS you will be shown a xml diff which is not very readable.

Luckily, MPS provides review integration into their IDE which makes thinks much easier and provides the opportunity to resolve conflicts in a human-readable way. The Git tab in MPS shows you the commits and by simply double-clicking on a change from another branch it will open a DiffPreview window that shows the changes from the editor perspective.

code reviewing 4 How do I use BitBucket to add review comments? MPS models are AST and persisted as an XMI, which makes them less trivial to code review remotely, meaningfully

There is a plugin for the MPS IDE which leverages BitBucket Server to allow meaningful code reviewing. (Code reviewing/adding/replying to comments is done in the IDE).

Firstly, create a Personal Access token in BitBucket from the Account page (it only needs read permissions), Copy the key.

code reviewing 1 Next, download both com.workday.mps.git4mps.zip and com.workday.mps.review.zip from github.com/Workday/mps-code-reviewer/releases. Once installed, hit the settings and there should now be an option for review integration that requires your access key and the repo url: bitbucket.iconsolutions.com/projects/IPFV/repos/flo-lang

code reviewing 2 Once set up the right-hand side pane should update as new pull request are added for the checked out branch.

The workflow is that to review someone else's code you need to check out the branch and then add comments in MPS.

Comments can be added to nodes or generally on the review if needed.

the branch name must match exactly the JIRA ID (no feature prefix) code reviewing 3 Best practices to minimize merge conflicts To minimise the chance of merge conflicts it's recommended to save the model per-root, which would create a xml file per root object (flow etc). To do this select the "Pre-installed XML-based File-Per-root Format" when creating a new model:

code reviewing 5

---- How do I use the generated code? This page explains how to get started with the code generated by the flo-lang project. It describes all the different interfaces that will be generated and reviews the central domain class used to interact with the domain itself.

The flo-lang language has three key levels to consider:

Solutions - this is effectively an overall project. A solution can contain many models.

Models - this represents a single self-contained domain. A model can contain many flows.

Flows - this is a single executable process flow.

From a generation perspective, generation is done at the model level.

This means that for each model an isolated set of code is generated that is not dependent on any other model. Each model will provide one central class for interacting with the domain, and that class can contain access to multiple different flows within it.

The Generated Interfaces Action Adapters: Connecting to things outside the domain For each external domain that is used within the model an action adapter

interface is generated. This class will provide a method which is what the domain will call when it determines it needs to interact with the external system.

The implementation of the action adapter methods, therefore is expected to include the code to physically pass the request onto the respective external domain:

@Override public CompletionStage<Void> execute(SampleExternalDomainAction action) { var externalDomainDto = convertToExternalDomain(action); return sampleExternalDomainConnector.send(action.getProcessingContext(), externalDomainDto) // during development it is sometimes useful to include debug information like this .thenAccept(report -> log.debug("Send completed with {}", report.getOutcome())); } Decision Adapters: Providing the business logic for the decision If the domain uses any decisions, then a decision adapter will be generated. The respective decision adapter methods will be called when the flow decides it needs to execute a decision, therefore the implementation should contain the business logic that is required to execute that decision correctly.

Domain Function Adapters: Providing the business logic for domain functions Domain functions are in reality very similar to external domains. The main difference is that whereas an external domain is expected to hand over control of processing to some external system, the domain function is expected to process that logic within it's own boundaries.

If domain functions are used within the model, then a domain function adapter interface will be generated providing method definitions for each domain function to be used.

Aggregate Function Adapters: Providing additional data manipulation within a flow An aggregate function provides the capability to hook into the arrival of an event within a flow and provide a utility to manipulate data. For each flow that uses aggregate functions, an interface will be generated. An implementation of these interfaces will be required.

Impacts of Aliasing An alias is a type of business data concept that allows the call to one of the capabilities defined above to be called in different circumstances using different data sets. As a result, when alias's are used then the generation may produce more than method on each interface, but with different parameter inputs to reflect the different aliases implementations.

Use of external models It is possible to define various supporting parts of a flow in a different model. For example, you may choose to define your external domains in a separate model that can then be reused from multiple different models rather than defining the same external domains repeatedly within different functional models. From a generation perspective, this is no different to the domains having been defined locally within the functional model itself.

Building the domain When a model is generated, it will generate a class that represents the domain as a whole. This class will be called <ModelName>v<VersionNumber>Domain (note that if there is no version number

then just the model name is used). This class provides all the functionality to interact with the domain.

Building the domain class A Builder is provided with the domain class, it allows for the provision to the domain of everything it needs to be able to work:

For each external domain used in the model, an implementation of the appropriate action adapter should be provided using the with<ActionAdapterName>ActionAdapter methods.

If decisions are required, an implementation of the model's decision adapter interface should be provided using the withDecisionAdapter method.

If domain functions are required, an implementation of the model's domain function adapter should be providing using the withDomainFunctionAdapter method.

When building the domain, the minimum that must be specified is:

The actor system for it to run on

Implementations of all the generated interfaces (external domains, domain functions & decisions)

The following shoes how a simple domain may be built.

new XYZDomain.Builder(actorSystem) .withDomainFunctionAdapter(new SampleXYZDomainFunctionAdapter()) .withDecisionAdapter(new SampleXYZDecisionAdapter()) .withABCFlowAggregateFunctionAdapter(new SampleABCFlowAggregateFunctionAdapter()) .withAnExternalDomainActionAdapter(new SampleExternalDomainActionAdapter()) .build(); When the build method is invoked, then all the flows within the model will be started.

Optional domain parameters When configuring the domain, in addition to the mandatory parameters defined above it is possible to define a number of optional attributes which, if not provided, are defaulted.

EventBus: the event bus implementation to be used when the domain raises system events.

ConfigAdapter

SchedulerAdapter

RetrySupport

EventProcessorSettings (per flow)

MetaDataExtractor (per flow)

Using the domain Handling responses from an external domain The domain class provides methods to be able to provide responses back into the domain. These are split out on a per external domain basis. To invoke these we call:

XYZDomain.externalDomainPort().handle(new ExternalDomainResponseInput()); The result of the handle method call is a CompletableFuture which completes once the flow has applied the input to the current state. The future carries with it a Done object, which provides a detailed view of what the end result of applying the input was:

@AllArgsConstructor @Getter public class Done implements CborSerializable { private final String aggregateId; private final String aggregateStatus; private final String commandName; private final Result result;

public enum Result { EXECUTED, DUPLICATE, UNEXPECTED, ERROR } } Out of the listed fields, the result is in most cases the important one to look at. Result it can have the following values, each representing a different outcome:

EXECUTED means that the input was successfully applied to the current state but offers no guarantees about the resulting operations (actions, notifications, domain functions)

DUPLICATE means that the input was deemed a business duplicate of a previously handled input and was thus not applied to the current state

UNEXPECTED means that the input was deemed inappropriate for the current state of the flow and was thus not applied to the state

ERROR means that an issue was encountered while executing one of the operations that were triggered as a result of applying the input (actions, notifications, domain functions)

Generally speaking, unless you have specific requirements (e.g. raising a system event on certain duplicate responses), you can ignore the outcome of applying the input and consider the operation a success as long as the future itself doesn't fail.

Handling of duplicate responses from an external domain By default, the external domain port that the domain provides comes bundled with configurable retry support. In order to ensure that retries are not interpreted as independent responses from the external domain, the port will assign random UUIDs as inputId and physicalMessageId if those values are not set on the input object.

inputId represents a unique business identifier (from the perspective of the flow) of an external domain's response and is the identifier based on which idempotent processing will be performed

physicalMessageId represents a unique transport identifier of an external domain's response, and it is useful for cases where the same physical message can be consumed multiple times (e.g. coming from message brokers), and the system needs to distinguish between a duplicate consumption (same message consumed twice) and a duplicate message (two messages with the same resulting inputId)

When using the IPF connector framework, all supported broker-based receive

connectors will contain a suitable PhysicalMessageId within their ReceivingContext object, which can then be used to populate the input.

If the flow receives two inputs with the same inputId, the second input will always skip execution and either result in a Done.Result.EXECUTED or a Done.Result.DUPLICATE outcome, depending on the physicalMessageId. If the physicalMessageId is also the same on both inputs, the second input is treated as a duplicate consumption and Done.Result.EXECUTED is returned. Otherwise, the input is treated as a duplicate message and Done.Result.DUPLICATE is returned.

While the default approach works for most scenarios, there are some corner cases where it may not be enough to ensure correct processing in your flows - e.g. when your flow handles the same input type from multiple non-exclusive states - thus requiring you to specify at least an inputId in order to avoid duplicate consumptions incorrectly triggering a flow transition:

XYZDomain.externalDomainPort().handle(new ExternalDomainResponseInput .Builder(id, ExternalDomainResponseCodes.OK) .withInputId(someInputIdFromResponseMessage) .build()); Calling another flow from an action When communicating between different flows that are part of the same logical unit of work, it is important to pass along the current flow's EventId to the other flow. An EventId carries within itself its owner flow's Lamport timestamp that the other flow will continue incrementing. Thus, by passing along an EventId from one flow to another, we enable domain events - belonging to different flows within a single logical unit of work - to be ordered in a causal, happened-before order.

Every generated Action provides access to the current EventId via its getCurrentEventId method, whereas every Input.Builder provides a withCausedByEventId that can be used to pass along the EventId of the collaborating flow:

@Override public CompletionStage<Void> execute(SampleExternalDomainAction action) { var externalDomainDto = convertToExternalDomain(action); return AnotherDomain.appropriateDomainPort().handle(new AppropriateDomainResponseInput .Builder(id, AppropriateDomainResponseCodes.OK) .withCausedByEventId(action.getCurrentEventId()) .build()); } Being able to determine the causal order of events in a distributed computer system is crucial when building read-side projections of a unit of work based on the domain events of its constituent flows. Without EventIds and their Lamport timestamps, projections would be forced to use system clock timestamps in their stead and as different nodes or processes will typically not be perfectly synchronized, clock skew would apply and incorrect event order would be arrived at as a result.

Informing the domain that a payment has timed out When a payment times out, it is necessary to tell the domain that this has occured. To do this, a helper class is available on the domain.

XYZDomain.timeout().execute(aggregateId, actionName); Ad-hoc functions within the domain The domain class also provides the ability to perform ad-hoc functions within the flow these are:

Aborting a payment

Accessing the payments status

Accessing the payments aggregate

These can all be done directly by calling the appropriate method on the domain class itself:

XYZDomain.abort(aggregateId, reason); XYZDomain.getStatus(aggregateId); XYZDomain.getAggregate(aggregateId); Identifying the available flows and versions To be able to clearly understand what flows are available within a generated domain, we can interogate the domain class. For each flow we have a static access available as per below:

XYZDomain.Flows.AFlow.name(); XYZDomain.Flows.AFlow.getAggregate(); XYZDomain.Flows.AFlow.getLatestVersion();

---------- How do I increase the memory of the MPS build? Depending on how complex your MPS flow is there could be the chance that you would get an OutOfMemoryError exception when building your flow through Maven. This would largely be due to fact that we generate BDD tests that have a test scenario per transitional state.

To increase the memory during the build and generation phase you can alter the "generation max heap size in mb" MPS Settings in the build script in as seen in the following screenshot:

memory 1 You also need to alter the heap size in the build.xml file of the root mps maven module as follows:

memory 2

---------- System Connectivity Connector Connectors provide an abstraction around communication transports which

aims to make integration simpler. This page introduces the core concepts used in the connector library.

What is a Connector? A connector is simply a set of common interfaces that provide a means to communicate with external systems, synchronously or asynchronously. Implementation details specific to the configured transport mechanism are hidden behind the interface. This helps to improve understandability and lets teams standardise integration instead of creating separate implementations each time a new system needs to be added.

The main benefit of using connectors is that they reduce the amount of boilerplate code required for integrating with systems which use common data interchange formats (such as XML or JSON) and transports (such as HTTP, JMS or Kafka).

Connector Features Connectors implement a number of features out of the box which are commonly required when integrating with external systems.

Backpressure

Message Throttling

Message Encryption

Message Validation

Message Association

Message Logging

Error Handling

Resilience

Metrics

Types of Connector Connectors can be used for either sending or receiving messages.

Detailed documentation specific to each type can be found in the following pages.

Sending Connector

Receiving Connector

Connector Stages Each connector is a composition of stages that a message flows through. The stages provide different capabilities, many of which are optional and can be tailored to specific requirements dependent on the situation.

Receive The purpose of a receiving connector is to take a message from the configured transport and map it into a known format which can then be handled. Received messages must be associated (correlated or identified) so that they can be linked to past/future messages.

The stages that can be configured for receiving are.

Payload Decryption (optional)

Conversion to Target Type

Message Association

Message Validation (optional)

Message Logging (optional)

Message Handling

See the Receive Stages section in the Receiving Connector documentation for more further details around each of the stages.

Send The purpose of a sending connector is to take a message and send it over the configured transport. Since networks are unreliable, the message sending can be configured to send over multiple transports (with load balancing or fallback strategies) and can employ circuit breaking and retries to overcome transient networking issues.

The stages which can be configured for sending are.

Message Validation (optional)

Message Correlation (optional)

Message Logging (optional)

Payload Encryption (optional)

Resilient Send

See the Send Stages section in the Sending Connector documentation for more further details around each of the stages.

Akka The Connector implementation uses Akka Streams. When a connector is built, the individual stages known in Akka Streams nomenclature as operators, are joined into a single stream.

The provided transport implementations use Alpakka, the Akka Streams subproject dedicated to integration. It is similar to Apache Camel in its goals; though is implemented with the actor model, that allows for highly scalable, concurrent processing.

The Alpakka project has implemented integrations with a large number of common technologies. For more information on what Alpakka supports, visit the following documentation.

---------- Concepts Receiving Connector

Sending Connector

Connector Transports

---------- Receiving Connector Receiving connectors are responsible for processing messages received on a configured transport and transforming the message into a known type, before delegating the handling of the message to the client.

All receiving connectors implement the ReceivingConnector interface, which itself extends OperableConnector.

public interface ReceivingConnector extends OperableConnector { }

public interface OperableConnector {

/** * Retrieve name of the connector. */ String getName();

/** * Starts the connector. */ void start();

/** * Starts the connector's health check procedure. */ void startHealthCheck();

ConnectorHealth getHealth();

/** * Shuts down the connector. * @param reason the reason for shutdown */ CompletionStage<Void> shutdown(ShutdownReason reason);

/** * Returns the connectors running status */ boolean isRunning(); } Generic Type T The ReceivingConnector interface is not concerned with types, though the default implementation ReceiveConnector<T> is generically typed with type T, where T is short for 'target type'. Messages received in the transport layer are passed to the connector wrapped in a TransportMessage object and are converted to the target type as soon as possible. This allows future stages to work with a known type which can optionally be validated to ensure data integrity.

Stages The connector implementation uses Akka Streams. Connectors are comprised of a number of configurable stages that are executed asynchronously, where each stage performs some transformation or checks to ensure the messages are valid before delegating control back to the client to handle the received message.

The image below roughly describes the stage-based approach when receiving messages. Note that some stages are optional and will be skipped if they are not configured when connector is built.

graphviz-diagram The following sections will briefly cover each stage.

Filtering Some connector transports such as JMS already have the functionality to filter a message (using JMS Selectors), but others such as Kafka or HTTP don't. For this reason the Connector framework offers a filtering functionality.

To learn how to filter messages, see the Encryption documentation.

Payload Decryption Sometimes transport encryption protocols, such as TLS are not sufficient. In these cases application level encryption can be applied to the messages themselves. When receiving an encrypted message, its payload must be decrypted before we can transform it to the target type.

To learn how to configure decryption, see the Encryption documentation.

Convert to Target Type The payload found in the TransportMessage object passed from the transport layer to the connector is untyped. The connector is responsible for mapping the payload type into a known type. This can be achieved by providing an implementation of the ReceiveTransportMessageConverter when building a receiving connector.

@FunctionalInterface public interface ReceiveTransportMessageConverter<T> { T convert(TransportMessage transportMessage); } Message Association Received messages are either a response to a request we made earlier or a new request. When setting up a connector to receive messages we typically know which of these scenarios is expected. If a message is a response to a previously sent request, then it should be correlated back to the original message using the correlation service. Otherwise, if the message is a request we should generate (or extract) an identifier for the message so that we can correlate it later.

To learn more about message association, see the Message Association documentation.

Message Validation When receiving messages, it often makes sense to make sure the message is valid before continuing processing. This can be in terms of schema validation or business logic validation. Connectors can optionally be configured to validate message by providing an implementation of the BeanValidator interface.

To learn more about message validation, see the Message Validation documentation.

Message Logging Message logging is a common requirement when building integrations across multiple systems, as it can aid in debugging or provide a comprehensive audit of all messaging. Connectors can optionally be configured to log messages by providing an implementation of the MessageLogger interface.

To learn more about message logging, see the Message Logging documentation.

Message Handling The final and arguably most important stage in the receiver flow is the message handling stage. This is where the message, after passing through decryption, validation, association and logging can finally be handled by the client code.

Handling the message is done by providing an implementation of the ReceiveHandler functional interface.

@FunctionalInterface public interface ReceiveHandler<T> { CompletionStage<Void> handle(ReceivingContext receivingContext, T payload); } The handle function takes a ReceivingContext and the generically typed payload and returns a void future.

Resilience The resilience of a given receive connector is fairly configurable and relies on two concepts: automated retries and a dead letter channel.

The image below shows some potential scenarios that may be encountered when processing a message and the mechanisms used to handle them.

receive connector resilliency Retries It is possible to configure a retry policy for receiving connectors by supplying an instance of ResiliencySettings. For more details, see the Resilience documentation.

The resilience settings are used to distinguish between recoverable (retryable) exceptions and their counterparts. If an exception is deemed recoverable, retries will be attempted according to the configuration.

Once the retries have been exhausted and no recovery was possible, the error and the connector message that caused it are then passed to the dead letter channel for further processing and the message is NACKd (negatively acknowledged).

Dead Letter The dead letter channel component is represented by the DeadletterAppender interface.

@FunctionalInterface public interface DeadletterAppender { CompletableFuture<Void> append(ReceiveConnectorException receiveConnectorException); } The general idea behind the appender is to allow clients to support potentially complex processing that may need to happen when messages fail to be received. Such as persisting them in an external storage, scheduling their reprocessing with a distributed scheduler, etc.

-- Sending connectors are responsible for taking messages supplied by a client and sending it over the configured transport mechanism.

All sending connectors implement the SendingConnector interface, which itself extends OperableConnector.

public interface SendingConnector<D, R> extends OperableConnector {

/** * Send a request via the connector without a supporting context */ default CompletionStage<R> send(@NotNull ProcessingContext processingContext, @NotNull D domainMessage) { return send(processingContext, domainMessage, SupportingContext.empty()); }

/** * Send a request via the connector with a full sending context * that allows users to supply a supporting context as well */ CompletionStage<R> send(@NotNull ProcessingContext processingContext, @NotNull D domainMessage, @NotNull SupportingContext supportingContext);

/** * Allows setting of the meterRegistry implementation for a connector * after connector build * This allows implementations to set the registry for all SendingConnector instances * at once instead of having to set them at a per connector bases at build time */ void setMeterRegistry(MeterRegistry meterRegistry); }

public interface OperableConnector {

/** * Retrieve name of the connector. */ String getName();

/** * Starts the connector. */ void start();

/** * Starts the connector's health check procedure. */ void startHealthCheck();

ConnectorHealth getHealth();

/** * Shuts down the connector. * @param reason the reason for shutdown */ CompletionStage<Void> shutdown(ShutdownReason reason);

/** * Returns the connectors running status */ boolean isRunning(); } Generic Types D and T Let's first start by describing what the D and T generic mean.

D is the Domain type. This is a message in our "internal" canonical data model

T is the Target type. This is a message converted to the "external" data model with which we're integrating.

Stages The connector implementation uses Akka Streams. Connectors are comprised of a number of configurable stages that are executed asynchronously, where each stage performs some transformation or checks to ensure the messages are valid before sending it to the transport mechanism. The image below roughly describes the stage-based approach when sending messages. Note that some stages are optional and will be skipped if they are not configured when connector is built.

graphviz-diagram The following sections will briefly cover each stage.

Message Validation When sending messages, we may need to check the validity of the payload, especially if any of it is user generated since this could contain errors. Connectors can optionally be configured to validate message by providing an implementation of the Validator interface.

To learn more about message validation, see the Message Validation documentation.

Message Correlation One of the limitations around sending and receiving message asynchronously is that responses to messages need to be correlated with one another. This can be achieved by providing an implementation of the CorrelationService interface when building a sending connector. The correlation

details will be persisted before sending the message and then can be retrieved later when the response is received in a separate receiving connector.

To learn more about message correlation, see the Message Association documentation.

Message Logging Message logging is a common requirement when building integrations across multiple systems, as it can aid in debugging or provide a comprehensive audit of all messaging. Connectors can optionally be configured to log messages by providing an implementation of the MessageLogger interface.

To learn more about message logging, see the Message Logging documentation.

Payload Encryption Sometimes transport encryption protocols, such as TLS are not sufficient. In these cases application level encryption can be applied to the messages themselves. When sending a message, its payload can optionally be encrypted before it is sent over the transport.

To learn how to configure encryption, see the Encryption documentation.

Resilient Send The final and only mandatory stage in the sending connector flow is the resilient sending stage. This is where we can configure how to handle failure scenarios typically encountered when sending messages over networks, by utilising resiliency strategies, such as retries, circuit breaking and re-routing.

To learn how to configure resilience settings, see the Resilience documentation.

Multiple Connector Transports Some systems represented by a connector can be reached over multiple transports. Example of this could be a series of non-clustered MQ queue managers or Kafka brokers. To cater to this use-case, it is possible to pass a list of connector transports when building a sending connector.

SendConnector<ExampleType, ExampleType> connector = SendConnector .<ExampleType, ExampleType>builder("ExampleSystem") .withConnectorTransports(List.of(transport1, transport2)) .withRoutingLogic(routingLogic) .withQueueSize(0) .withMaxConcurrentOffers(1) .withSendTransportMessageConverter(converter) .withCorrelationIdExtractor(correlationIdExtractor) .withActorSystem(actorSystem) .build(); Passes transport1 and transport2 to the connector. Configures the routing strategy.

The combination of withRoutingLogic and withConnectorTransports allows for load-balancing of requests over a connector across multiple transports, using various implementations of routing logic. The default routing logic is round-robin. For further details around this topic, see the Routing Logic section in the Resilience documentation.

Exactly one of the builder methods withConnectorTransports or withConnectorTransport should be called. If both are used then an IllegalArgumentException will be thrown when calling the build method.

Backpressure Backpressure is discussed in the Backpressure documentation page, check it out for more details on this topic. Streams are started (materialised)

with a queue. This means that it is possible to configure the amount of work that a connector, and by association the system with which it is communicating, can handle at a time. This is useful for backpressure situations where there is a fast producer which is producing at a rate much faster than what the downstream system (represented by this connector) can handle.

The queue size is configurable using the withQueueSize builder method.

SendConnector<ExampleType, ExampleType> sendConnector; sendConnector = new SendConnector .Builder<ExampleType, ExampleType>(connectorName) .withQueueSize(1) .withConnectorTransport(transport) .withSendTransportMessageConverter(messageConverter) .withCorrelationIdExtractor(correlationIdExtractor) .withActorSystem(actorSystem) .build();

1.The queue size argument.

It is possible to source this value from a placeholder, for example, to enable replacing this value from a configuration file.

If no queue size is provided, then the default queue size is defined by the DEFAULT_BUFFER_SIZE which can be found on the SendingConnector interface.

int DEFAULT_BUFFER_SIZE = 50; int DEFAULT_MAX_CONCURRENT_OFFERS = 500; int DEFAULT_PARALLELISM = 500; int DEFAULT_CALL_TIMEOUT_SECONDS = 30;

---------- Connector Transports The purpose of connectors is to abstract away the details of the underlying transport by placing the implementation behind a common set of interfaces. This enables us to plug in any transport mechanism that implements the required interfaces for sending and receiving messages and the connector core library takes care of all other aspects, such as mapping, validation, logging, etc.

When creating a new connector transport there are two interfaces that need to be implemented. These are SendingConnectorTransport and ReceivingConnectorTransport.

The connector library provides some of the most commonly used transport mechanisms ”out of the box”. These are for Kafka, JMS, HTTP and Filesystem, though custom transports can be developed with relative ease, and they should ”just work” with the connector library.

Receiving Transports Receiving connector transports implement the ReceivingConnectorTransport interface, that has the following signature.

public interface ReceivingConnectorTransport extends OperableConnectorTransport {

void startProcessingMessagesVia(Flow<ReceivedMessage<TransportMessage>, ReceivedMessage<Void>, ? extends KillSwitch> receiverFlow, Criteria filterCriteria);

default void acknowledge(ReceivedMessage<?> receivedMessage) { }

} A well implemented startProcessingMessagesVia method takes an unmaterialised flow and filter criteria. If filter criteria is null it will pass any received messages to the flow. If filter criteria is specified, only messages which fulfill the criteria will be passed down the flow.

The acknowledge method can optionally be implemented if the transport mechanism needs to acknowledge the receipt of a message.

Sending Transports Sending connector transports implement the SendingConnectorTransport interface, that has the following signature.

public interface SendingConnectorTransport<T> extends OperableConnectorTransport {

void startProcessingMessages(Integer maxConcurrentOffers);

CompletionStage<DeliveryOutcome> offer(MessageDelivery<T> messageDelivery); } The generic type T is "Target Type", i.e. the type of message to be sent to the target system, before serialisation.

The offer method takes a MessageDelivery<T> and returns a future containing the outcome of the message send. Typically, the offer method takes the message to send and adds it to an internal queue of messages that are to be sent to the transport mechanism.

The startProcessingMessage method should enable message to be sent via the transport, in most cases this is where the transport flow is materialised.

Custom Connector Transports To develop a custom transport, some basic knowledge about Akka Streams is required since this is ultimately what is used to drive the flow of messages through the system. For an introduction to Akka Streams, consider the Lightbend Akka Streams for Java course that is provided by the Lightbend Academy.

Features Connectors implement a number of features out of the box which are commonly required when integrating with external systems.

The following pages describe the features provided by the connector library. Examples and configuration details are provided, wherever possible.

Backpressure

Message Throttling

Message Encryption

Message Validation

Message Association

Message Logging

Error Handling

Resilience

Metrics

---------- Backpressure By virtue of using the Akka Streams library, the connectors are built to be reactive, i.e. demand-based.

This means we benefit from backpressure and are able to deal with heavy loads without a large performance hit, or systems crashing due to over-consumption of resources (provided the system is tuned correctly).

Akka Streams are composed of three main components which, together, form a runnable graph. These are sources, sinks and flows. Sources are producers (emit messages), sinks are consumers (receive messages) and flows are a combination of both (transform messages).

Consumers are able to signal demand for more messages, which is beneficial as this way consumers don't get inundated with messages. Using backpressure this way is most effective when messages are written to some intermediate data storage, i.e. event store, db, kafka topic or jms queue as this way peaks in message production won't affect the downstream consumers.

In some situations (e.g. HTTP transports), backpressure can cause requests to be rejected under heavy load as messages have nowhere to go if the consumers cannot process them fast enough. Though this situation is the same with or without backpressure. ----------

---

---------- Message Throttling The rate of processing messages may sometimes need to be throttled to avoid overloading downstream systems and even out spiky traffic. Both sending and receiving connectors can be configured to throttle the rate of messages they can process.

This is in addition to backpressure (demand signalling) we get by virtue of using Akka Streams in the implementation of the connector library. That is to say

if some part of the processing pipeline is running slowly then the backpressure would kick in and would effectively rate limit the system.

The difference with throttling is we can tune the maximum rate of messages over a window of time. This is especially useful if we have a slower downstream system which we don't want to overload.

Tuning Considerations The two parameters that can be tuned for throttling are the window duration and the message count. The throttling mechanism will allow messages through at whatever rate they come in up until the maximum number of messages have been reached within the window duration.

This is important because we could have two configurations that provide the same overall throughput, for example:

Window Duration Message Count Throughput (TPS) 10 seconds 1000 100

1 second 100 100

In the first example it is possible that 1000 message are processed within the first second, this would mean for the remaining nine seconds, nothing would be processed. Whereas in the second example, under the same load we would see a smoother rate of processing as the window is smaller.

Throttle Configuration To configure throttling on both sending and receiving connectors, a duration and count must be supplied when building the connector. If the throttle parameters are not set, then the connector will process messages as they come, unconstrained, albeit with backpressure from the downstream consumers.

initiatingReceiveConnectorBuilder() .withThrottleDuration(Duration.ofMillis(500)) .withThrottleCount(1);

---------- Message Encryption Sometimes messages require security above and beyond transport level security, such as SSL. For these cases connectors can be configured to encrypt or decrypt messages when sending or receiving messages, respectively. This places security at the application level in addition to any transport level security mechanisms that may be in place.

Crypto Interface To add message encryption, an implementation of the Crypto interface must be supplied when building the connector. If no Crypto implementation is provided then this stage is skipped and the transport message is received as is.

The interface is simple and consists of two methods, encrypt and decrypt.

public interface Crypto { /** * @param plaintext message to send * @param cryptoProperties properties to define encryption * @return encrypted message */ String encrypt(String plaintext, CryptoProperties cryptoProperties);

/** * @param payload encrypted message received * @param cryptoProperties properties to define encryption * @return decrypted message */ String decrypt(String payload, CryptoProperties cryptoProperties); } Crypto Configuration The connector library provides two implementations of the Crypto interface. These are:

NoopCrypto This is a plaintext passthrough class which will log the data at DEBUG level. It is generally used for testing.

NoopCrypto requires no configuration.

SymmetricCrypto Requires a key for encryption and decryption which is then used by the encryption methods. It is assumed that the key password is the same as the keystore password.

The SymmetricCrypto class requires some configuration in order to function. The following properties should be provided when building an instance of SymmetricCrypto.

Key Description keystorePath

The absolute path to the keystore

keystoreType

Keystore type such as PKCS12

keystorePassword

Password used for the keystore

transformation

Cipher transformation to be used, e.g. "AES/CBC/NoPadding" (See the Javadocs)

Using the SymmetricCrypto builder we can instantiate an instance.

SymmetricCrypto symmetricCrypto = SymmetricCrypto.builder() .withKeystorePath(keystorePath) .withKeystoreType("PKCS12") .withKeystorePassword("keystorepassword") .withTransformation("AES/CBC/PKCS5Padding") .build(); Usage Both send and receive connectors have a crypto field, which allows the developer to provide a Crypto implementation, or not if it is not required.

CryptoHeaders are required to ensure a key is passed and that encryption can be switched on or off at a message level.

Send Connector Example Crypto crypto = getSymmetricCrypto(); SendConnector .<ExampleType, ExampleType>builder(connectorName) .withActorSystem(actorSystem) .withConnectorTransport(connectorTransport) .with-

SendTransportMessageConverter(messageConverter) .withCorrelationIdExtractor(correlationIdExtractor) .withCorrelationService(correlationService) .withCrypto(crypto) .build(); Gets a Crypto implementation (symmetric in this example) Provides the Crypto implementation. Receive Connector Example Crypto crypto = new NoopCrypto(); ReceiveConnector .<ExampleType>builder(connectorName) .withActorSystem(actorSystem) .withConnectorTransport(connectorTransport) .withReceiveTransportMessageConverter(messageConverter) .withReceiveHandler(receiver) .withMessageLogger(messageLogger) .withProcessingContextExtractor(processingContextExtractor) .withCrypto(crypto) .build(); Instantiates the Crypto implementation (no-operation in this example) Provides the Crypto implementation. Crypto Message Headers When using Crypto with connectors, messages need to be sent with require the following headers. If they are not provided then the defaults values will be used instead.

Key Description Default keyAlias The key alias being used empty string

encryptionScheme The encryption scheme in use. Can be set to NOPS to disable encryption AES

The CryptoHeader enum is included in the API.

public enum CryptoHeaders { KEY_ID(”keyAlias”), SCHEME(”encryptionScheme”);

private final String headerName;

CryptoHeaders(String headerName) { this.headerName = headerName; } } Putting this all together we can create a method to instantiate the required message headers.

private MessageHeaders messageHeaders() { return new MessageHeaders() .putHeader(CryptoHeaders.KEY_ID.getHeaderName(), ”GWPAY01”) .putHeader(CryptoHeaders.SCHEME.getHeaderName(), ”AES”) .putHeader(”msgKey”, ”value1”); } Then set them on the transport message, alongside the payload in the SendTransportMessageConverter functional interface that can be declared with using a lambda.

private SendTransportMessageConverter<ExampleType> sendTransportMessageConverter() { return payload -> new TransportMessage(messageHeaders(), serialisePayload(payload)); } The SendTransportMessageConverter can be used when building a sending connector and all messages sent through it will have the CryptoHeaders set. This simple example demonstrates how to set the headers statically, however they can be set dynamically based off of the payload if this is required with minimal additional effort. ----------------------
----

--- Message Validation When working with messages sent between separate machines, even

when working against a specific schema, there can often be issues with the validity of the messages. These are often due to bugs, data entry errors or schema version mismatches. Both sending and receiving connectors can be optionally configured to validate messages before sending or after receiving.

Send Connectors To add validation to a sending connector, an implementation of the Validator interface must be provided when building the connector.

Validator.java public interface Validator { ValidationReport validate(TransportMessage transportMessage); } The connector library provides two validators, one for JSON Schema and another for XML Schema. Validation is performed against a TransportMessage before it is sent to the transport.

XML Schema Validation The XML validation implementation of Validator is XmlSchemaValidator. It takes an InputStream representing the schema to use against which all messages passing through this validator will be validated.

String xmlSchema = "/validation/sample-schema.xsd"; InputStream xmlSchemaStream = getClass().getResourceAsStream(xmlSchema); XmlSchemaValidator xmlValidator = new XmlSchemaValidator(xmlSchemaStream); SendConnector<ExampleType, ExampleType> sendConnectorWithXmlValidator = new SendConnector.Builder<ExampleType, ExampleType>(connectorName) .withActorSystem(actorSystem) .withConnectorTransport(transport) .withCorrelationIdExtractor(correlationIdExtractor) .withValidator(xmlValidator) .build(); Adds the xml schema validator to the connector In the above example, we are loading an XML Schema document from the classpath as an input stream. This is the most common usage pattern for XmlSchemaValidator. It is possible to set up using other types of input stream, such as a ByteArrayInputStream for loading the schema from a string.

The XML validator implements external entity injection (XXE) protection according to the OWASP cheat sheet on the topic.

JSON Schema Validation The JSON validation implementation of Validator is JsonSchemaValidator. It is configured similarly to the XML validator, where we must pass an InputStream representing the schema to validate messages against.

String jsonSchema = "/com/github/fge/jsonschema/examples/fstab.json"; InputStream jsonSchemaStream = getClass().getResourceAsStream(jsonSchema); JsonSchemaValidator jsonValidator = new JsonSchemaValidator(jsonSchemaStream); SendConnector<ExampleType, ExampleType> sendConnectorWithJsonValidator = new SendConnector.Builder<ExampleType, ExampleType>(connectorName) .withActorSystem(actorSystem) .withConnectorTransport(transport) .withCorrelationIdExtractor(correlationIdExtractor) .withValidator(jsonValidator) .build(); Adds the json schema validator to the connector The default JSON schema version is draft-04 which is the de facto standard.

Receive Connectors To add validation to a sending connector, an implementation of the BeanValidator interface must be provided when building the connec-

tor.

BeanValidator.java public interface BeanValidator<T> { BeanValidationReport validate(T message); BeanValidator<T> withConnectorName(String connectorName); } The connector library provides a validator for bean validation using the Hibernate validator implementation. Validation is performed against the message once it has been transformed from a TransportMessage to the target type by the provided ReceiveTransportMessageConverter function.

Bean Validator Messages are expected to be annotated with javax.validation annotations if they are to be validated with the bean validator implementation.

Bean validation is available with the BeanValidatorImpl. It optionally takes a javax.validation.Validator representing the validator to use against which all messages passing through this validator will be validated. If no validator is passed in, then a default validator is created.

public BeanValidatorImpl() { this(Validation.buildDefaultValidatorFactory().getValidator()); }

public BeanValidatorImpl(javax.validation.Validator validator) { this(null, validator); }

BeanValidatorImpl(String connectorName, javax.validation.Validator validator) { this.connectorName = connectorName; this.validator = validator; } Validation is optional and can be enabled by providing an implementation of BeanValidator when building the connector.

receiveConnector = initiatingReceiveConnectorBuilder() .withConnectorTransport(connectorTransport) .withEventBus(eventBus) .withBeanValidator(new BeanValidatorImpl<>()) .build(); JAVA copy icon Copied! Adds the bean validator to the connector

---

---------- Message Association In the context of connectors, association is a hypernym (umbrella term) for correlation and identification. Correlation is used to relate a response to a message sent asynchronously by another process, whereas identification is used to create an identity for new message which has no relation to any previous messages sent before.

Correlation Correlation Identifier Extraction When sending or receiving a message that requires correlation an implementation of CorrelationIdExtractor or ConnectorMessageCorrelationIdExtractor must be provided.

These are both functional interfaces, where the CorrelationIdExtractor can extract from the target message type and the ConnectorMessageCorrelationIdExtractor has access to headers passed from the transport layer which could be

used to store the correlation identifier. In both cases we return an instance of CorrelationId that is used to either persist or fetch the context.

@FunctionalInterface public interface ConnectorMessageCorrelationIdExtractor<T> { CorrelationId extract(ConnectorMessage<T> connectorMessage); }

@FunctionalInterface public interface CorrelationIdExtractor<T> { CorrelationId extract(T payload);

static <T> ConnectorMessageCorrelationIdExtractor<T> forConnectorMessage(CorrelationIdExtractor<T> correlationIdExtractor) { return connectorMessage -> correlationIdExtractor.extract(connectorMessage.getTargetMessage()); } } Send Correlation Messages sent using a sending connector must pass a ProcessingContext and can optionally pass a SupportingContext too, see the Send Connector documentation for more details around this.

Both these context objects will be persisted via an implementation of CorrelationService, whose purpose is store the context into a datastore against a correlation identifier. The default datastore implementation used for the correlation service is MongoDB, though this can be swapped out with any other repository, e.g. Redis, PostgreSQL.

Receive Correlation When messages are received in response to another message sent in a separate process, provided the receiving connector has been configured with a correlation extractor and a correlation service, correlation will be performed.

The correlation identifier will be extracted from the received message and is used to retrieve the context which was persisted during the send request. If the context is found it is passed through to later processing stages, otherwise an exception is thrown and passed to error handlers.

Identification Identification is used for initiating requests. These requests are not responses to messages sent previously, so they need to generate a ProcessingContext to be used going forward, since there is no context to fetch from the correlation service.

In some cases the message might be correlated in some way to previous messages and the generated ProcessingContext may need to reflect this.

Setting up identification for a receiving connector only requires an implementation of ProcessingContextExtractor. This makes identification flexible as the context can be either generated randomly or be derived from the received message.

@FunctionalInterface public interface ProcessingContextExtractor<T> { ProcessingContext extract(ConnectorMessage<T> connectorMessage); } Helper Class The InitiatingProcessingContextExtractor is an implementation of ProcessingContextExtractor which aims to reduce boilerplate, especially in the case

where want to either fully or partially generate random values for processing context fields.

For the simplest case the generateRandom static factory method can be used. Regardless of the input it will generate a ProcessingContext with a randomly generated UnitOfWorkId; and unknown ProcessingEntity and ClientRequestId.

ProcessingContextExtractor<TestObject> extractor = InitiatingProcessingContextExtractor.generateRandom(); Alternatively, the extractor can be constructed using the builder pattern and individual extractors for each field can be supplied, otherwise the defaults will be used.

var extractor = InitiatingProcessingContextExtractor.<TestObject>builder()
.unitOfWorkIdExtractor(message -> UnitOfWorkId.of(message.getTargetMessage().unitOfWorkId))
.clientRequestIdExtractor(message -> ClientRequestId.of(message.getTargetMessage().clientRequestId))
.processingEntityExtractor(message -> ProcessingEntity.of(message.getTargetMessage().processingEntity))
.build(); Since it is often the case that the ProcessingEntity does not change between request and remains static, there is static factory method defined to supply a static ProcessingEntity to cut down on the boilerplate code.

private static final ProcessingEntity PROCESSING_ENTITY = ProcessingEntity.of("processingEntity"); var extractor = InitiatingProcessingContextExtractor.<TestObject>builder() .processingEntityExtractor(staticSupplier(PROCESSING_ENTITY)) .build();

---------- Message Logging In many applications there are often requirements around keeping a record of every message, this could be for monitoring, auditing, or data warehousing purposes. Regardless of the use case, the connector library provides an optional message logging facility that will publish both sent and received messages.

Logged messages are published using the MessageLogEntry class defined in the message-logger-api.

Message Logger Interface To add message logging, an implementation of the MessageLogger functional interface must be provided when building the connector. The provided implementation can be more complex than demonstrated here and may publish to a database or a queue.

MessageLogger messageLogger = messageLogEntry -> { log.info("logging message: {}", messageLogEntry); }; initiatingReceiveConnectorBuilder() .withMessageLogger(messageLogger); Supporting context The MessageLogEntryEnricher class contains an element called supportingData. This has a key of type String and a value of type Object, and gets populated with a combination

of header elements that were added to the TransportMessage and then any passed supportingData, in that order.

Any key clashes between the two maps will be resolved by adding header details as they are, and clashing supportingData elements will get a supportingData- prefix, i.e. supportingData-kafka-key.

Message Log Entry Enrichment MessagesLogEntry objects can optionally be enriched before they are published by the MessageLogger.

public interface MessageLogEntryEnricher<T> { void enrich(ConnectorMessage<T> connectorMessage, MessageLogEntry messageLogEntry); } To add message log entry enrichment, provide an implementation of the MessageLogEntryEnrichment functional interface when building the connector.

MessageLogEntryEnricher<ExampleType> messageLogEntryEnricher = (connectorMessage, messageLogEntry) -> messageLogEntry.setSupportingData(Map.of("enriched", "true")); initiatingReceiveConnectorBuilder() .withMessageLogger(messageLogger) .withMessageLogEntryEnricher(messageLogEntryEnricher);

Error Handling Errors are an inevitable consequence when processing messages from external systems over unreliable networks. The connector library attempts to mitigate these with error handling strategies, which differ depending on the nature and context of the error in question.

The main distinguishing factor on how errors are handled is whether the error occurs whilst sending or receiving.

Error Handling in Send Connectors Error handling in the send connectors is mostly focused around handling cases where a message fails to be delivered over the transport. This class of error is transient in nature. To overcome these errors the send connector utilises the resilience4j library to wrap calls with circuit breaking, fallback/routing and retries, all of which can be configured. Refer to the Resilient Message Sending documentation for a more detailed explanation of how this works.

If a message fails to send, despite retries and other resilience strategies, then the message delivery completes exceptionally and the sender must handle the exception themselves as there is nothing more the connector can do without knowing about the client application.

Error Handing in Receive Connectors Exceptions can occur during receipt of a message, such as failing to deserialize a TransportMessage to the target type for example.

All failed messages are appended to a configurable deadletter, this way the failed message can be acknowledged and processing can continue.

Deadletter Appender The DeadletterAppender is a functional interface which is called whenever a message fails during processing.

@FunctionalInterface public interface DeadletterAppender { CompletableFuture<Void> append(ReceiveConnectorException receiveConnectorException); } Providing a DeadletterAppender implementation is optional and if one is not provided the connector will use the default implementation. The default behaviour is to simply log both the failed message and the exception that caused the error.

All failed messages will be provided as an exception that extends ReceiveConnectorException. ReceiveConnectorException wraps the original exception as the cause alongside the received message.

public class ReceiveConnectorException extends RuntimeException { private final ReceivedMessage<? extends ConnectorMessage<?» receivedMessage;

public ReceiveConnectorException(ReceivedMessage<? extends ConnectorMessage<?» receivedMessage, Throwable e) { super(e); this.receivedMessage = receivedMessage; }

public ReceiveConnectorException(ReceivedMessage<? extends ConnectorMessage<?» receivedMessage, String message) { super(message); this.receivedMessage = receivedMessage; }

} Exception Classification There are some exceptions that occur due to transient failures and for these it may make sense to attempt to recover from these. However, since the majority of connector functionality is built on passing immutable objects through functions, we can avoid attempting to retry failing operations and move on as fast as possible since the result should never change.

To account for both of these situations, we classify exceptions as recoverable or unrecoverable. Exceptions can be classified as unrecoverable by extending UnrecoverableReceiveConnectorException, a marker class which itself extends the base exception class ReceiveConnectorException. Otherwise, for exceptions which are recoverable, extend ReceiveConnectorException and add the exception to the recoverable-exceptions key in the configuration.

recoverable-exceptions is a list of RecoverableExceptionFilter objects consisting of the following:

When an exception is thrown the entire stack trace is scanned for Recoverable exceptions and if found, then the exception is classified as Recoverable. key definition example filter Regular expression to match against the fully ".*\\$HttpServerErrorException" qualified exception name e.g. HttpErrors$HttpServerErrorException

properties List of key-value pairs, all of which must be [{"statusCode": "503 Service Unavailable"}] matched in addition to the filter for the exception to be classified as a recoverable exception.

The default configuration is included below:

recoverable-exceptions = [{ filter = ".*\\$HttpServerErrorException" properties = [{"statusCode": "503 Service Unavailable"}] }, { filter = ".*\\.ReceiveConnectorException" }] Acknowledgements Depending on the transport implementation, acknowledgments may need to be made by the receiving connector. After handling an error the receiving connector will call the acknowledge method on the transport so that processing can continue.

Error Handlers For each stage we can optionally provide a function which takes a ReceiveConnectorException and returns a CompletionStage<Void>.

public DefaultMapToConnectorMessageStage(ReceiveTransportMessageConverter<T> transportMessageConverter, Integer parallelism, ExecutionContextExecutor dispatcher, ReceiveErrorHandler doOnError) { if (transportMessageConverter == null) { throw new IllegalArgumentException("'transportMessageConverter' must not be null"); } if (dispatcher == null) { throw new IllegalArgumentException("'dispatcher' must not be null"); } this.transportMessageConverter = transportMessageConverter; this.parallelism = Objects.requireNonNullElseGet(parallelism, () -> { final int processors = Runtime.getRuntime().availableProcessors(); log.debug("Using default parallelism of {}", processors); return processors; }); this.dispatcher = dispatcher; this.doOnError = doOnError; } This makes it easy to add error handling specific to each stage. We can also provide the doOnError parameter to the ReceiveConnector, which will be called on every ReceiveConnectorException in addition to the stage error handler.

Resilience Messaging is always a fraught process since networks and remote systems are not up 100% of the time. The connector library incorporates a number of strategies to cope with transient failures in external systems. The included strategies are, retries, circuit breaking and re-routing. As with other connector features, resilience is configurable; so as much, or as little of it can be used, dependent on the situation.

Retry One of the simplest resilience strategies is to retry the failed operation. This can resolve transient failures that may have caused a previous message send to fail or timeout.

Retries can be configured for both sending and receiving messages. For sending the retry decorates the message send over the transport. When receiving messages the client supplied receiver function is decorated with the retry.

Retry Parameters Where retries can get more complex is in determining what circumstances a retry is performed and when to give up. The answer to these questions are contextual and as such it has been made highly configurable so that it can fit the needs of most use cases.

To configure resilience settings as a whole the data class ResilienceSettings should be provided when building a connector. Below is table of all the resilience setting parameters that relate specifically to the retry mechanism.

Parameter Type Description Default maxAttempts Integer Determines the maximum number of retries 1 to be made. Note that this includes the first failed attempt.

initialRetryWaitDuration Duration How long to wait before retrying. This sets 1 second the initial duration and may increase on successive retry attempts due to the backoff multiplier.

backoffMultiplier Integer Each successive retry will wait the previous 2 wait duration multiplied by the backoff multiplier.

retrySchedulerThreadPoolSize Integer Determines how many threads the retry scheduler Equal to number of CPUs available can use.

retryOnFailureWhen Predicate<throwable>Given an exception thrown by the decorated code, Returns true for all exceptions returns a boolean to determine whether to retry. This is to avoid retrying on exceptions where multiple attempts will not resolve the failure.

retryOnResultWhen Predicate<Object> Given a successful result, i.e. no exception was Returns false for all results thrown, returns a boolean to determine whether to retry.This is used to trigger retries based on the object returned from the decorated code in a receiving connector.

retryOnSendResultWhen Predicate<Object> Given a successful result, i.e. no exception was Returns true when the returned DeliveryOutcome is a failure and retryOnFailureWhen returns true for the exception that caused the failure

thrown, returns a boolean to determine whether to retry. This is used to trigger retries based on the object returned from the decorated code in a sending connector. An example of where this may be useful is if a http server responds with a successful http response code to a request, but the response body contains error details that indicates a retry may be necessary.

Circuit Breaking Electronics Analogy Circuit breakers are based on the same named term in electronics. In electronics a circuit breaker is effectively a resettable fuse which cuts the power whenever dangerous amounts of current pass through. The circuit breaker keeps the power off until the problem is resolved at which point it can be reset.

Circuit Breakers in Connectors Circuit breakers used in messaging system, such as the connector library, work in very much the same way. Their purpose is to protect systems from being bombarded with messages while they attempt to recover.

If a connector transport is failing for an extended period of time, the circuit breaker may deem it unhealthy and will prevent any messages from being sent

to it. In circuit breaking terms this is referred to as an OPEN circuit, i.e. rejects messages fast, without sending anything to the downstream system. This gives the remote system some time to come back, and also prevents the sender from wasting time trying to send messages that will probably fail.

After some time passes, the circuit breaker may begin to allow some messages to be sent through to the transport. The circuit breaker is now considered to be HALF OPEN. If the transport continues to fail, then the circuit breaker goes back to OPEN and the circuit breaker will wait again before attempting to close again.

While in the HALF OPEN state; if messages are responded to successfully, then the circuit breaker will change state to CLOSED and will begin to function as normal.

Circuit Breaker Parameters Below is table of all the resilience setting parameters that relate specifically to the circuit breaker mechanism.

Parameter Type Description Default minimumNumberOfCalls Integer Determines the minimum number of calls 1 (within a sliding window period) that need to be made before the circuit breaker can calculate the error rate to determine the transport health.

resetTimeout Duration How long to wait while in the OPEN state before 1 second transitioning to HALF_OPEN and attempting to close the circuit.

Routing Logic Routing logic is another resilience mechanism specific to sending connectors. Sending connectors can be configured to send using one or more transports and routing logic determines the strategy used to select the transport to use when sending a message.

Routing logic ties in quite nicely with circuit breaking and typically the strategies will avoid selecting transports whose circuit breaker is not closed.

To configure the routing strategy, an implementation of the RoutingLogic interface must be provided when building a sending connector.

RoutingLogic.java @FunctionalInterface public interface RoutingLogic<T> { CircuitBreakerTransport<T> select(List<CircuitBreakerTransport<T» transports); } The connector library provides three RoutingLogic implementations, these are Failover, Round Robin and Weighted Round Robin. If no RoutingLogic is provided then the round-robin strategy will be used by default.

Failover The failover strategy is used with two transports, where one is the primary and the other a failover. The failover transport is selected when the primary transport is failing, i.e. has a non-closed circuit breaker.

Round Robin The round-robin strategy can be used with 1 or more transports, where each message sent will use the next transport, effectively load balancing equally across transports.

Weighted Round Robin Weighted round-robin is similar to the standard round-robin strategy. The main difference is that each transport is given a weighting that will cause more or less messages to be sent to it relative to the others.

--------- Metrics The connector library exposes a series of metrics which can be used to monitor the health of a particular connector.

By default, metrics are exposed using Prometheus. The default location where connector metrics are hosted is localhost:9001. If statistics are not displaying, then ensure that the Lightbend Telemetry agent is running. For more information check out the Cinnamon Agent Documentation.

Connector Metrics Breakdown Each connector has its own set of metrics in the table below. Every metric is tagged with the relevant connector's name. To distinguish which metrics relate to which connector, use the connector hint in the Prometheus exporter.

Metric name Type Description Source application_ipf_requests_sent

Counter

Logs the number of messages sent through a sending connector

Akka Cinnamon (default TCP/9001)

application_ipf_requests_received

Counter

Logs the number of messages received through a receiving connector

Akka Cinnamon (default TCP/9001)

application_ipf_correlations_saved

Counter

The number of correlation records persisted by the correlation service

Akka Cinnamon (default TCP/9001)

application_ipf_correlations_found

Counter

The number of correlation records fetched from the correlation service

Akka Cinnamon (default TCP/9001)

application_ipf_failed_correlations

Histogram/Recorder

The number of failed correlation lookups from the correlation service

Akka Cinnamon (default TCP/9001)

application_ipf_failed_requests

Counter

The number of failed message requests through a sending connector

Akka Cinnamon (default TCP/9001)

application_ipf_failed_receives

Counter

The number of failed message receives through a receiving connector

Akka Cinnamon (default TCP/9001)

application_ipf_response_times

Histogram/Recorder

A complex Recorder type which records min/max/average and percentiles for response times for this connector

Akka Cinnamon (default TCP/9001)

WARNING: These metrics are based on the CorrelationFound system event, so it only applies for async request reply operations where we found the correlation, so it's per-connector and only for connectors that participate in a request-reply session by using the correlation ID service Circuit Breaker Metrics Circuit breaker metrics use MeterRegistry implementations to publish metrics. This can be set at a per connector level through ResiliencySettings:

ResiliencySettings.builder() .initialRetryWaitDuration(Duration.ofMillis(10)) .maxAttempts(1) .resetTimeout(Duration.ofSeconds(1)) .meterRegistry(SIMPLE_METER_REGISTRY) .build() Alternatively this can be set through the SendingConnector interface. This allows setting the meterRegistry implementation at an application level.

For example, if we want to set all the sendingConnectors to have the same meterRegistry implementation we can loop through and set them with the API after we have already built the connectors. This allows us to quickly switch implementations without the need to change each connector definition.

Application Metrics Aggregated application level metrics are also exposed with the same mechanism

Metric name Type Description Event information Source ipf_behaviour_end_to_end_latency_seconds_bucke

Histogram

Time in seconds that a flow execution takes

behaviour - IPF flow name, event - time taken from start of flow until this event, type - one of the follow CSM_STATES_ONLY, NO_CSM_STATES or FULL_FLOW

Spring Actuator (TCP/8080)

ipf_behaviour_end_to_end_latency_seconds_count

Counter

Total number of invocations

same tags as used in ipf_behaviour_end_to_end_latency_seconds_bucket

Spring Actuator (TCP/8080)

ipf_behaviour_end_to_end_latency_seconds_sum

Counter

Total time spent by all invocations

same tags as used in ipf_behaviour_end_to_end_latency_seconds_bucket

Spring Actuator (TCP/8080)

ipf_behaviour_per_state_latency_seconds_bucket

Histogram

Time spent by flows in that state

behaviour - Tge flow name, status - status

Spring Actuator (TCP/8080)

ipf_behaviour_per_state_latency_seconds_count

Counter

Total number of calls to that state

behaviour - The flow name, status - status

Spring Actuator (TCP/8080)

ipf_behaviour_per_state_latency_seconds_sum

Counter

Time spent by all calls to that state

behaviour - The flow name, status - status

Spring Actuator (TCP/8080)

flow_started_total

Counter

Number of transactions have been created (started)

behaviour - The flow name, description - same as behaviour

Spring Actuator (TCP/8080)

flow_finished_total

Counter

Number of transactions that have reached a terminal (final) state

behaviour - The flow name, description - same as behaviour, reasonCode - reasonCode used in final state, state - name of the terminal state

Spring Actuator (TCP/8080)

state_timeout_total

Counter

Number of flows that have raised a timeout system event

behaviour - The flow name, description - state that timed out

Spring Actuator (TCP/8080)

action_invoked_total

Counter

Raised when the IPF domain invokes an action on an external system

behaviour - The flow name, description - called action name

Spring Actuator (TCP/8080)

action_timeout_total

Counter

Raised when an action invoked by IPF has not received a reply within the configured timeout

behaviour - The flow name, description - called action name

Spring Actuator (TCP/8080)

domain_event_persisted_total

Counter

Raised when a Domain Event has been successfully persisted

behaviour - The flow name, description - event name

Spring Actuator (TCP/8080)

unexpected_command_total

Counter

Raised when the IPF domain receives a command that cannot be handled in the current state of the aggregate

command_name - The command name, status - status of the flow when command recieved

Spring Actuator (TCP/8080)

ipf_processing_data_journal_latency_seconds_bucket

Histogram

which records the duration between the time a domain event has been created and the time it has been sent to ODS; the durations will be sensitive to time skew between the servers, so they should be treated as estimates only

Spring Actuator (TCP/8080)

ipf_processing_data_journal_latency_seconds_count

Counter

Count of the total number of domain events sent to ODS

ipf_processing_data_journal_latency_seconds_sum

Type Usage CSM_STATES_ONLY

Records only the time spent in CSM-related states (e.g clearing and settling, requesting status etc)

NO_CSM_STATES

Records the time between the first and last event in the flow, minus the time spent in CSM-related states (e.g clearing and settling, requesting status etc)

FULL_FLOW

Records the time between the first and last event in the flow

Grafana There are Grafana dashboards available as part of the standard IPF Grafana setup which graphs these data points meaningfully to the user.

Worked Example of Using PromQL to Generate a Custom Graph Requirement A pie-chart to represent all the CSM completion states of all transactions that happened in this calendar day that took less than the SLA of 5 seconds. — Business Analyst Step 1 - Choice of Metric First you should find the closest metric that would provide the data required, in this case the ipf_behaviour_end_to_end_latency_seconds histogram data should work.

sum by(event) (increase(ipf_behaviour_end_to_end_latency_seconds_bucket{type="CSM_STATES_ONI
service=~".*creditor.*", le="5.0"}[15m])) This query will return the number of states completed in 15 minute interval grouped by event

increase - calculates the increase in the time series in the range vector

sum - calculates the total in the time series

Further PromQL functions that can be used can be found here.

step2 Step 2 - Grouping into Days Now you can create a daily today with the following query

last_over_time(sum by(event) (increase(ipf_behaviour_end_to_end_latency_seconds_bucket{type=”CSM service=~”.*creditor.*”, le=”5.0”}[1d] offset -1d)) [1d:1d]) last_over_time - the most recent point value in specified interval

step1 Step 3 - Limit to a Single Day Alter the panel’s Query options to add a relative time of now/d

step3 Step 4 - Change the Graph Style Using the Visualization picker on the right hand side, choose the Pie Chart option, and alter the Title to Calls to Scheme. You will also be able to add the following changes:

step4a Add Piechart labels of Name

Change Legend mode to Table, set placement to Right and add Label Values of Percent and Value

You can now save the dashboard which will then look something like

step4b

---------- Getting Started Getting started guides for getting connectors up and running in as few steps as possible.

These can be augmented with additional features provided by the connector library to fit the specific needs of a particular use case.

Connector Operations Quickstart

Receiving Connector Quickstart

Sending Connector Quickstart

Connector Configuration

---------- Connector Operations Quickstart This page provides details on how to get started with the connector operations API.

OperableConnector Interface Before getting started, ensure the connectors you want to operate implement the OperableConnector interface. Without this, the API cannot perform operations on them.

public interface OperableConnector {

/** * Retrieve name of the connector. */ String getName();

/** * Starts the connector. */ void start();

/** * Starts the connector's health check procedure. */ void startHealthCheck();

ConnectorHealth getHealth();

/** * Shuts down the connector. * @param reason the reason for shutdown */ CompletionStage<Void> shutdown(ShutdownReason reason);

/** * Returns the connectors running status */ boolean isRunning(); } Getting Started To register the controller simply include the following maven dependency. Spring autoconfiguration will do the rest.

<dependency> <groupId>com.iconsolutions.ipf.core.connector</groupId> <artifactId>connector-operations-api</artifactId> <version>${connector-operations-api.version}</version> </dependency> The latest version of the connector operations library can be found using this Nexus search.

Auditing This section goes over how to set up audit logging on connector operator endpoints.

Prerequisites Auditing requests to the connector operations API assumes that the Spring security context has been configured and only authenticated users can make requests to protected resources. Without an authenticated user it is difficult to audit who made the request.

Reactive Web Setup Auditing is implemented with the Spring framework's reactive web stack. An auditing filter stage can be added as part of request handling by implementing the WebFilter interface and annotating it with @Component. An example of this is shown below.

@Slf4j @Component public class AuditLogFilter implements WebFilter {

private final ServerWebExchangeMatcher matcher;

public AuditLogFilter() { webExchangeMatcher = ServerWebExchangeMatchers.pathMatchers("/connectors/**"); }

@Override public Mono<Void> filter(ServerWebExchange exchange, WebFilterChain chain) { ServerHttpRequest request = exchange.getRequest(); return webExchangeMatcher.matches(exchange) .filter(ServerWebExchangeMatcher.MatchResult::isMatch) .flatMap(m -> exchange.getPrincipal() .doOnNext(principal -> logPrincipal(request, principal))) .then(chain.filter(exchange)); }

private void logPrincipal(ServerHttpRequest request, Principal principal) { log.info("{} {} principal.name: {}", request.getMethod(), request.getPath(),

principal.getName()); } } The web exchange filter allows us to only log on specific endpoints. The principal will only be logged if it is set on the exchange, otherwise nothing will happen. The logging can be customized, in this example it logs the request method and path with the principal's name. Blocking Web Setup Traditional blocking web servers need to be configured differently. Fortunately this is also quite simple as Spring has implemented a Filter that can be used for logging requests. This can also be configured to only work on specific endpoints if required.

The code snippet below shows how this can be set up by registering a couple of beans in a configuration class.

@Configuration public class AuditLogConfig { @Bean public FilterRegistrationBean<AbstractRequestLoggingFilter> loggingFilterRegistration(AbstractRequestLoggingFilter requestLoggingFilter) { var registration = new FilterRegistrationBean<>(requestLoggingFilter); registration.addUrlPatterns("/connectors/*"); return registration; }

@Bean public AbstractRequestLoggingFilter requestLoggingFilter() { CommonsRequestLoggingFilter loggingFilter = new CommonsRequestLoggingFilter(); loggingFilter.setIncludeClientInfo(true); loggingFilter.setIncludeQueryString(true); loggingFilter.setIncludePayload(true); loggingFilter.setMaxPayloadLength(64000); return loggingFilter; } } The filter registration bean can be configured to only log requests matching the provided url patterns. Here, the logging filter is configured. A custom implementation of the AbstractRequestLoggingFilter could be used instead if more control over the behaviour is required.

---------- Receiving Connector Quickstart This page provides details on how to get started with receiving messages from external systems, using receiving connectors provided by the connector library.

Dependencies Before building a receiving connector, the connector-core library must be included as a dependency.

<dependency> <groupId>com.iconsolutions.ipf.core</groupId> <artifactId>connector-core</artifactId> <version>${connector.version}</version> </dependency> The latest version of the connector library can be found using this Nexus search.

Unless providing your own implementation, at least one transport library should be declared. The naming scheme for all transports included in the connector library is connector-[transport], where [transport] matches the transport scheme that this connector should use. For more details on connector transports check out the Connector Transports documentation.

Here's an example of declaring the dependency to use JMS.

<dependency> <groupId>com.iconsolutions.ipf.core</groupId> <artifactId>connector-jms</artifactId> <version>${connector.version}</version> </dependency> Getting Started: Receiving Connector Receiving connectors are used for receiving messages, either as a response to a previously sent message or a new request.

Builder Pattern Receiving connectors are instantiated using the builder pattern. This is because connectors have many parameters to configure and most are optional or have default values.

Let's see how we use the builder pattern to instantiate a receiving connector.

When building a receiving connector we set it up to be either an initiating receiver or a response receiver. An initiating receiver receives requests from an external system, whereas a response receiver expects messages to be responses to requests made previously via a sending connector.

Initiating Receiver The following example demonstrates the minimum properties that must be provided when building an initiating receive connector.

ReceiveConnector<ExampleType> connector = ReceiveConnector .<ExampleType>builder("ExampleSystem") .withConnectorTransport(transport) .withReceiveTransportMessageConverter(converter) .withProcessingContextExtractor(processingContextExtractor) .withReceiveHandler(receiver) .withActorSystem(actorSystem) .build(); Sets the name of the connector. The name should represent what the connector is connecting to. Provides an implementation of the ReceivingConnectorTransport interface. Provides an implementation of the ReceiveTransportMessageConverter interface. Takes the received TransportMessage and converts it to the target type T (ExampleType in this instance). Provides an implementation of the ProcessingContextExtractor interface. This field is what makes this an initiating receiving connector as it extracts (or generates) a ProcessingContext from the message instead of fetching one from the correlation service as would be the case in a response receiving connector. An implementation of ReceiveHandler. This is where application logic would go to decide how to handle requests. Sets the actor system used throughout the application. Response Receiver The next example demonstrates how to build a minimal response receiving connector.

ReceiveConnector<ExampleType> connector = ReceiveConnector .<ExampleType>builder("connector-name") .withConnectorTransport(transport) .withReceiveTransportMessage-Converter(converter) .withCorrelationIdExtractor(correlationIdExtractor) .withCorrelationService(correlationService) .withReceiveHandler(receiver) .withActorSystem(actorSystem) .build(); Set the name of the connector. The name should represent what the connector is connecting to. Provides an implementation of the ReceivingConnectorTransport interface. Provides an implementation of the ReceiveTransportMessageConverter interface. Takes the received TransportMessage and converts it to the target type, ExampleType in this case. Provides an implementation of the CorrelationIdExtractor interface. Takes the received message and extracts the correlation identifier so that we can

correlate it with the original request made via a sending connector. Provides an implementation of the CorrelationService interface. The correlation service takes the extracted correlation identifier and returns the associated ProcessingContext used when the original request was sent via a sending connector. An implementation of ReceiveHandler. This is where application logic would go to decide how to handle responses. Sets the actor system used throughout the application. Start Receiving Messages The final step is to start the connector by calling its start method. At this point you should have a connector that can receive messages via the configured transport.

Connector Builder Helper In the previous sections we saw how to instantiate a receiving connector using the builder pattern. One way to reduce some boilerplate and share configuration is to use the ReceiveConnectorBuilderHelper and a configuration file.

To create a connector builder with default configuration, the ReceiveConnectorBuilderHelper.defaultBuilder static method can be used.

ReceiveConnector<Object> connector = ReceiveConnectorBuilderHelper .defaultBuilder("ExampleSystem", actorSystem) .withConnectorTransport(connectorTransport) .withReceiveTransportMessageConverter(transportMessageConverter) .withProcessingContextExtractor(processingContextExtractor) .withReceiveHandler(receiver) .build(); For reference, the default configuration is.

default-receive-connector { manual-start: true } To change any of the configuration, create a .conf file and pass it to the ReceiveConnectorBuilderHelper.builder static method.

application.conf example-receive-connector { manual-start = false throttle-count = 5 throttle-duration = 10s } ReceiveConnector<Object> connector = ReceiveConnectorBuilderHelper.builder("our-receive-connector", "example-receive-connector", actorSystem) .withConnectorTransport(connectorTransport) .withReceiveTransportMessageConverter(t -> t) .withProcessingContextExtractor(processingContextExtractor) .withReceiveHandler(receiver) .build(); The values that can be configured via configuration properties are shown in the following table.

Property Description Example manual-start

When set to 'false', connector is started automatically after creation

throttle-count

If the value is set, limits the throughput to a specified number of consumed messages per time unit. If it is set. If this value is set, throttle-duration also needs to be set.

10

throttle-duration

If set, it is used along with 'throttle-count' to set the maximum rate for consuming messages. For more details, see doc.akka.io/japi/akka/2.6/akka/stream/javadsl/Flow.html#throttle(int,jav

1s

mapping-parallelism

If set, limits the number of concurrent mapping operations executed on consumed messages. Defaults to the number of available processors.

receiver-parallelism

If set, limits the rate at which mapped messages are sent to behaviors for further processing. Defaults to the number of available processors.

---------- Sending Connector Quickstart This page explains details on how to get started with sending messages to external systems, using sending connectors provided by creating the connector library.

Dependencies Before building a sending connector, the connector-core library must be included as a dependency.

<dependency> <groupId>com.iconsolutions.ipf.core</groupId> <artifactId>connector-core</artifactId> <version>${connector.version}</version> </dependency> The latest version of the connector library can be found using this Nexus search.

Unless providing your own implementation, at least one transport library should be declared. The naming scheme for all transports included in the connector library is connector-[transport], where [transport] matches the transport scheme that this connector should use. For more details on connector transports check out the Connector Transports documentation.

Here's an example of declaring the dependency to use JMS.

<dependency> <groupId>com.iconsolutions.ipf.core</groupId> <artifactId>connector-jms</artifactId> <version>${connector.version}</version> </dependency> Getting Started: Sending Connector Sending connectors are used for sending messages to some destination, either as a reply to a previously received message or to make a request.

Builder Pattern Sending connectors are instantiated using the builder pattern. This is because connectors have many parameters to configure and most are optional or have default values.

Let's see how we use the builder pattern to instantiate a sending connector.

SendConnector<ExampleType, ExampleType> connector = SendConnector .<ExampleType, ExampleType>builder("ExampleSystem") .withConnectorTransport(transport) .withSendTransportMessageConverter(converter)

.withCorrelationIdExtractor(correlationIdExtractor) .withCorrelationService(correlationService) .withActorSystem(actorSystem) .build(); Sets the name of the connector. The name should represent what the connector is connecting to. Provides an implementation of the SendingConnectorTransport interface. Provides an implementation of the SendTransportMessageConverter interface. Takes the message payload of type T (ExampleType in this instance) and converts it to a TransportMessage object. Provides an implementation of the CorrelationIdExtractor interface. Takes the message payload and extracts (or generates) a correlation identifier to use when persisting the ProcessingContext via the correlation service. Provides an implementation of the CorrelationService. The correlation service takes a ProcessingContext and a CorrelationId and persists them using the services' configured repository. This is used to correlate a request with its response which will be handled by a separate process. Sets the actor system used throughout the application. Start Sending Messages At this point we should have successfully instantiated a SendConnector<ExampleType>, which can be used to send messages over the configured transport.

The SendingConnector interface defines two methods for sending messages, the signatures for these are.

public interface SendingConnector<D, R> extends OperableConnector {

/** * Send a request via the connector without a supporting context */ default CompletionStage<R> send(@NotNull ProcessingContext processingContext, @NotNull D domainMessage) { return send(processingContext, domainMessage, SupportingContext.empty()); }

/** * Send a request via the connector with a full sending context * that allows users to supply a supporting context as well */ CompletionStage<R> send(@NotNull ProcessingContext processingContext, @NotNull D domainMessage, @NotNull SupportingContext supportingContext);

/** * Allows setting of the meterRegistry implementation for a connector * after connector build * This allows implementations to set the registry for all SendingConnector instances * at once instead of having to set them at a per connector bases at build time */ void setMeterRegistry(MeterRegistry meterRegistry); } The generic domain message type D in this instance is our ExampleType. ExampleType has the following definition.

@Data @AllArgsConstructor public static class ExampleType { private final String name; private final LocalDate dob; private final int shoeSize; private final String context;

public ExampleType(String name, LocalDate dob, int shoeSize) { this(name, dob, shoeSize, null); } } We can now create an instance of the domain message and send it to the transport via the connector.

ExampleType message = new ExampleType("Isaac Newton", LocalDate.of(1642, 12, 25), 8); CompletionStage<DeliveryOutcome> future =

connector.send(processingContext, message); This returns a CompletionStage<DeliveryOutcome> which we can either block to await the DeliveryOutcome (not ideal), or use various methods on CompletionStage to perform other tasks asynchronously when the future completes.

Getting Started: Request-Reply Sending Connector The default sending connector works in a one-way fashion where the response of the request is the delivery result. However, for some protocols (namely HTTP) it is desirable and expected that a well-formed reply would be returned from a request.

A RequestReplySendConnector is built similarly to other connectors, using the builder pattern.

RequestReplySendConnector<ExampleRequest, ExampleRequest, ExampleResponse, ExampleResponse> connector; connector = RequestReplySendConnector .<ExampleRequest, ExampleRequest, ExampleResponse, ExampleResponse>builder("ExampleSystem") .withConnectorTransport(transport) .withCorrelationIdExtractor(correlationIdExtractor) .withSendTransportMessageConverter(sendMessageConverter) .withReceiveTransportMessageConverter(receiveMessageConverter) .withActorSystem(actorSystem) .build(); This connector is slightly more complex as it requires converting to and from the TransportMessage. However, some complexity is reduced because there is no longer any need for correlation related parameters.

The signature of the send method now returns a future containing the response. Since the transport is unreliable, a fail-safe exists to ensure that the request eventually completes, albeit with an exception after some timeout duration. A default is provided, but can be configured with the builder method, .withCallTimeoutSeconds.

When a message send times out, the response future will complete exceptionally with a TimeoutException as the cause.

Connector Builder Helper In the previous sections we saw how to instantiate sending connectors using the builder pattern. One way to reduce some boilerplate and share configuration is to use the SendConnectorBuilderHelper and a configuration file.

To create a connector builder with default configuration, the SendConnectorBuilderHelper.defaultBuilder static method can be used.

SendConnector<Object, Object> connector = SendConnectorBuilderHelper .defaultBuilder("ExampleSystem", actorSystem) .withConnectorTransport(connectorTransport) .withDomainToTargetTypeConverter(domainToTargetTypeConverter) .withCorrelationIdExtractor(correlationIdExtractor) .withCorrelationService(correlationService) .withSendTransportMessageConverter(transportMessageConverter) .build(); For reference, the default configuration is.

default-send-connector { manual-start = false call-timeout = 30s queue-size = 50 max-concurrent-offers = 500 parallelism = 500 resiliency-settings { minimum-

number-of-calls = 1 max-attempts = 1 reset-timeout = 1s initial-retry-wait-duration = 1s backoff-multiplier = 2 } } To change any of the configuration, create a .conf file and pass it to the SendConnectorBuilderHelper.builder static method:

application.conf example-send-connector { parallelism = 1000 throttle-count = 5 queue-size = 100 resiliency-settings { max-attempts = 10 } } SendConnector<Object, Object> sendConnector = SendConnector-BuilderHelper .builder("send-connector", "example-send-connector", actorSystem) .withConnectorTransport(connectorTransport) .withDomainTo-TargetTypeConverter(domainToTargetTypeConverter) .withCorrelationIdExtractor(correlationIdExtractor) .withCorrelationService(correlationService) .withSendTransportMessageConverter(transportMessageConverter) .build(); The values that can be configured via configuration properties are shown in the following table.

Property Description Example manual-start

When set to 'false', connector is started automatically after creation.

false

queue-size

Size of a source queue which can be used to handle backpressure, for example fast producer situations.

50

call-timeout

Maximum duration to wait for an acknowledgement before completing the returned future exceptionally with a TimeoutException.

30s

max-concurrent-offers

Maximum number of pending offers when buffer is full.

500

parallelism

Maximum number of parallel calls for send connector

500

throttle-count

Limits the throughput to a specified number of consumed messages per time unit. When this value is set, throttle-duration must also be provided.

10

throttle-duration

Is used with 'throttle-count' to set the maximum rate for consuming messages. For more details, see the Message Throttling documentation.

1s

resiliency-settings

The resiliency settings that will be used when sending. For more details, see the Resilience documentation.

Connector Configuration The configuration for a Connector can be set in IPF configuration files. This section explains the various configuration options available for the various types of connector, their default values, and the fallback mechanism which allows you to only supply a partial config and otherwise use the defaults documented below.

Configuration Options Sending Connectors The allowed configuration options for a sending connector (send connector & request-reply connectors) are:

Property Name

Description

Type

Default Value

manual-start

If true, the connector will not start on application startup but will require an invocation to it's 'start' method.

boolean

false

call-timeout

The length of time the connector will wait until it timesout the call (in seconds)

Integer

30

queue-size

TBC

Integer

50

max-concurrent-offers

TBC

500

parallelism

TBC

500

send-message-association

Defines whether the connector should add (as headers) the processing context information of the outbound request

true

throttle-count

TBC

Integer

Not set

throttle-duration

TBC

Duration

Not set

resiliency-settings

TBC

ResiliencyConfig

Receiving Connectors The allowed configuration for a receiving connector is:

Property Name

Description

Type

Default Value

manual-start

If true, the connector will not start on application startup but will require an invocation to it's 'start' method.

boolean

true

mapping-parallelism

TBC

Integer

Not set

receiver-parallelism

TBC

Integer

Not set

throttle-count

TBC

Integer

Not set

throttle-duration

TBC

Duration

Not set

resiliency-settings

TBC

ResiliencyConfig

default resiliency config

Resiliency Settings Both the sending and receiving connector config allows definition of the resiliency config with a 'resiliency-settings' block. The options for this are:

Property Name

Description

Type

Default Value

enabled

Allows resiliency to be toggled on / off

boolean

true

minimum-number-of-calls

TBC

Integer

1

max-attempts

TBC

Integer

1

reset-timeout

TBC

Duration

1s

initial-retry-wait-duration

TBC

Duration

1s

backoff-multiplier

TBC

Integer

2

retry-on-failure-when

TBC

boolean

true

retry-on-result-when

TBC

boolean

false

Default Configuration All connectors come with a pre-defined set of base configuration which will be used if not overriden at the point of construction of the connector.

The default configuration is provided below, and matches that defined in the previous section.

ipf.connector {

default-resiliency-settings { enabled = true minimum-number-of-calls = 1 max-attempts = 1 reset-timeout = 1s initial-retry-wait-duration = 1s backoff-multiplier = 2 retry-on-failure-when = true retry-on-result-when = false }

tag::default-receive-connector-config[] default-receive-connector { manual-start: false resiliency-settings = ${ipf.connector.default-resiliency-settings} } # end::default-receive-connector-config[]

tag::default-send-connector-config[] default-send-connector { manual-start = false call-timeout = 30s queue-size = 50 max-concurrent-offers = 500 parallelism = 500 send-message-association = true resiliency-settings = ${ipf.connector.default-resiliency-settings} } } Using Configuration When constructing a connector, it is possible to override these properties by providing a configRoot. When provided, the connector will use this configRoot as the base of the properties to use. So for example, if we construct a connector like:

var sendConnector = new SendConnector.Builder<String, String>("TestConnector") .withActorSystem(actorSystem) .withConfigRoot("test-config") .build(); Here we are supplying both:

the ActorSystem the config root - a string value. We are telling our connector that our config root here is test-config. What this means is that where in our default connector settings, the send connector defaults are at ipf.connector.default-send-connector, here we will use test-config as the primary config and then ipf.connector.default-send-connector as the fallback.

To illustrate this, suppose we were to set the config file like:

test-config { parallelism = 700 manual-start = false } Here we are defining new configuration values for parallelism and manual-start. We could just as easily have provided the config values for the other properties. By only supplying these two, our connector will then have:

A parallelism of 700

A manual-start setting of false.

All other properties inherited from the default config, so for example the queue-size will be set to 50.

We can also supply overrides to the configuration settings in the constructor through our java code. So for example, consider the following setup:

var sendConnector = new SendConnector.Builder<String, String>("TestConnector") .withActorSystem(actorSystem) .withParallelism(700) .withManualStart(false) .build(); This would lead to the same connector being built as per our configuration implementation.

------ Asynchronous Request-Reply How do I correlate messages from my own internal format to some external domain, if responses are async?

This Connector example connects to a fake service which takes a person's name and returns their age. The service on the other end takes requests on a request queue, and sends responses back on a response queue with a correlation ID to help the consumer determine which request this response is for.

Diagram This guide is also available in the separate connector-samples Git repository here. Correlation We use an in-memory correlation service to store mappings from our "internal" flow ID concept to the external "correlation ID" concept.

There might be multiple reasons why we cannot use our own internal ID as the correlation ID for an external service (hence the need for a correlation ID service).

I don't want to publish my internal ID to the outside world.

Technical limitations, e.g. external system's correlation IDs must follow a specific format.

There are multiple interactions with this external system for a single one of my internal ID, which would make correlation non-unique per individual invocation.

Connector Setup We have set up a SendConnector and ReceiveConnector pair. First the SendConnector.

var sendConnector = new SendConnector .Builder<AgeRequest, TheirAgeRequest>("OurSender") .withActorSystem(actorSystem) .withCorrelationService(correlationService) .withConnectorTransport(sendingTransport) .withMessageLogger(logger()) .withCorrelationIdExtractor(request -> CorrelationId.of(request.getCorrelationId())) .withDomainToTargetTypeConverter(this::createTheirAgeRequest) .withSendTransportMessageConverter(toJson()) .build(); Telling the SendConnector where to look in the target message type for the correlation ID to save in the Correlation ID service. Creating an external domain request with a random number as the external Correlation ID. Converting the external domain POJO type to a JSON representation. And here's the ReceiveConnector.

new ReceiveConnector .Builder<TheirAgeResponse>("OurReceiver") .withActorSystem(actorSystem) .withConnectorTransport(receivingTransport) .withMessageLogger(logger()) .withCorrelationService(correlationService) .withCorrelationIdExtractor(response -> CorrelationId.of(response.getCorrelationId())) .withReceiveTransportMessageConverter(fromJson(TheirAgeResponse.class)) .withManualStart(false) .withReceiveHandler((receivingContext, response) -> { var processingContext = receivingContext.getProcessingContext();

ageMap.put(processingContext.getUnitOfWorkId(), response.getAge()); return CompletableFuture.completedFuture(null); }) .build(); Function to tell the ReceiveConnector where in the response message to look for the correlation ID. Converting the TransportMessage from JSON to a TheirAgeResponse POJO. Populating the age in the map for our flow ID. Exercise At the moment this example uses an in-memory Map implementation of CorrelationIdService to store and retrieve correlation IDs. If this application were deployed in a distributed environment then it would not work in a multi-node setup.

You may have noticed that this sample starts an ActiveMQ container using Testcontainers. Try the following.

Tell Testcontainers to also start a database container (RDBMS, MongoDB, etc).

Create a new CorrelationIdService implementation which saves and stores correlation IDs to and from your newly. created database container, and wire it into the test

Create a new MessageLogger implementation which saves logged messages to this database, and wire it into the test.

---------- Chained Request-Reply with OAuth How do I use the output of one synchronous call as the input to another synchronous call?

This connector example demonstrates the following features of connectors.

Message logging

Chaining request-reply connector calls (output of one call as an input to the next)

Request decoration for security

Media upload

Diagram This guide is also available in the separate connector-samples Git repository here. Setup This example was originally written back when it was much simpler to get started developing applications for the Twitter API. Unfortunately there are some hoops we must jump through to get things set up before running the example, but hopefully it'll be worth it!

Twitter Account To run this example, you will need a Twitter account. If you already have one then skip to the next step.

You can sign up to Twitter using this link.

Twitter Developer Account You also need a developer account as we will be creating a Twitter application to call APIs on behalf of your user.

You should be able to sign up for a developer account by going to the developer portal.

Twitter App Once you have a developer account you should be able to create a new App from this page.

Elevated Access To use the upload feature, our app must have elevated API access. To get this we must apply for it, though it just a short questionnaire to make sure we don't have any nefarious plans with all of their data.

Elevated access can be applied for on this page.

API Keys At this point our Twitter app should be ready to go. All we need to do is generate some credentials that can be used to authenticate requests to Twitters APIs on behalf of us. The page for the app should have a tab for "Keys and tokens" where all the credentials can be generated.

The first of the credentials are the Consumer Keys. You may have saved them when you first created the app but if not you can regenerate them.

The second set of credentials is an Access Token and Access Token Secret, which need to be generated.

Once all the credentials have been generated, the following environment variables need to be set using them.

.setOAuthConsumerKey(System.getenv("TWITTER_CONSUMER_KEY")) // API KEY .setOAuthConsumerSecret(System.getenv("TWITTER_CONSUMER_SECRET")) // API KEY SECRET .setOAuthAccessToken(System.getenv("TWITTER_ACCESS_TOKEN")) .setOAuthAccessTokenSecret(System.getenv("TWITTER_ACCESS_TOKEN_SECRET"))
Domain Types Our Twitter connector supports two operations, and as a result we have two domain types which both extend the TwitterRequest abstract type. We could also create two separate for each of these request if we wanted to.

UploadMediaRequest: Upload some media to share in a tweet later

StatusUpdateRequest: Update a Twitter status

And we also have their respective responses, under the TwitterResponse abstract type.

UploadMediaResponse: Response to media upload (containing the media ID)

StatusUpdateResponse: Response to tweet (containing the newly created tweet)

Connector Setup We first create an HTTP request-reply connector transport like so.

HttpConnectorTransportConfiguration transportConfiguration = HttpConnectorTransportConfiguration .create(actorSystem.settings().config(), "chained-request-reply-example"); return new HttpConnectorTransport .Builder<TwitterRequest>() .withName("TwitterHttp") .withActorSystem(actorSystem) .withEnricher(httpRequest -> { HttpHeader authorization

= HttpHeader.parse("Authorization", getAuthorizationHeader(httpRequest)); HttpRequest request = httpRequest.addHeader(authorization); return CompletableFuture.completedFuture(request); }) .withTransportConfiguration(transportConfiguration) .build(); Note the enricher here, which adds the Authorization header containing the important OAuth 1.0 user context which Twitter will authenticate. The inputs to this method are the HTTP method (always POST) and the URL (differs depending on whether this is a media upload or a status update) We then create the following RequestReplySendConnector.

var connector = new RequestReplySendConnector .Builder<TwitterRequest, TwitterRequest, TwitterResponse, TwitterResponse>("Twitter") .withConnectorTransport(connectorTransport) .withActorSystem(actorSystem) .withMessageLogger(logger()) .withSendTransportMessageConverter(request -> { var messageHeaders = new MessageHeaders(Map.of("httpUrl", getUrl(request), "httpMethod", "POST")); if (request instanceof UploadMediaRequest) { byte[] bytes = ((UploadMediaRequest) request).getData(); var entity = HttpEntities.create(ContentTypes.APPLICATION_OCTET_STREAM, bytes); var payload = createStrictFormDataFromParts(createFormDataBodyPartStrict("media", entity)).toEntity(); return new TransportMessage(messageHeaders, payload); } return new TransportMessage(messageHeaders, ""); }) .withReceiveTransportMessageConverter(transportMessage -> { if (transportMessage.getPayload().toString().contains("media_id_string")) { return fromJson(UploadMediaResponse.class).convert(transportMessage); } else { return fromJson(StatusUpdateResponse.class).convert(transportMessage); } }) .withManualStart(false) .build(); Here we are defining a MessageLogger interface. We take both sent and received messages and log. This can be replaced with a database implementation where all message interactions are stored relating to this message association. Here we are defining a function which takes the TwitterRequest that we want to send, and creating a TransportMessage out of it, which is the representation of the request over the wire. In this case we need to define a different URL based on whether this is a status update or a media upload. Since this is a request-reply connector, we also need to define the reverse operation, which converts the received response back into a model POJO that we understand. We use Jackson to determine whether this is a media update response or a status update response, and map it accordingly back to the right POJO. Call Chain The Twitter documentation for posting media (images, GIFs, videos, etc.) states that we have to upload the media first using one of the media upload methods (simple or chunked), which will return a media_id that expires in 24 hours unless if used in a tweet before then.

We then use that media_id in our subsequent status update call, which will attach the previously uploaded media to the tweet.

connector .send(ProcessingContext.unknown(), new UploadMediaRequest(kittenPicBytes())) .thenApply(twitterResponse -> (UploadMediaResponse) twitterResponse) .thenCompose(uploadMediaResponse -> { String status = String.format("I am a status update at %s! Also check out this cat

photo", LocalDateTime.now()); var statusUpdateRequest = new StatusUpdateRequest(status, List.of(uploadMediaResponse.getMediaId())); return connector.send(ProcessingContext.unknown(), statusUpdateRequest); }) .toCompletableFuture() .join(); Calling the connector for the first time to upload the photo of the kitten Casting the response into an UploadMediaResponse Calling the connector for the second time, with the mediaId from UploadMediaResponse to post a tweet containing the media as an attachment Exercises Exercise 1: Retrieve a Remote Image At the moment the sample is loading kitten.jpg from src/test/resources. Try instead to add a call to the chain to - for example - the NASA Astronomy Picture of the Day (APOD) API to dynamically retrieve an image instead. The APOD API returns a structure like this.

{ "copyright": "Giancarlo Tinè", "date": "2021-11-15", "explanation": "What happening above that volcano? Something very unusual -- a volcanic light pillar. More typically, light pillars are caused by sunlight and so appear as a bright column that extends upward above a rising or setting Sun. Alternatively, other light pillars -- some quite colorful -- have been recorded above street and house lights. This light pillar, though, was illuminated by the red light emitted by the glowing magma of an erupting volcano. The volcano is Italy's Mount Etna, and the featured image was captured with a single shot a few hours after sunset in mid-June. Freezing temperatures above the volcano's ash cloud created ice-crystals either in cirrus clouds high above the volcano -- or in condensed water vapor expelled by Mount Etna. These ice crystals -- mostly flat toward the ground but fluttering -- then reflected away light from the volcano's caldera. Explore Your Universe: Random APOD Generator", "hdurl": "https://apod.nasa.gov/apod/image/2111/EtnaLightPillar_Tine_5100.jpg", "media_type": "image", "service_version": "v1", "title": "Light Pillar over Volcanic Etna", "url": "https://apod.nasa.gov/apod/image/2111/EtnaLightPillar_Tine_960.jpg" } Try extending the current chain by adding two more calls at the beginning.

A new RequestReplySendConnector call to retrieve the link to the picture of the day

A new RequestReplySendConnector call to request the image located at url from the response (you will need to convert it into a byte array)

You will then need to change the contents of the MediaUpload to reference the byte array of the APOD image instead of the local image of the kitten.

Exercise 2: Using the chunked media upload API The sample currently uses the "simple" API for uploading images to Twitter, which is a brittle method for uploading large media as it is not resilient to network outages and cannot be resumed. Twitter offers a more resilient method of uploading media called the chunked media upload API which supports pausing and resuming uploads.

The documentation for that can be found here.

Try changing the existing implementation to use the chunked media upload API,

and perhaps upload some larger media, like a video or GIF. The steps would be.

Call to INIT to declare the start of a media upload

Multiple calls to APPEND chunks of the media as fragments, with an incrementing index

Call to FINALIZE the media upload and receive the all-important media_id

Call to update our status using the media_id from step 3

---------- Metrics How can I receive events and metrics relating to my connector?

Connectors, by default, come with Lightbend Telemetry for reporting metrics. This enables the metrics to be exposed via various backends such as Prometheus, Datadog, New Relic, and so on.

The metrics example will show how to set up metrics with Prometheus, one of the most prevalent monitoring systems.

Lightbend Telemetry was formerly known as Cinnamon. You will see references to both in these samples. This guide is also available in the separate connector-samples Git repository here. Cinnamon Agent Before we can run the example, first run the following maven goal.

mvn clean compile This is important because it will copy the cinnamon agent JAR into the target directory. This achieved by adding the following plugin configuration in the pom.xml file.

<plugin> <groupId>org.apache.maven.plugins</groupId> <artifactId>maven-dependency-plugin</artifactId> <version>3.1.2</version> <executions> <execution> <id>copy</id> <phase>compile</phase> <goals> <goal>copy</goal> </goals> <configuration> <artifactItems> <artifactItem> <groupId>com.lightbend.cinnamon</groupId> <artifactId>cinnamon-agent</artifactId> <version>${cinnamon.version}</version> <overWrite>true</overWrite> <destFileName>cinnamon-agent.jar</destFileName> </artifactItem> </artifactItems> </configuration> </execution> </executions> </plugin> After running the maven compile goal, there should be a dependency/cinnamon-agent.jar file inside the target directory. We can now add the following VM argument when running the test.

-javaagent:target/dependency/cinnamon-agent.jar If using IntelliJ IDEA, it might be necessary to disable the IntelliJ Java agent, as it can cause issues with the Cinnamon agent. The agent can be disabled by going selecting Help > Edit Custom Properties.. and then adding idea.no.launcher=true to the properties

file. The Lightbend Telemetry (Cinnamon) Agent can be acquired as part of a Lightbend subscription. More info on setting that up can be found here.

This test sends a message via a SendConnector using a no-op ConnectorTransport for an hour.

Verify Metrics If the VM argument above is set correctly the metrics should be published to a locally running prometheus server which is exposes on port 9001.

Navigate to localhost::9001.

There should be a page full of text, displaying various metrics.

Search for "application_ipf_requests_sent"

The search should show the following result. This metric shows the total count of messages sent via connectors.

application_ipf_requests_sent{application="com.intellij.rt.junit.JUnitStarter",host="somehostname",connector="MetricsSendConnector",type="send",} 125.0 This statistic (and others) can be scraped by Prometheus and fed into a graphing library, such as Grafana to produce interesting visualisations of the data. Prebuilt Grafana graphs by Lightbend can be found here.

---------- Filter Messages How do I filter messages so they are not passed to my receive handler?

This Connector example shows how you can filter messages based on their header values.

Defining a Filter Simply define a filter by defining Criteria as such:

var propNameToFilterOn = "myprop";

new ReceiveConnector .Builder<String>("OurReceiver") .withActorSystem(actorSystem) .withConnectorTransport(receivingTransport) .withMessageLogger(logger()) .withFilterCriteria(new MessageHeaderCriteria(propNameToFilterOn, "1")) .withReceiveTransportMessageConverter(a -> a.getPayload().toString()) .withManualStart(false) .withProcessingContextExtractor(a -> ProcessingContext.unknown()) .withSkipCorrelationOn(a -> true) .withReceiveHandler((receivingContext, response) -> CompletableFuture.supplyAsync(() -> list.add(response)) .thenApply(a -> null)) .build(); Defining the filter In the above example, we filter out (i.e. do not accept) any messages that have a property name myprop having the value 1.

Note that it is possible to write custom Criteria and also use logical AND and OR operators to combine operators. See Filtering for more information on this.

---------- PascalCase Messages How do I accept messages that are incoming in PascalCase format?

Defining a Custom Jackson ObjectMapper IPF currently has a utility class called the SerializationHelper which is a wrapper around a Jackson ObjectMapper that registers custom serialisers/deserialisers. This allows us to have a consistent approach when serialisers/deserialisers messages across the IPF estate.

By default Jackson support CamelCase and the SerializationHelper therefore also only support CamelCase messages by default. This can be easily changed to support PascalCase.

To support PascalCase you need to define a new instance of the ObjectMapper with the following property:

private final ObjectMapper pascalCaseMapper = SerializationHelper.objectMapper()
.setPropertyNamingStrategy(PropertyNamingStrategies.UPPER_CAMEL_CASE);
This can then be used when mapping messaged to/from external system within the connector code.

Below is an example ReceiveConnector and SendConnector that would receive or send messages in PascalCase respectively:

package com.iconsolutions.connector.samples.pascal;

import akka.actor.ActorSystem; import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper; import com.fasterxml.jackson.databind.PropertyNamin
import com.iconsolutions.common.exception.IconRuntimeException; import
com.iconsolutions.ipf.core.connector.ReceiveConnector; import com.iconsolutions.ipf.core.connector.ReceiveCo
import com.iconsolutions.ipf.core.connector.SendConnector; import com.iconsolutions.ipf.core.connector.SendC
import com.iconsolutions.ipf.core.connector.message.MessageHeaders; im-
port com.iconsolutions.ipf.core.connector.message.TransportMessage; import
com.iconsolutions.ipf.core.connector.transport.ConnectorTransport; import
com.iconsolutions.ipf.core.connector.transport.ReceiveConnectorTransport;
import com.iconsolutions.ipf.core.shared.correlation.CorrelationId; import
com.iconsolutions.ipf.core.shared.domain.context.ProcessingContext; im-
port com.iconsolutions.ipf.core.shared.domain.context.UnitOfWorkId; import
com.iconsolutions.ipf.payments.domain.clearing_and_settlement.pacs002.FIToFIPaymentStatusReport;
import com.iconsolutions.ipf.payments.domain.clearing_and_settlement.pacs008.FIToFICustomerCreditTrans
import com.iconsolutions.ipf.payments.domain.payment_initiation.pain001.CustomerCreditTransferInitiation;
import com.iconsolutions.samplesystems.shared.model.header.CryptoHelper;
import com.iconsolutions.simulator.api.RequestHandler; import lombok.extern.slf4j.Slf4j;
import org.springframework.context.annotation.Bean;

@Slf4j public class PascalConnectors {

/** * Here set the UPPER_CASE_CAMEL (PascalCase) naming strategy * on your ObjectMapper instance */ private final ObjectMapper pascalCaseMapper = SerializationHelper.objectMapper() .setPropertyNamingStrategy(PropertyNamingStrategies.UPPER_CAMEL_CASE);

@Bean public ReceiveConnector<CustomerCreditTransferInitiation> executePaymentReceiveConnector(ReceiveConnectorTransport initPaymentReceiveConnectorTransport, RequestHandler someRequestHandler, ActorSystem actorSystem) { return ReceiveConnectorBuilderHelper.<CustomerCreditTransferInitiation>builder("InitPaym
"initpayment.receive-connector", actorSystem) .withMessageLogger(m -> log.debug("Receive connector has identified message: {}", m.getMessage())) .withProcessingContextExtractor(tm -> ProcessingContext.builder() .unitOfWorkId(UnitOfWorkId.createRandom()) .build()) .withConnectorTransport(initPaymentReceiveConnectorTransport) .withReceiveTransportMessageConverter(message -> this.convertResponse(message.getPayload().toString())) .withReceiveHandler((context, payload) -> someRequestHandler.process(payload)) .build(); }

/** * Here you would convert the incoming PAIN001 JSON message in Pascal format into your PAIN001 domain object */ public CustomerCreditTransferInitiation convertResponse(String messageText) { try { return pascalCaseMapper.readValue(messageText, CustomerCreditTransferInitiation.class); } catch (JsonProcessingException e) { throw new IconRuntimeException(e); } }

@Bean public SendConnector<FIToFIPaymentStatusReport, FIToFIPaymentStatusReport> executePaymentSendConnector(ConnectorTransport<FIToFIPaymentStatusReport> executePaymentReceiveConnectorTransport, ActorSystem actorSystem) { return SendConnectorBuilderHelper .<FIToFIPaymentStatusReport, FIToFIPaymentStatusReport>builder("ExecutePaymentSend", "executepayment.send-connector", actorSystem) .withMessageLogger(m -> log.debug("Send connector has identified message: {}", m.getMessage())) .withCorrelationIdExtractor(it -> CorrelationId.of(it.getTxInfAndSts().get(0).getOrgnlTxId())) .withConnectorTransport(executePaymentReceiveConnectorTransport) .withSendTransportMessageConverter(this::convertToTransport) .build(); }

/** * Here you would convert the PACS002 domain object into Pascal formatted JSON before sending out */ public TransportMessage convertToTransport(FIToFIPaymentStatusReport response) { try { return new TransportMessage(new MessageHeaders(CryptoHelper.messageHeaders()), pascalCaseMapper.writeValueAsString(response)); } catch (JsonProcessingException e) { throw new IllegalStateException(e); } } } ----------
---

---------- Application Builder This section covers the core components or modules which exist to get you started building IPF applications.

Flo Starter - this consists of all the necessary core dependencies required to wire together an IPF flow implementation.

Archetype Starter - this is the quickest way to bootstrap a new project using the IPF SDK. The Maven archetype will generate a Maven project (using the Icon BOM as its parent) containing an application skeleton, the flow domain and a test module.

IPF Flo Starter

IPF Flo Starter bootstraps an IPF flow implementation by providing an Akka ActorSystem and the Spring Boot autoconfiguration to wire all the necessary core dependencies that are required for running a command (write) and query (read) side.

Those core dependencies are often used features documented in the Features section and you will find common implementation activities in the Getting Started.

Concepts

Flo Starter Projects

Understanding Passivation, Remembering Entities, and Schedulers

Domain Operations

Features

Application Configuration

Transaction Caching

Application Health Indicators

Flow Scheduling

Automated Retries

Monitoring and Observability

Getting Started

Create a new IPF Application Project

Defining The Read Side

How to guides…

How to handle schema evolution in IPF events

How to troubleshoot error messages

How to secure IPF HTTP Endpoints

How to use the persistent scheduler with a flow

How to implement a duplicate check function

How to get aggregate data for use in an external domain request ----------
---

---------- Concepts Discover the key concepts that IPF Flo Starter provides

Flo Starter Projects

Understanding Passivation, Remembering Entities, and Schedulers

Domain Operations ----------
---------- Flo Starter Projects Modules ipf-common-starter This is the base module that provides these foundational capabilities. It is used by the other starter projects.

Akka actor system and other sub-systems (e.g. cluster sharding, persistence)

HOCON and Spring configuration initialiser. See IPF Application Configuration. Spring info contributors to display application properties on /actuator/info

ipf-write-starter Provides write capabilities for persisting events to the selected persistence type. Additionally, this provides:

BehaviourRetriesSupport for sending commands to Akka Event Sourced Behaviour in a reliable way

Metrics recorders that provide metrics for Prometheus

TransactionCacheService for detecting functional and technical duplicate checking. See Transaction Caching

ipf-journal-processor-starter Provides capabilities that continuously reads persisted events and then delegates the processing of those events. See ipf-read-starter for implementations of EventProcessor.

This currently supports processing events from three persistence types:

Mongo

Cassandra

JDBC

Error Handling In the case the EventProcessor implementation fails - Resiliency Settings are in place to handle recovery of messages from the event stream. Optionally a dead letter strategy can be implemented in the case of retry failures.

Deadletter Appender The DeadletterAppender is a functional interface which is called whenever a message fails during processing of the event stream after all retries have been exhausted.

@FunctionalInterface public interface DeadletterAppender { /** * Appends the {@link EventProcessorException} to the deadletter queue. * * @param eventProcessorException contains the failed {@link EventEnvelope message}, {@link

PersistenceId id} and the cause of failure * @return a {@link CompletableFuture} */ CompletableFuture<Void> append(EventProcessorException eventProcessorException); } Providing a DeadletterAppender implementation is optional and if one is not provided the journal processor will use the default implementation. The default behaviour is to simply log both the failed message and the exception that caused the error.

All failed messages will be provided as an exception that extends EventProcessorException. EventProcessorException wraps the original exception as the cause alongside the received event.

ipf-read-starter Provides capabilities to process the events that were read from ipf-journal-processor-starter to construct a domain aggregate.

See ReadSideEventProcessor as an example of an EventProcessor implementation.

A static /index.html is provided out of the box for a simple view of the read aggregates that have been successfully processed.

Default Configuration ipf-common-starter ipf.conf # Default IPF configuration to allow bootstrapped execution of applications that depend on ipf-common-starter and provide # core functionalities such metrics, health, ports and cluster set-up.

Name of the actor system that is required by Akka. There will only be a single actor system per application actor-system-name = ipf-flow

The name to be used as a unique identifier of the source of IPF system events ipf.system-events.source = 0.0.0.0 ipf.system-events.source = ${?akka.remote.artery.canonical.hostname}

ipf.application.name = ${actor-system-name} spring.application.name = ${ipf.application.name}

The duration after which the connector event processor will # check connector health and perform the startup logic connector.event-processor.keep-alive-interval = 5s

Default cinnamon to allow for monitoring cinnamon { prometheus.exporters += "http-server" }

Exposing Spring management endpoints for further metrics management { endpoint.metrics.enabled = true endpoints.web.exposure.include = "health,info,metrics,prometheus" endpoint.health.probes.enabled=true health.livenessState.enabled=true health.readinessState.enabled=true endpoint.prometheus.enabled = true endpoint.health.show-details = always metrics.export.prometheus.enabled = true } ipf-journal-processor-starter ipf.conf # Default IPF journal processor configuration

Identifies that this node is on the read side when the cluster endpoint is invoked. It's added additionally with += so nodes can have more roles.

akka.cluster.roles += read-model

event-processor { # The ID used to create the key for the Entity Type of events. # The KeepAlive cluster singleton is also created with the name keepAlive-[id] id = EventProcessor

The interval at which the KeepAlive actor probes itself with a Probe.INSTANCE keep-alive-interval = 2 seconds

The tag prefix for events generated by the write side. # This value must match the ‘ipf.behaviour.event-processor.tag-prefix‘ setting # used by the write side or else the processor will not be consuming all events. tag-prefix = ["tag"]

Ensure configurations match if journal processor is running on the write side tag-prefix = ${?ipf.behaviour.event-processor.tag-prefix}

The number of partitions configured for our event journal. # Each partition will be processed in parallel by a dedicated Akka stream. # This value must match the ‘ipf.behaviour.event-processor.parallelism‘ setting # used by the write side - in case the value is lower than ‘parallelism‘ the processor # will not be consuming all events, and if the value is higher, the processor will # not balance the work between the nodes equally. number-of-partitions = 4

Maintaining backward compatibility with previous configuration setting. number-of-partitions = ${?event-processor.parallelism}

Ensure configurations match if journal processor is running on the write side number-of-partitions = ${?ipf.behaviour.event-processor.parallelism}

The number of events to demand per partition from upstream, and process in parallel upstream-event-demand = 1

restart-settings { # The starting backoff interval to use when restarting event processor streams. min-backoff = 500 millis

The starting backoff interval to use when restarting event processor streams. max-backoff = 20 seconds

Maintaining backward compatibility with previous configuration. max-backoff = ${?event-processor.backoff}

The amount of restarts is capped within a timeframe of max-restarts-within. max-restarts = 86400000

The amount of restarts is capped to max-restarts within a timeframe of within. max-restarts-within = 1 days

The starting backoff interval to use when restarting event processor streams. jitter = 0.1 }

To improve throughput, offsets of successfully processed events are # not checkpointed for each event but are grouped together in # size and time based windows and the last event offset in a window # is used as a checkpoint.

The window is considered complete when either it is filled by ‘size‘ # offsets or the ‘timeout‘ interval has elapsed. commit-offset-window { # The size of the window. size = 1000

The amount of time to wait for ‘size‘ events to complete. timeout = 1 minute } }

Enables bean creation in *ReadSideConfig Spring config. The flow domain module contains configuration for both the read and write sides. This allows the selective initiation of *ReadSideConfig or *WriteSideConfig based on which Maven modules are included. ipf.read-side.enabled = true ipf-write-starter ipf.conf ipf { behaviour { retries { initial-timeout = 100ms backoff-factor = 2 max-attempts = 3 } metrics { behaviour-configurations = [{ behaviour = com.iconsolutions.ipf.core.platform.write.test.TestApp.Beh1 enabled = true, csm-related-states = [Csm1, Csm2] }, { behaviour = com.iconsolutions.ipf.core.platform.write.test.TestApp.Beh2 enabled = true event-whitelist = [com.iconsolutions.ipf.core.platform.write.test.TestApp.Evt1, com.iconsolutions.ipf.core.platform.write.test.TestApp.Evt2] event-blacklist = [com.iconsolutions.ipf.core.platform.write.test.TestApp.Evt1] }, { behaviour = com.iconsolutions.ipf.core.platform.write.test.TestApp.Beh3 enabled = false }] } } }

---------- Understanding Passivation, Remembering Entities, and Schedulers When IPF flows are inflight (i.e. have not yet reached a terminal state), IPF's default behaviour is to keep them in memory while they are in progress until they are terminated. Flows are distributed more-or-less evenly across the cluster with Akka Cluster Sharding.

This article explains the concepts of passivation, remembering entities, and how to restart failed flows.

Remembering Entities If a node dies or is gracefully stopped (or if all nodes are stopped and the application is cold-started), then all in-flight flows (shards) are loaded back into memory on the next startup. This is called remembering entities.

The way Akka determines whether a flow is or is not completed is by also storing this state in a store of some description. The options for this are eventsourced (the IPF default) or ddata. The eventsourced option is the IPF default because it is more performant and is resilient to total restarts of the cluster with no additional configuration.

Any data stored in ddata is saved to disk by default, so cannot survive a total cluster restart in a containerised environment like Kubernetes. On such environments you will have to configure a Persistent Volume to store this state, and change the IPF configuration to point the application at the persistent volume.

More information on the different remember entities stores (and how to configure them) here.

Passivation and Automatic Passivation By default, a flow is considered "stopped" when the flow determines that it has reached a terminal state and stops itself. This is called passivation.

Other ways to stop a flow - even if it isn't finished - are:

Manually passivating using domain operations

Auto-passivate after entities reach a certain age. This is called automatic passivation.

If remembering entities is enabled, automatic passivation cannot also be enabled Action Revival When a flow is loaded back into memory with remembering entites after another node leaving (or crashing), the IPF flow is notified of this and reacts by sending itself an ActionRecoveryCommand. This special command attempts to "poke" the flow to perform the next action that it needs to perform based on the last known state. More information on the topic is available here.

Default IPF behaviour The default IPF behaviour is:

Remember entities with eventsourced as the store

Automatic Passivation off (must be off if remember entities is on)

Akka Scheduler

This default behaviour is suitable for flows that match the following profile:

Short-lived

Any sort of volume (high or low)

However this approach can become problematic when there are lots of long-running flows: if there is a step that waits for hours or days, then there might be a large number of flows that are parked and consuming memory when they will not be needed for a long time.

If this is the case, then it might be worth changing how the cluster behaves by:

Switching off remembering entities

Enabling automatic passivation

Using IPF's persistent Quartz Scheduler to schedule retries

If they were to be laid out in a table, the options would be:

ID Remember entities Auto-passivation Scheduler implementation Good for How node failures handled? 1

Enabled

Disabled

Akka (not persistent)

High- or low-volume flows that complete quickly (e.g. within a minute)

Flows are always held in memory and ActionRevival process (see above) is triggered when entities are restarted on another node after a failure or shutdown

2

Disabled

Enabled

Quartz (persistent)

Long-running flows

Retrying of tasks is managed separately to the flow itself and is also resilient against failures

Triggering for missed events in the past If using combination 2 (remember entities disabled/auto passivation enabled/Quartz) then the Quartz Scheduler implementation will manage the retries for any tasks relating to flows. This is resilient because it too uses a Cluster Singleton to manage this work and in the event of the failure of the node that is holding the Cluster Singleton, the singleton is migrated onto the next oldest node and the jobs are re-queued for retrying.

This process should only take a few seconds and so for long-running processes this is generally not a big problem. However in the unlikely case that a job has needed to run while the Cluster Singleton was in the process of being migrated, it is currently skipped with a warning in the logs alerting the user to this issue (i.e. a trigger was missed while the Cluster Singleton was being moved). This functionality can be toggleable in the future if so desired.

Database usage concerns On startup and with remember entities enabled, IPF will read in all in-flight entities by reading in all of their events to build the current state of that particular flow. This is a read-heavy operation, and for environments where database usage is metered in such a way (e.g. Azure Cosmos DB) then it might be a better idea to use Combination 2 to avoid this heavy startup cost.

Configuration To configure the non-default Combination 2, set the following configuration keys in ipf-impl.conf or application.conf:

Disable remembering entities akka.cluster.sharding.remember-entities = off # Set the passivation strategy to be the default i.e. a maximum of active entities at any one time akka.cluster.sharding.passivation.strategy = default-strategy

Change the below if you want to use a separate MongoDB instance for persisting scheduled tasks ipf-scheduler { mongodb = { uri = ${ipf.mongodb.uri}

database-name = ”scheduled” } }

---------- Domain Operations This page lists the various operations that are available on the global domain level as static method calls.

passivate: Unload a flow from memory This will stop a flow instance (even if it’s not reached a terminal state) to save some memory, if it is known that it has reached a long-running step, for example.

Note that messages directed at this instance will still be delivered, but IPF will reload the events of that flow from the journal instead of having them already stored in memory.

The call is:

XxxDomain.passivate(”MyBehaviour|some-id-23149082”); Where Xxx is the name of the IPF solution containing the relevant flow, and MyBehaviour is the name of the flow itself.

getAggregate: Get a flow’s aggregate (state) To retrieve the current state of a flow instance, you can call:

XxxDomain.getAggregate(”MyBehaviour|some-id-23149082”); This will return a CompletionStage<Aggregate> which completes when the aggregate is returned.

The Aggregate will be of the type MyBehaviourAggregate which will contain:

The current status

The last failure reason code(s)/description(s)

All business data

All events

getStatus: Get a flow’s status This operation is similar to that above, but will only return the current status. Also returns as a CompletionStage<AggregateStatus>.

To retrieve the status:

XxxDomain.getStatus(”MyBehaviour|some-id-23149082”); abort: Stop a flow’s execution This command takes a reason argument and:

Sets the flow’s status to ABORTED

Sets the resulting status to Aborted

Publishes an AbortedEvent with the reason specified in the call

To abort a flow:

XxxDomain.abort(”MyBehaviour|some-id-23149082”, ”some special reason”); Once a flow has been aborted, it cannot be resumed, even with the resume function below. resume: Continue a flow’s execution This operation will invoke the action revival process for a flow to attempt to move a transaction that appears to be stuck onto the next state.

Note that this operation cannot resume aborted transactions.

To resume a flow:

XxxDomain.resume(”MyBehaviour|some-id-23149082”);

---------- Features Foundational IPF capabilities often used in IPF application and solution builds

Application Configuration

Transaction Caching

Application Health Indicators

Flow Scheduling

Automated Retries

Monitoring and Observability

---------- Application Configuration A convention has been introduced for an IPF application start-up configuration. The aim was to provide a consistent and predictable approach to configuration, yet flexible enough for integration tests that utilise various Spring annotations to inject configuration dynamically.

Config Hierarchy When preparing the application for deployment, the configuration hierarchy will be observed as long as these rules are followed:

Only HOCON (.conf files) are used for application configuration

Developers must populate the appropriate configuration file following this descending hierarchy of precedence:

application-override.conf Where? Deployment config (i.e. k8s configmap) or on file-system mounted on [app-name]/conf to the container

When to use? This is to be used in an emergency that cannot be fulfilled by the existing hierarchy

application.conf Where? Deployment config (i.e. k8s configmap) or on file-system mounted on [app-name]/conf to the container

When to use? Should contain environment specific configuration such as URLs or security config. If we had to deploy the same application to different environments, the differences in the application.conf should mostly consist of environment specific configuration.

ipf-impl.conf Where? src/main/resources of IPF based applications

When to use? As defaults for standing the application for local execution. Can also provide additional placeholders as overrides to be set by application.conf for common configuration across the modules. For example, ipf-impl.conf in sample-client provides overrides via an include file called endpoint-overrides.conf that override encryption config for every known endpoint.

ipf.conf Where? src/main/resources of modules used by IPF based application

When to use? Defining sensible defaults for applications that will use this module. Modules should not set the same configuration since they are at the same level. The problem is that it will not be predictable as to which config will "win". The exception is if += is used on lists (e.g. akka.cluster.roles). Placeholders should be avoided unless necessary or recommended by the underlying library. For example, Alpakka Kafka recommend using config inheritance to assign defaults for Kafka configuration.

This can be seen as IPFs version of Akka's reference.conf for the various modules.

Further Overriding If configuration via the file hierarchy is not enough, further overriding is possible with environment variables and JVM properties.

Environment Variables By setting the JVM property -Dconfig.override_with_env_vars=true (via IPF_JAVA_ARGS on IPF containers) it is possible to override any configuration value using environment variables even if an explicit substitution is not specified.

The environment variable value will override any pre-existing value and also any value provided as Java property.

With this option enabled only environment variables starting with CONFIG_FORCE_ are considered, and the name is mangled as follows:

the prefix CONFIG_FORCE_ is stripped

single underscore(_) is converted into a dot(.)

double underscore(___) is converted into a dash(-)

triple underscore(_) is converted into a single underscore(_)

i.e. The environment variable CONFIG_FORCE_a_bc_d set the configuration key a.b-c_d

This is only supported from com.typesafe:config:1.4.0 onwards. If this does not work, it's likely because your application is pulling in an older version as a

transitive dependency. System Properties Any configuration value can be overridden as a Java -D property (again via IPF_JAVA_ARGS on IPF containers). These will override configuration properties set in the file hierarchy. However, environment variables set with CONFIG_FORCE_ still take precedence.

Lightbend HOCON + Config The process is powered by the combination of HOCON and the Lightbend Config library. While the Config library also supports YAML and .properties files, adopting only HOCON provides consistency and flexibility as it has additional capabilities over the other two file formats.

Noteworthy are the block inheritance and substitution functionalities but also that Akka serves as the foundation to IPF. Many standard configuration options would be difficult or awkward to do such as defining seed node lists.

The readability of HOCON makes it suitable as documentation as well as default configuration. In this way much effort is saved from writing additional support documentation when the default configuration can be commented on. There is also added impetus for developers to keep configuration tidy as it can be used as customer facing collateral.

Implementation Detail The config hierarchy is used as fallback to Spring PropertySource, loaded before the application context is initialised. This is purposely to maintain the usage of the various Spring annotations that can be used to dynamically inject properties into integration tests. As we avoid using any Spring configuration mechanisms (e.g. application.* or Spring profiles) HOCON and the Config hierarchy should become the only source of configuration in actual deployments.

This means that it is still possible to use Spring config when deployed. However, this is strongly discouraged to maintain consistency in configuring our expanding landscape of components.

/** * Combine the two configurations in Spring through the environment and HOCON through config. * <p> * .defaultOverrides() is required so when -Dconfig.override_with_env_vars=true you can set environment variables * with CONFIG_FORCE_ to override any configuration. * * @param applicationContext Combined configuration from Spring and HOCON */ @Override public void initialize(ConfigurableApplicationContext applicationContext) { ConfigurableEnvironment env = applicationContext.getEnvironment();

// Skip the Spring Cloud context and set the configuration of the main application context instead if (!"bootstrap".equals(env.getProperty("spring.config.name")) && shouldInitialiseConfig(applicationContext)) { env.getPropertySources().remove(COMBINED_CONFIG);

Config springConfig = new ConfigBuilder(env).buildHoconConfigFromEnvironment(); Config hoconConfig = parseWithFallback(getConfigFileHierarchy());

Config allConfig = ConfigFactory.defaultOverrides() .withFallback(springConfig) .withFallback(hoconConfig);

var additionalConfig = getAdditionalConfig(allConfig); if (additionalConfig.isPresent()) { allConfig = allConfig.withFallback(additionalConfig.get()); }

allConfig = allConfig.resolve();

PropertySource<?> allProperties = ConfigPropertySourceConverter.createPropertySource(COMBINED_CON allConfig);

env.getPropertySources().addFirst(allProperties);

Object singleton = applicationContext.getBeanFactory().getSingleton(DEFAULT_CONFIG); if(singleton != null) { ((DefaultListableBeanFactory)applicationContext.getBeanFactory()).destroySingleton(D } applicationContext.getBeanFactory().registerSingleton(DEFAULT_CONFIG, allConfig); } } Masking Sensitive Data in Logs Masking is done with our MaskPatternLayout class, which extends logback's PatternLayout. Basically, we have replaced the default layouts with the new one in the DefaultLogbackConfigInitializer, which gets triggered after HoconConfigInitialiser. Hocon configuration needs to be initialized first since masking configuration exists there.

Masking configuration contains two parts:

masking.enabled - boolean value that says if the feature itself is enabled or not

masking.mask-config-objects - that represents an array of objects. Each object contains reg-ex and strategy information for the value we want to mask

We have introduced the following masking strategies:

MASK_ALL - mask all characters

MASK_FIRST_N_CHARS - mask first n characters

MASK_LAST_N_CHARS - mask last n characters

MASK_M_TO_N_CHARS - mask chars from position m to n

NO_MASK_FIRST_N_CHARS - do not mask first n characters

NO_MASK_LAST_N_CHARS - do not mask last n characters

NO_MASK_M_TO_N_CHARS - do not mask chars from position m to n

Here's an example how masking configuration looks like:

masking { enabled = true mask-config-objects = [//objects matching json fields { pattern = "\"Nm\"\\s*:\\s*\"([^\"]+)\"", strategy { name = "MASK_ALL", args = {} } }, { pattern = "\"Id\"\\s*:\\s*\"([^\"]+)\"", strategy { name = "MASK_FIRST_N_CHARS", args = { n = 3 } } }, { pattern = "\"Dept\"\\s*:\\s*\"([^\"]+)\"", strategy { name = "NO_MASK_M_TO_N_CHARS", args = { m = 3, n = 5 } } }] } So as mentioned above, there is a regex for each value we want to mask. This will

catch the whole line with the key/tag and the value, but only the group that represents the value is matched for masking.

Here are some examples of masking xml tags, showing how each strategy should work:

MASK_ALL -> <Dept>****</Dept>, MASK_FIRST_N_CHARS(3) -> <SubDept>***Dept</SubDept>, MASK_LAST_N_CHARS(2) -> <StrtNm>Strt**</StrtNm>, NO_MASK_FIRST_N_CHARS(4) -> <BldgNb>Bldg**</BldgNb>, NO_MASK_LAST_N_CHARS(2) -> <BldgNm>****Nm</BldgNm>. MASK_M_TO_N_CHARS(4,7) -> <TwnLctnNm>Twn****Nm</TwnLctnNm>, NO_MASK_M_TO_N_CHARS(5,7) -> <CtrySubDvsn>****Sub****</CtrySubDvsn> Already defined masking objects Important thing to mention is, because this feature's implementation depends on logback library, in order for it to work, there can be used no other logger implementation.

---------- Transaction Caching IPF is an event-sourced application and as a result the read ("query") side is eventually consistent. This means that for cases where we need to look into the application's very recent history, it might not be representative to look into the read side, as it may not yet have caught up with all events that have taken place on the write side.

As a result, users have the ability to implement a transaction cache, using the supplied transaction cache service. It can be used to satisfy business requirements such as:

Functional duplicate checks

Technical duplicate checks

Populating the Transaction Cache The transaction cache requires some assembly, but it's not that complicated to get started! Here's what you need to do:

1. Identify the transaction types to persist You must identify what you wish to persist into the cache. The PersistentTransactionCacheService has a generic type T which you can use to insert any sort of MongoDB persist-able POJO. In our example we want to persist this payment object:

public static class Payment { private final String from; private final String to; private final BigDecimal amount; private final LocalDate paymentDate; } 2. Select business data that's important We need to implement an "ID extractor" to determine which fields are important when determining if we've seen this transaction before. Some examples might be:

Amount

End-to-end ID

From Identifier

To Identifier

Unstructured data

This is implemented as a Function<T, List<String».

This gives the service a way to extract the relevant fields and hash them together to efficiently attempt to look them up later.

Here's an example of how to initialise the transaction cache service:

var transactionCacheService = new PersistentTransactionCacheService<>(payment -> List.of(payment.getFrom(), payment.getTo(), payment.getAmount().toString()) , repo, repositoryRetryProvider); The first argument consists of the list of fields (which must be protected against nulls!) which form part of the hash. The particular implementation of cache service we have selected here is MongoDB-based, so the second argument takes a repository for storing cache entries. Different implementations may have different signatures depending on their requirements.

1. Create an entry type enum The cache can potentially contain different types of transactions. For that reason we need to be able to enumerate the different types. This is represented by the TransactionCacheEntryType interface. Here's its definition:

public interface TransactionCacheEntryType { String getName(); } We can see that it's really just a way to be able to differentiate between different types of transactions being cached. We need this because some transaction flows can share the same root message type (think incoming and outgoing messages of the same type e.g. pacs.008).

Here's an example implementation of a TransactionCacheEntryType:

public enum ExampleTransactionCacheEntryType implements TransactionCacheEntryType { TYPE_ONE, TYPE_TWO;

@Override public String getName() { return name(); } } This is an enum which implements the TransactionCacheEntryType interface and can support two different types of cache entries: TYPE_ONE and TYPE_TWO.

We can then use the service to persist our types to persist.

3. Wrap and save We can now call the transaction cache service to save our Payment with its type like this:

var payment = new Payment("Me", "You", new BigDecimal("4.20"), LocalDate.now());

transactionCacheService.saveToCache(TYPE_ONE, payment) .toCompletableFuture() .join(); In case we want to prevent saving two entries for the same physical message (e.g. in case of a retry), we should call saveToCache method

with messageId parameter. MessageId should be a unique identifier for a content we are storing in cache. New entry will not be saved if entry with the same hash and messageId already exists in the cache for the given type, and it will return existing one instead.

var payment = new Payment(”Me”, ”You”, new BigDecimal(”4.20”), LocalDate.now());

transactionCacheService.saveToCache(TYPE_ONE, payment, ”messageId”) .toCompletableFuture() .join(); We save it with the transactionCacheService. The join() call is for testing; we should not block while this operation completes.

Checking the Transaction Cache The cache service has the following method for retrieving data from the cache:

CompletionStage<List<TransactionCacheEntry<T»> findInCache(TransactionCacheEntryType type, T content); It needs the type of entry you wish to find, followed by the T type you wish to check to see if it’s a functional duplicate.

It returns a future containing a list of matching cache entries. You may wish to inspect their creationDate to check for functional duplicates within some window of time.

Implementation Considerations Housekeeping For the MongoDB implementation, consider using a MongoDB TTL index on the creationDate field to expire entries. An index for searching by hash is created by default, but you may wish to add this TTL index to expire (delete) entries after a specific period of time if they are no longer required.

---------- Application Health Indicators IPF flow starter provides a spring backed health actuator, this is responsible for updating the health endpoint of the application.

The health of the app is built up from existing spring health indicators, i.e. mongo, diskspace, etc, plus the health of the connector transports.

Connector Transports The IPF connectors emit one of TransportAvailable, TransportUnavailable, TopicUnavailable or CircuitBreakerStateChanged which are then converted into a health state by the ConnectorEventProcessor.

Health Reports Reports the general health of the app by aggregating all health indicators.

localhost:8081/actuator/health

{ ”status”:”UP”, ”components”:{ ”akkaClusterMembership”:{ ”status”:”UP” }, ”connectors”:{ ”status”:”UP”, ”details”:{ ”FraudSendKAFKA”:{ ”connectorName”:”Fraud”, ”status”:{ ”status”:”UP” }, ”circuitBreakerState”:”CLOSED”,

”circuitBreakerClosed”:true }, ”KafkaReceiveConnectorTransport”:{ ”connectorName”:”ExecutePaymentRequestReceive”, ”status”:{ ”status”:”UP” }, ”circuitBreakerState”:”CLOSED”, ”circuitBreakerClosed”:true }, ”KafkaSendConnectorTransport”:{ ”connectorName”:”ExecutePaymentSend”, ”status”:{ ”status”:”UP” }, ”circuitBreakerState”:”CLOSED”, ”circuitBreakerClosed”:true }, ”FraudReceiveConnectorTransport”:{ ”connectorName”:”FraudReceive”, ”status”:{ ”status”:”UP” }, ”circuitBreakerState”:”CLOSED”, ”circuitBreakerClosed”:true } } }, ”diskSpace”:{ ”status”:”UP”, ”details”:{ ”total”:368110161920, ”free”:78524358656, ”threshold”:10485760, ”exists”:true } }, ”livenessState”:{ ”status”:”UP” }, ”mongo”:{ ”status”:”UP”, ”details”:{ ”version”:”4.4.4” } }, ”ping”:{ ”status”:”UP” }, ”readinessState”:{ ”status”:”UP” } }, ”groups”:[”liveness”, ”readiness”] } If we don’t enable health-check it will raise TransportAvailable by default.

Application Configuration To enable spring endpoints add the following enablers to your application config.

management.endpoint.health.probes.enabled=true management.health.livenessState.enabled=true management.health.readinessState.enabled=true Liveness Reports whether the app is alive.

localhost:8081/actuator/health/liveness

{ ”status”: ”UP” } Readiness Reports whether application is ready to process data.

This relies on the transports being in the spring context otherwise the readiness of those transports is not checked.

localhost:8081/actuator/health/readiness

{ ”status”: ”OUT_OF_SERVICE” }

---------- Flow Scheduling The ipf-flo-scheduler module provides the ability to interact with IPF flows and activate the scheduling capabilities. There are three types of scheduling features currently available within a flow:

Action Timeouts: provides the ability for the flow to continue processing if a response from an action is not provided in a defined timeframe.

Retries: this provides a mechanism for the flow to be able to invoke retries of actions when responses are not returned in defined timeframe.

Processing Timeouts: this provides a mechanism to define that a section of flow (between any two states) has to be completed within a defined timeframe.

There are then two different types of scheduler to provide the implementations of these depending on requirements:

Akka scheduler

Persistent scheduler

To use a scheduler, it is simply necessary to import the relevant scheduler as defined below and then inject the scheduler port when instantiating the domain as such:

@Bean public QuickStartModelDomain initialiseDomain(ActorSystem actorSystem, SchedulerPort schedulerAdapter) { // All adapters should be added to the domain model return new QuickStartModelDomain.Builder(actorSystem) .withSchedulerAdapter(schedulerAdapter) .build(); } The relevant scheduler implementation will then provide the required implementation.

Akka Scheduler All documentation on akka scheduling can be found here.

The scheduler in Akka is designed for high-throughput of thousands up to millions of triggers. The prime use-case being triggering Actor receive timeouts, Future timeouts, circuit breakers and other time dependent events which happen all-the-time and in many instances at the same time.

The Akka scheduler is not designed for long-term scheduling and for that you should use Quartz scheduler. Nor is it to be used for highly precise firing of the events. The maximum amount of time into the future you can schedule an event to trigger is around 8 months, which in practice is too much to be useful since this would assume the system never went down during that period. If you need long-term scheduling we highly recommend looking into alternative schedulers, as this is not the use-case the Akka scheduler is implemented for.

To use the akka scheduler implementation, all you need to do is provide the dependency:

<dependency> <groupId>com.iconsolutions.ipf.core.platform</groupId> <artifactId>ipf-flo-scheduler-akka</artifactId> </dependency> Persistent Scheduler The persistent scheduler implementation uses Icon's ipf-persistent-scheduler as it's underlying scheduler implementation.

All documentation on Quartz scheduling can be found here.

Quartz is a richly featured, open source job scheduling library that can be integrated within virtually any Java application.

If your application has tasks that need to occur at given moments in time, or if your system has recurring maintenance jobs then Quartz may be your ideal solution.

In addition to Quartz scheduling, IPF also supports persisting scheduled jobs. In case of system crash, when it restarts, the system is able to rehydrate still active jobs.

To use the quartz scheduler, simply add its dependency:

<dependency> <groupId>com.iconsolutions.ipf.core.platform</groupId> <artifactId>ipf-flo-scheduler-persistent</artifactId> </dependency> In addition, for quartz scheduling it is necessary to provide a datastore for persistance. The following configuration show's how to do this:

ipf-scheduler { mongodb = { uri = "mongodb://localhost/ipf" database-name = "scheduled" } } Akka Scheduler Vs Quartz Scheduler Scheduler Type Pros Cons Akka

Short-term events that should last seconds to minutes

- Perhaps the name "Scheduler" was unfortunate, "Deferer" is probably more indicative of what it does.

- The Akka Scheduler is designed to setup events that happen based on durations from the current moment: You can say "fire this job in 15 minutes, every 30 minutes thereafter" but not "fire a job every day at 3pm".

- Akka's default scheduler is executed around a HashedWheelTimer – a potential precision loss for jobs, as it does not provide strong guarantees on the timeliness of execution.

- Scheduled jobs get lost when the system restarts

Quartz

- Jobs are scheduled to run when a given Trigger occurs. Triggers can be created with nearly any combination of the following directives. At a certain time of day (to the second), on certain days of the week, and so on.

- Jobs are given names by their creator and can also be organized into named groups. Triggers may also be given names and placed into groups, in order to easily organize them within the scheduler. Jobs can be added to the scheduler once, but registered with multiple Triggers.

More complex for short "schedule once" jobs

Configuration Action Timeouts HOCON configuration can be provided (usually in Akka application.conf) to configure the time-out duration of each action. When the duration has expired, the flow will receive an Action Timeout event for that configured Action.

The format of the config items are currently in flux and subject to change

The current format for the configuration is

FlowName.STATE_NAME.ActionName.Type=[Duration|Integer] FlowName : flow identifier or Any for wildcard

STATE_NAME : State identifier or Any for wildcard

ActionName : Action identifier or Any for wildcard

Type : one of timeout-duration, initial-retry-interval, or max-retries.

Where Duration is any format supported by HOCON

Specific example OBCreditTransfer.VALIDATING_SCHEME_RULES.ValidateAgainstSchemeRules.timeout-duration=2s This equates to: The Validate Against Scheme Rules action in the VALIDATING_SCHEME_RULES state in the O B Credit Transfer flow will time-out if not responded to in 2 seconds

Each part in the config also supports the Any keyword which will match on anything for that given part. It is applicable to flows, states and actions.

Any example Any.Any.ValidateAgainstSchemeRules.timeout-duration=10s Any.CheckingFraud.CheckFraud.timeout-duration=20s The Validate Against Scheme Rules action in the any state in any flow will time-out if not responded to in 10 seconds The Check Fraud action in the Checking Fraud state in any flow will time-out if not responded to in 20 seconds Backoff Types and Jitter The configuration allows for determining different backoff types:

EXPONENTIAL: 2^n scaling (where n is the initial delay). This is the default type.

LINEAR: 2n scaling (where n is the initial delay)

USER_DEFINED: Custom intervals that are defined in configuration

Jitter is enabled by default, but the configuration also allows for disabling jitter in the case that retries are so large that jitter would add a significant amount of delta. Imagine a retry happening in a day's time, the jitter for that would be +/- 5 hours each way. This might not be desirable.

So - for example - to configure 5 attempts of the CheckFraud action with no jitter, initially retrying after 10 seconds but then every 30 minutes, the configuration would be:

Any.CheckingFraud.CheckFraud.max-retries = 4 Any.CheckingFraud.CheckFraud.backoff-type = "USER_DEFINED" Any.CheckingFraud.CheckFraud.custom-backoffs = [10s,30m] Any.CheckingFraud.CheckFraud.jitter-enabled = false Using the HOCON duration format Another example here shows a linear retry (with jitter present so omitted because on by default)

Any.CheckingFraud.CheckFraud.initial-retry-interval = 1000 Any.CheckingFraud.CheckFraud.max-retries = 4 Any.CheckingFraud.CheckFraud.backoff-type = "LINEAR" This example will retry at 1/2/3/4 seconds.

Precedence The most specific configuration takes precedence i.e. if it matches on all 3 parts (flow, state and action). For actions, when there are multiple configurations items that might apply, the more specific state will override the more specific flow configuration.

Example action: Flow: Flow1 State: State1 Action: Action1 This would be the order of precedence of all the possible configurations that might apply to this action

1. Flow1.State1.Action1.timeout-duration 2. Any.State1.Action1.timeout-duration 3. Flow1.Any.Action1.timeout-duration 4. Any.Any.Action1.timeout-duration 5. Flow1.State1.Any.timeout-duration 6. Any.State1.Any.timeout-duration 7. Flow1.Any.Any.timeout-duration 8. Any.Any.Any.timeout-duration Processing Time Durations HOCON configuration can also be provided (usually in Akka application.conf) to configure how much time is allowed to be spent in passing between a pair of states, irrespective of the journey taken to reach the destination.

When the duration has expired, the flow will receive an ProcessingTimeElapsedEvent for the destination state.

The current format for the configuration is

ipf.flow.<flow-name>.processing-time.<source-state>.<destination-state>.timeout-duration=Duration|Integer] flow-name : flow identifier

source-state : State identifier

destination-state : The state the flow should reach in the allotted time

Where Duration is any format supported by HOCON

Specific example ipf.flow.OBCreditTransfer.processing-time.ValidatingAgainstSchemeRules.Complete.timeout-duration=2s This equates to:

The flow will produce a processing time elapsed event if the time taken from the ValidatingAgainstSchemeRules state to the Complete state exceeds 2 seconds

Offsets Offsets provide an enhanced mechanism of defining the timeout duration. The basic processing time looks simply at the time between two states. However, it may be necessary for the timeout to consider for example a specific customer defined start time. In this case we are able to use offsets as a way of enriching the duration.

For example, suppose that the client provides an accepted timestamp and that the flow must complete within 5 seconds of that time - rather than 5 seconds after the flow initiating. In this case, IPF can be provided with the accepted timestamp as an offset and then define a 5 second duration, but the actual timeout will be fired by allowing for the offset.

There are two types of offset:

System Offsets: each time IPF reaches a new state, a new offset is created.

Custom Offsets: these are user defined and can be provided to IPF.

Each offset has two attributes - a unique offset-id and the offset time itself.

To use an offset in the scheduling configuration, it is simply necessary to define in the configuration the offset-id to use. This is done by:

ipf.flow.OBCreditTransfer.processing-time.ValidatingAgainstSchemeRules.Complete.offset-id=anOffsetId To provide a custom offset to any flow, we simply need to provide it on the input via the withFlowOffset method.

It is also possible to provide the offsets from one flow to another. This is useful in parent-child relationships where there is a requirement for an offset to span multiple flows. To do this, on all actions the offset map of the current flow is provided as a parameter and this can simply be passed to the new flow via it's withOffsetMap method.

Automated Retries Recovery of the 'client implementation' can take shape in one of two ways;

Action retries - retry action after if transaction's state has not changed in X seconds

Action revivals - retry action on newly started cluster shard, and using an exponential backoff starting with the initial duration of X seconds any transactions in a non-terminal state will be retried.

Action Retries Action retries are used to prevent transaction's remaining stuck in a state, by issuing retries if an action does not change state within an acceptable (configurable) duration.

Interaction with Action timeouts For action timeouts see Scheduling.

Timeouts would usually result in a new state (Terminal) and therefore would not be subjected to retrying.

When timeouts cause a new state (non-terminal) then a retry would be attempted on the ActionTimeout if it remains stuck in its state.

Action Retry Configuration The configuration utilises the configuration policy of the ipf-scheduler (see Scheduling for configuration and action timeouts).

There are 2 configuration items necessary for action retries;

initial-retry-interval - the initial duration between retries, subsequent retries multiplied by a backoff factor of 2, i.e. if duration is 1 then 1,2,4,8.

max-retries - the maximum number of retries to attempt, i.e. [initial] + [max-retries]. 0 retries will effectively turn this functionality off.

jitter-factor - The percentage of randomness to use when retrying actions, default is 0.2.

For all Actions application.conf

Any.Any.Any.timeout-duration=10s Any.Any.Any.initial-retry-interval=3s Any.Any.Any.max-retries=2 Any.Any.Any.jitter-factor=0.2 For Specific Action

application.conf

Any.Any.Any.timeout-duration=10s Flow1.State1.Action1.initial-retry-interval=3s Flow1.State1.Action1.max-retries=2 Using the above configuration would create the following effect for Action1;

The following assumes ActionTimeout will lead to a terminal state, or at least a change of state.

Time (t+seconds) State Action 0

State1

Action1

3

State1

ActionRetry (Action1)

6

State1

ActionRetry (Action1)

10

Timeout (or whatever state ActionTimeout causes)

ActionTimeout

Action Revival Action revival is designed to recover transactions on a failed node. They differ from Action retries in the fact that they only fire when the cluster is started or re-started, a scenario not covered by the Action retries.

Revival will utilise action retries and continue from any retry attempt history, i.e. if a behaviour had already attempted 1 of say the configured 2 attempts then only 1 retry will be attempted.

The revival process will not attempt to recover a transaction in INITIAL or any terminal states. This was to protect the system from attempting to recover on all newly started shards.

The revival process will not attempt revival if the state has changed before the actionRevivalOffset (see configuration) has complete, as the transaction will no longer be deemed stuck.

Action revival is based upon Akka recovery signals this means that a recovery of state will occur when any of the following happen;

An Event Sourced Behaviour (ESB) is initialised for the first time

An ESB is revived after having been passivated (happens automatically after 120s by default)

An ESB was killed by an exception

An ESB is rebalanced and therefore restarted on another node

Action Revival Configuration There are 2 important configurations required to activate revival;

remember-entities - an Akka configuration item which causes Akka to automatically restart shards upon a shard restart. see akka docs

action-recovery-delay - an offset configured as duration which is imposed upon the system to allow any actions to change state before sending additional requests. An offset of 0 will turn this functionality off.

appliation.conf

akka.cluster.sharding.remember-entities = on application.properties

ipf.behaviour.config.action-recovery-delay=3s

Monitoring and Observability The strategy on IPF for monitoring and observability is achieved by utilising an event-driven paradigm, strict categorising of application behaviour, and extensibility for exposing this data through appropriate channels, with the best tooling.

Monitoring of IPF application services can be done in three primary ways:

HTTP APIs

Time-series metrics via Prometheus and Grafana

Application logging

Out of Scope:

Application performance monitoring (APM) - This is something to consider if such software is available at the customer's site. APM software such as Dynatrace or AppDynamics can help diagnose potential problems before they materialise.

Infrastructure Monitoring e.g. Brokers, containers

Definitions The "IPF application" is actually comprised of multiple runtime software packages. This section will describe the terminology that will be used herein:

Component Sometimes known as Description Needs to be in Akka cluster with other similar nodes? Payment Services (Customised)

Write side

The set of payment flows that the client has defined.

There can be multiple of these representing different sets of flows (e.g. credit transfer, recall, value-added service).

Yes

CSM Service

CSM Pack, Scheme Pack

An adapter for a payment provider such as an payment scheme, wallet, etc.

No

Support Services

Notification Service

Additional processing of events to third party systems.

Yes

Individual Service APIs (HTTP) There are certain HTTP APIs that are enabled by default which can be interrogated when the application is running, each serving a specific purpose.

Here's a summary of those APIs and the configuration items that set their hostnames and ports:

What is it? What does it do? Example use case API reference (or similar) Default hostname Default port Host override env var Port override env var Available on Spring Boot Actuator API

Spring Boot style metrics for the JVM, Spring config properties, beans, health

To verify the version of active libraries

Click here

0.0.0.0

8080

MANAGEMENT_SERVER_ADDRESS

MANAGEMENT_SERVER_PORT (set to -1 to disable)

Client Implementation(s)

Scheme Connector

Akka Management Cluster HTTP API

Information on the currently running Akka cluster for debug purposes

To verify cluster state and manually manage nodes

Click here

(the result of InetAddress.getLocalHost().getHostAddress() which is usually - but not always - 127.0.0.1)

8558

AKKA_MANAGEMENT_HTTP_HOSTNAME

AKKA_MANAGEMENT_HTTP_PORT

Client Implementation(s)

Spring Boot Actuator API Use this API for inquiring on the Spring ApplicationContext that's currently running on this particular node. Some interesting Spring Actuator endpoints for IPF:

conditions: Check that the write and read side of the application have been configured on the relevant nodes correctly

env: Display environment variables to check overrides are correct

health: Useful for liveness probes and the like

info: General application info (also useful for liveness probes)

More endpoints are available. Visit the Spring Boot Actuator link in the table above to see all the details. Also please note this particular section on how to enable and disable particular Actuator endpoints (MANAGEMENT_ENDPOINT_X_ENABLED where X is the relevant Actuator part).

For information on how to configure TLS for the Actuator endpoints, see this section.

Akka Cluster HTTP Management This API allows the user to interact with IPF's underlying Akka cluster using an HTTP interface.

The primary use for this API is to verify the cluster state. IPF client implementation components require that all "write" nodes serving the same set of flows (e.g. credit transfer, recall, digital wallet payout) are in an Akka cluster together. If this is not the case, no new work will be taken to avoid losing transactions.

The cluster finds other similar nodes by itself, but in case the application appears to not be consuming any new work, this should be the first port of call to ensure that the cluster is in a valid state and that the application is not in a "split brain" (multiple nodes being created in separate clusters).

Split brain situations can be resolved using the Akka Split Brain Resolver. More information is available on this topic on the Akka website.

The Akka Cluster HTTP Management endpoint also allows for update operations. If this behaviour is not desired, set the AKKA_MANAGEMENT_READ_ONLY environment variable to true to enable read-only mode where cluster information can only be retrieved but not updated.

TLS settings for Akka Management are the same as those for Spring Boot, but the server or management.server prefixes are replaced with akka.management. So for example to set the keystore path, the property would be akka.management.ssl.keystore or AKKA_MANAGEMENT_SSL_KEYSTORE (the Spring Boot equivalent being server.ssl.keystore or SERVER_SSL_KEYSTORE).

Time-Series Metrics Metrics are exposed via a Prometheus HTTP server which is interrogated over some set interval by Prometheus, and visualised using tools like Grafana and Kibana.

The AlertManager component performs configurable aggregation of events based on thresholds and conditions to translate this system behaviour into something that may need action for an Operator.

monitoring Spring Boot Dashboards Basic JVM metrics via the prometheus Spring Boot Actuator endpoint. There are several dashboards that can be used to visualise this data, but we recommend this one from the Grafana dashboard collection.

Akka Dashboards There are some out-of-the-box Grafana metrics that are available for Akka, documented and available for download here. The interesting Akka dashboards to look out for - as far as IPF is concerned - are:

Event Sourced Behaviours: Metrics on events and command being processed by IPF flows

Akka Streams and Akka Streams (extended): Connector stream processing performance metrics

IPF Specific Dashboards There are also some custom IPF-specific dashboards which are available:

Name Description Required Data IPF Connectors

Per-connector statistics on numbers of requests sent, received, and average response times per connector.

IPF Flow Timings

Per-flow statistics on how long flows are taking to be processed from initiation until they reach a final state.

IPF Resilience4j

Reports statistics on the connector circuitbreakers using metrics defined in Resilience4J.

IPF JVM Overview

Provides a few statistics on the JVM, for example heap memory, per pod in the cluster. For a more detailed dashboard see IPF JVM (Actuator)

IPF CSM Simulator

Reports on any the request/response metrics of CSM simulators that are deployed in the cluster.

Requires a namespace label in Prometheus

IPF Node Degradation

Reports on a wide range of application service metrics (MongoDB/Akka…) to determine the overall health of the IPF ecosystem, and help identify if there has been any degradation of performance over time.

IPF ODS Ingestion

Statistics relating to the ODS ingestion service reporting on any lag between ingestion of data as well as metrics relating to Kafka/MongoDB.

Pod name is assumed to be ods-ingestion.

Kafka consumer group for ingress is assumed to be called ipf-processing-data-ingress-consumer-group.

Kafka egress topic is assumed to be called IPF_PROCESSING_DATA.

IPF Transaction Status

Reports on statistics based on flow metrics to show transactions in completed or non-completed states

Completion state name is assumed to end with the word Completed or else all events will end in the Rejected panel

IPF ESB Actions & Inputs

Reports on Event Sourced Behaviours (ESBs) action and input metrics providing the rate of execution between these functions.

IPF JVM (Actuator)

Provides detailed statistics on the JVM from metrics provided by the Spring actuator.

IPF Requests Info

Provides overall and per-connector metrics on request and response timings.

Requires a namespace label in Prometheus

Exporter Configuration Spring Boot Actuator metrics are exposed via the same Actuator HTTP server documented above. See above for how to change the Actuator host and port.

Both Akka and IPF-specific Prometheus metrics are available on the same Prometheus exporter web server, which by default is configured to listen on all interfaces (0.0.0.0) and port 9001. To change these details, the relevant environment variables are:

CINNAMON_PROMETHEUS_HTTP_SERVER_HOST

CINNAMON_PROMETHEUS_HTTP_SERVER_PORT

If these properties are changed, remember to also change the Prometheus side so that Prometheus can collect time-series data from the correct address(es).

Logging IPF uses Logback for logging configuration. This allows the user to configure a logging setup which can mirror that of other applications developed inside the organisation, and make IPF report log data in the same way.

This document will explain some typical logging setups which can be used to output log data in various ways.

For all setups, the Logback configuration file needs to be mounted on the app's classpath. For an Icon-built image, this is always available at /[name of container]/conf. So if the container is named credit-transfer, then the Logback configuration can be mounted at /credit-transfer/conf/logback.xml.

Option 1: Elasticsearch/Logstash/Kibana A popular stack in the industry is ELK: a combination of Elasticsearch, Logstash (and/or Beats) and Kibana. It was formerly known as the ELK stack, but with the introduction of Beats, Elastic have been pushing the more generic "Elastic Stack" naming for this setup.

Either way, the setup looks like this:

elk First we need to configure Logstash to listen to a TCP port. Here's an example of how to configure that in Logstash:

input { tcp { port => 4560 codec => json_lines } } This makes Logstash listen on port 4560 for lines of JSON separated by the newline character \n. The Logstash Logback appender does this for us, and can be configured like so:

<?xml version="1.0" encoding="UTF-8"?> <configuration> <appender name="stash" class="net.logstash.logback.appender.LogstashAccessTcpSocketAppender"> <destination>127.0.0.1:4560</destination> <encoder class="net.logstash.logback.encoder.LogstashAccessEnc /> </appender>

<appender-ref ref="stash" /> </configuration> This will configure IPF application logs to be sent to Logstash. Configuring Logstash to connect to Elasticsearch (and Kibana to Elasticsearch) is out of scope of this document but can be found on the Elastic website here.

More examples, including:

Configuring TLS over the TCP connection, and;

A UDP variant of the appender

Can be found on the Logstash Logback appender's GitHub page here.

Configuring a Logstash Appender for System Events If you wish to aggregate IPF system events, consider using the com.iconsolutions.payments.systemevents.utils.Slf4jEventLogger

which forwards all received system events to an appender. This can be used in conjunction with this Logstash appender to push system events to an aggregator such as Elasticsearch as mentioned above.

Here's an example Logback config which takes Slf4jEventLogger events and sends them to our STASH appender:

<logger name="com.iconsolutions.payments.systemevents.utils.Slf4jEventLogger" level="DEBUG" additivity="false"> <appender-ref ref="STASH"/> </logger> Option 2: Splunk Aside from a specific setup such as the one above, a true twelve-factor app should output its logs - unbuffered - to stdout, and this can be analysed by software such as Splunk.

splunk Splunk provides an HTTP appender for Logback. This is documented here. That document also outlines some performance considerations for logging with HTTP, and also a TCP appender which can be used instead of HTTP.

The Logback template referred to in that document can be found here.

There are three mandatory fields:

url: The Splunk URL to forward to

token: The token provided by Splunk for authentication and authorisation

index: The Splunk index (repository) for storing this log data

Option 3: Files (Not Recommended) Logging to file breaks cloud-native principles about not making assumptions about an underlying file system. Logs should be treated as data streams instead of files that need maintaining.

Only use this approach as a last resort when it's absolutely not possible to use a more modern approach for logging. For more information please see XI. Logs on the Twelve-Factor App.

It is possible to specify a normal Logback file appender. A typical logback.xml might look like:

logback.xml <?xml version="1.0" encoding="UTF-8" scan="true"?> <configuration> <jmxConfigurator /> <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppen
<file>/opt/ipf/logs/credit-transfer-service.log</file> <rollingPolicy class="ch.qos.logback.core.rolling.FixedW
<fileNamePattern>/opt/ipf/logs/credit-transfer-service.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> This file creates the following configuration:

The logfile is parsed for live changes every minute (scan="true")

Logging configuration can be modified live with JMX. More information on this is available here

File logging to /opt/ipf/logs/credit-transfer-service.log

Files rolled over to /opt/ipf/logs/credit-transfer-service.log.n, where n is a number between 1-20, and is rolled over when credit-transfer-service.log reaches 50 MB. Note that only 20 such files (i.e. a total of 1 GB of log data) will be kept

This file can also be scraped by a framework such as Splunk forwarder or Beats as shown in the diagram below:

file System Events What are IPF System Events IPF applications are underpinned by a System Event framework, providing pub/sub capabilities between application components.

Hierarchical, extensible, versioned, catalogued. All application areas: Technical, Functional & Lifecycle. All events include common properties: Source location, Creation time, Level, Type, association context, simple to get all System Events for a given payment

The fundamental data framework for capturing application behaviour, providing a place to build extensible event-driven functionality, which can then be feed into supporting tasks such as providing data for monitoring and alerting.

System Event processors subscription pattern : "Act on all WARN level Infrastructure System Events"

Configure multiple processors, functional style, store /transform – emit over a Connector

All the System Events that an IPF Service may emit are catalogued and have a schema, they are versioned against the deployed software

New client-specific events are encouraged to be added for a given Solution. Providing a very clean extension for leverage the existing framework and providing additional data insights

events Where they are Defined The full list of system events that IPF produces are listed here

Troubleshooting Error Messages The events in this table are ERROR events logged by IPF and describe recommended remediation steps to follow if encountered.

Getting Started Create a new IPF Application Project

Defining The Read Side

---------- Create a new IPF Application Project To create a brand new IPF Application project we advise using the Archetype starter as this is the quickest way to bootstrap a new project using the IPF Toolkit. The Maven archetype will generate a Maven project (using the Icon BOM as its parent) containing;

an application skeleton (home for the application runtime code)

a flow domain (artefacts related to the DSL and flow definitions)

a test module (containing the docker application configuration)

From this base you can then wire in any additional capability you need, either from the Flo Starter Projects or other IPF component libraries.

---------- Defining The Read Side The ipf-read-starter provides a base mongo setup that can be extended for use in implementations. The read side supports multiple different flows running with a single read side.

The key to the read side setup is the abstract 'MongoReadModelEntity' class. It provides:

persistenceId - the persistence id (the IPF ID e.g. Payoutflow|ABC)

originatingId - the originating id of the flow (e.g. UPRID)

originatingMessageType - the originating message type (e.g. ExecutePaymentRequest)

firstEventTime - the time the first event was record on the transaction

lastProcessedEventTime - the time the most recent event was record on the transaction

lastEventName - the most recent event name

initiatingTimestamp - when the flow as initiated.

alternativeIds - a list of alternative ids that the transaction may be known by

eventSummaries - a list of event summary details for all the events on the transaction

payload - the payload for the transaction, as per the api (for example could be CreditTransferPayload, ReturnsPayload etc.)

The implementation of this class, if application specific as it depends on being populated by data received on custom events. For this reason, it is necessary

for the application to provide the concrete implementation. To do this, the following methods will need to be implemented:

protected abstract Supplier<T> payloadCreator();

protected abstract BiFunction<Event, T, T> payloadUpdater(); payloadCreator - this is required to provide a new instance of the payload type.

payloadUpdater - this is required to determine how the given event should update the current payload.

originatingMessageTypeSupplier - this is required to specify the name (from the message log) of the initiating message for the flow.

In addition to the above 3 methods, there are a number of methods that have default implementeds that can be overriden:

protected Set<String> getAlternativeIdsFrom(Event event) { return Collections.emptySet(); } getAlternativeIdsFrom - by defaulting returning an empty collection, this method should extract any id's that should be used for searching on.

getInitiatingTimestampFrom - by default this uses the createdAt time of the first event received, but can be overriden to use the data from any event.

getOriginatingIdFrom - by default this uses the commandId of the first event received, but can be overriden to use the data from any event.

Once the concrete implementation of the MongoReadModelEntity has been created, it needs to be registered by creating a factory that will determine which implementation should be used based on the IPF ID that is received.

public interface ReadModelEntityFactory { ReadModelEntity create(String persistenceId); } com.iconsolutions.ipf.platform.read.transaction This needs to be registered as a spring bean.

Example The following shows an example implementation of the MongoReadModelEntity.

@Data @AllArgsConstructor public class TestCreditTransferTransactionReadModel extends MongoReadModelEntity<CreditTransferPayload> {

@Override protected Supplier<CreditTransferPayload> payloadCreator() { return () -> CreditTransferPayload.builder().build(); }

@Override protected BiFunction<Event, CreditTransferPayload, CreditTransferPayload> payloadUpdater() { return (evt, returnPayload) -> { if (evt instanceof TestFIToFICustomerCreditTransferEvent) { returnPayload.setCreditTransfer(((TestFIToFICustomerCreditTransferEvent) evt).getCreditTransfer()); } return returnPayload; }; }

} To register this we create the factory:

@Slf4j public class ReadModelEntityTestFactory implements ReadModelEntityFactory {

@Override public ReadModelEntity create(String persistenceId) { log.debug("PersistenceId found on read side: {}", persistenceId); return new ReadSideFlowTestModel(); } }

---------- Handle Schema Evolution in IPF Events Over the life of an IPF flow, changes will be made to the events that are persisted by that flow. Changes to flows can happen for many reasons, such as:

Better business understanding of the flow leading to a change in requirements

Design-time assumptions not being correct and therefore requiring a change to the initially agreed-upon events

Regulatory reasons for needing to capture more data in an event

The technical nature of these changes can vary, but they are generally variants on the below patterns, which will also serve as the subheadings for this guide.

The patterns for schema evolution are:

Adding a data field to an event

Changing the fully-qualified class name (FQCN) of an event

Changing the name of a data element in an event

Changing the data type of an element in an event

Splitting an event into multiple, smaller events

Removing an event

The process for updating events is out of scope of this document. Follow the DSL portion of the tutorial for event definition and modification.

These event migrations should only be implemented if the flow itself is not changing.

If the flow itself is also changing in addition to its events, then consider creating a new version of the flow altogether. See the DSL portion of the tutorial for how to define multiple versions of flows

Adding a Data Field to an Event Consider v1 of an event being as follows:

public class UserRegistered extends DomainEvent { private final String name; } This would be serialised in JSON as:

{"name": "jeff"} If the new data being added is simply a new optional field, like this:

package com.myorg.events;

import java.time.LocalDate;

public class UserRegistered extends DomainEvent { private final String name; private final LocalDate dob; } The old serialised version will still be successfully parsed as a UserRegistered, and no change is required. But of course the dob field will be null.

However, business rules may not allow the dob field to be null, but there may be a special placeholder date of birth 0001-01-01 that you can use to indicate that the lack of availability of a date of birth for this user. In which case, this JacksonMigration will check for dob, and - if absent - will set it to the default value before it is deserialised as UserRegistered:

package com.myorg.migrations;

import com.fasterxml.jackson.databind.JsonNode; import com.fasterxml.jackson.databind.node.ObjectNode;

public class DobFillerMigration extends JacksonMigration {

private static final String DOB_FIELD_NAME = ”dob”;

@Override public int currentVersion() { return 2; }

@Override public JsonNode transform(int fromVersion, JsonNode json) { if (!json.has(DOB_FIELD_NAME)) { ((ObjectNode) json).set(DOB_FIELD_NAME, ”0001-01-01”); } } } And then bound to the UserRegistered event by adding it to the IPF configuration like so:

akka.serialization.jackson.migrations { ”com.myorg.events.UserRegistered” = ”com.myorg.migrations.DobFillerMigration” } Remember, if the new field is allowed to be null, no migration needs to be written.

Changing the fully-qualified class name (FQCN) of an event, or renaming an event This cannot be remedied by fixing a JacksonMigration, since JacksonMigration works on the body and not the type.

However, if nothing else has changed apart from the fully qualified classname (or just the name) of the event, and the Icon MongoDB Akka Persistence plugin is being used, this can be remedied using a MongoDB update statement.

As an example, if the package name of the events was misspelt as com.myorg.evnets and we want to correct the typo, the following update statement will change the package name of all com.myorg.evnets events to com.myorg.events:

const updates = []; db.journal.find({”eventPayloads.payload.type”:/com.iconsolutions.instantpayments/}) .forEach(doc => { doc.eventPayloads.forEach(pld => { const oldFQCN = pld.payload.type; const newFQCN = oldFQCN.replace(”com.iconsolutions”, ”com.monkey”); updates.push({”updateOne”: { ”filter”: {$and: [{”_id”: doc._id}, {”eventPayloads.payload.type”:oldFQCN}]}, ”update”: {$set: {”eventPayloads.$.payload.type”: newFQCN}} }}) }) }); print(updates); // uncomment after this line to actually run the update // const result =

db.journal.bulkWrite(updates); // print(JSON.stringify(result)); Some things to note about the snippet above:

The default name of the journal is journal but this may be overridden with iconsolutions.akka.persistence.mongodb.journal-collection

Changing the Name of a Data Element in an Event This can also be resolved by writing a JacksonMigration.

Consider v1 of the event being:

import java.time.LocalDate;

public class UserRegistered extends DomainEvent { private final String name; private final LocalDate dob; } This would be serialised in JSON as:

{”name”: ”jeff”, ”dob”: ”1985-01-01”} But you decide that dob might not be so clear and decide to rename it to dateOfBirth:

package com.myorg.events;

import java.time.LocalDate;

public class UserRegistered extends DomainEvent { private final String name; private final LocalDate dateOfBirth; } Write the following migration to rename dob to dateOfBirth:

package com.myorg.migrations;

import com.fasterxml.jackson.databind.JsonNode; import com.fasterxml.jackson.databind.node.ObjectNode;

public class DobRenameMigration extends JacksonMigration {

private static final String OLD_FIELD_NAME = ”dob”; private static final String NEW_FIELD_NAME = ”dateOfBirth”;

@Override public int currentVersion() { return 2; }

@Override public JsonNode transform(int fromVersion, JsonNode json) { if (json.has(OLD_FIELD_NAME)) { //get value of dob var seqValue = json.get(OLD_FIELD_NAME); //set it to new field ((ObjectNode) json).set(NEW_FIELD_NAME, seqValue); //remove old field ((ObjectNode) json).remove(OLD_FIELD_NAME); } return json; } } And then bound to the UserRegistered event by adding it to the IPF configuration like so:

akka.serialization.jackson.migrations { ”com.myorg.events.UserRegistered” = ”com.myorg.migrations.DobRenameMigration” } Changing the Data Type of an Element in an Event Changing a type can mean multiple things, such as:

A data element splitting into multiple elements

Moving from a simple type to a complex type

But both are handled in more-or-less the same way, which is writing a JacksonMigration to map the values from the old version of the event to its new representation.

Continuing with the previous example v1 event:

package com.myorg.events;

import java.time.LocalDate;

public class UserRegistered extends DomainEvent { private final String name; private final LocalDate dateOfBirth; } Imagine the business wants to devolve the name details into a separate Name object:

package com.myorg.model;

import java.time.LocalDate;

public class Name { private final String firstName; private final String middleName; private final String lastName; } So the new event looks like this:

package com.myorg.events;

import com.myorg.model.Name; import java.time.LocalDate;

public class UserRegistered extends DomainEvent { private final Name name; private final LocalDate dateOfBirth; } Assuming the following imaginary migration rule from the business:

If a name contains two tokens, split into first and last name in that order

If a name contains three tokens, split into first, middle and last name in that order

This is not a good way to devolve someone's name into constituent parts! Under these rules, the migration would be:

package com.myorg.migrations;

import com.fasterxml.jackson.databind.JsonNode; import com.fasterxml.jackson.databind.node.JsonNodeFacto
com.fasterxml.jackson.databind.node.ObjectNode;

public class NameMigration extends JacksonMigration {

@Override public int currentVersion() { return 2; }

@Override public JsonNode transform(int fromVersion, JsonNode json) { var name = json.get("name").asText(); var nameNode = JsonNodeFactory.instance.objectNode(); var nameSplit = name.split("\s"); if (nameSplit.length == 2) { nameNode.set("firstName", nameSplit[0]); nameNode.set("lastName", nameSplit[1]); } else if (nameSplit.length == 3) { nameNode.set("firstName", nameSplit[0]); nameNode.set("middleName", nameSplit[1]); nameNode.set("lastName", nameSplit[2]); } ((ObjectNode) json).set("name", nameNode); return json; } } And in the IPF configuration:

akka.serialization.jackson.migrations { "com.myorg.events.UserRegistered" = "com.myorg.migrations.NameMigration" } Splitting an Event Into Multiple, Smaller Events This can be implemented using an EventAdapter. The approach is similar to writing a JacksonMigration, but is implemented after the events have been deserialised. Consult the Akka docs for more information on this topic: Split large event into fine-grained events

To implement an EventAdapter that will be passed into the EventSourcedBehaviour for the flow, when initiating the domain, use withBehaviourExtensions to supply a BehaviourExtensions implementation like this:

import java.util.Optional;

public class MyBehaviourExtensions implements BehaviourExtensions { @Override public Optional<EventAdapter<Event, Event» eventAdapter() { return Optional.of(new EventAdapter<>() { @Override public Event toJournal(Event event) { return event; }

@Override public String manifest(Event event) { return ""; }

@Override public EventSeq<Event> fromJournal(Event event, String manifest) { //if it's MySuperEvent, devolve it into two smaller events if (event instanceof MySuperEvent) { var mse = (MySuperEvent) event; return EventSeq.create(List.of(new MySmallerEvent1(mse.data1()), new MySmallerEvent2(mse.data2()))); } //otherwise return any other event as-is return EventSeq.single(event); } }); } } and then…

@EventListener public void init(ContextRefreshedEvent event, MyBehaviourExtensions myBehaviourExtensions) { new MyDomain.Builder(actorSystem) .withEventBus(eventBus) .withSchedulerAdapter(schedulerAdapter) .withSystemAActionAdapter(new SampleSanctionsActionAdapter(sanctionsAdapter)) .withSystemBActionAdapter(new SampleSanctionsActionAdapter(sanctionsAdapter)) .withBehaviourExtensions(myBehaviourExtensions) .build(); } How to add the BehaviourExtensions implementation Removing an Event This is the same as above, but instead emit an EventSeq.empty (the rest of the implementation is the same):

@Override public EventSeq<Event> fromJournal(Event event, String manifest) { //if it's MyUnwantedEvent, pretend we didn't see it if (event instanceof MyUnwantedEvent) { return EventSeq.empty(); } //otherwise return any other event as-is return EventSeq.single(event); } JAVA copy icon Copied!

---------- Troubleshooting Error Messages The events in the following table are ERROR events logged by IPF and describe recommended remediation steps to follow if encountered.

Error events Name Description Level Remediation AuthenticationFailed

Published when authentication fails

ERROR

If an Authentication error occurs please check the following:

the logs of the service the connector belongs to for any errors

CorrelationNotFound

Published when an CorrelationId cannot be matched with a UnitOfWorkId

ERROR

If an CorrelationId cannot be found please check the following:

the correlation service is available

the mapping of fields is correct

the field is available in the input message

InboundMessageFailed

Event published when any aspect of the message receive process fails (mapping, receiver function,etc.) - contains the exception that was raised

ERROR

If an Inbound Message Failure occurs please check the following:

the logs of the service the connector belongs to for any errors

OutboundMessageFailed

Published when any error relating to sending of messages is raised (mapping, enqueue, etc.)

ERROR

If an Outbound Message Failure occurs please check the following:

the logs of the service the connector belongs to for any errors

OutboundMessageValidationFailed

Published in case of outgoing message validation failure

ERROR

---------- Securing HTTP Endpoints By default, the IPF APIs do not authenticate inbound requests. This can be configured in the client implementation by including the spring-boot-starter-security dependency to the project. The full documentation on Spring Security can be found here.

Example This section will demonstrate how a client implementation might add validation of JWT access tokens on the endpoints exposed by the ipf-read-starter.

Prerequisites The read-side client implementation should be implemented and working with all valid requests being accepted.

Add Maven Dependencies For this example, two dependencies are required. The first is the aforementioned spring-boot-starter-security module provided by the Spring framework.

<dependency> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-security</artifactId> </dependency> The second is a JWT library that will be used to validate tokens on incoming requests.

<dependency> <groupId>com.auth0</groupId> <artifactId>java-jwt</artifactId> </dependency> Service to Validate JWTs Next, a service will be created to validate and decode JWTs. The service requires the secret used to sign the token so that it can verify the token's signature. The secret is autowired using the @Value annotation provided by Spring.

@Service public class JWTService {

private final String secret;

@Autowired public JWTService(@Value("${ipf.security.jwt.secret}") String secret) { this.secret = secret; }

public Optional<DecodedJWT> decodeAccessToken(String accessToken) { return decode(secret, accessToken); }

public List<GrantedAuthority> getRoles(DecodedJWT decodedJWT) { return decodedJWT.getClaim("role").asList(String.class).stream() .map(SimpleGrantedAuthority::new) .collect(Collectors.toList()); }

private Optional<DecodedJWT> decode(String signature, String token) { try { return Optional.of(JWT.require(Algorithm.HMAC512(signature.getBytes(StandardCharsets.UTF_8))) .build() .verify(token.replace("Bearer ", ""))); } catch (Exception ex) { return Optional.empty(); } } } Authorization Filter Component When a request is made, it is passed through a chain of WebFilter classes. Most of the chain is provided by Spring out of the box, though additional filters can be provided by the client. In this example one such WebFilter will be implemented to validate that requests made to secure endpoints have set the Authorization header with a valid token, otherwise the request should fail (unless another authentication filter is provided to allow requests to be authenticated a different way).

@Component public class JWTAuthorizationFilter implements WebFilter {

private final JWTService jwtService;

public JWTAuthorizationFilter(JWTService jwtService) { this.jwtService = jwtService; }

@Override public Mono<Void> filter(ServerWebExchange exchange, @NotNull WebFilterChain chain) { String header = exchange.getRequest().getHeaders().getFirst(HttpHeaders.AUTHOR

if (header == null || !header.startsWith("Bearer ")) { return chain.filter(exchange); }

return jwtService.decodeAccessToken(header) .map(token -> new UsernamePasswordAuthenticationToken(token.getSubject(), null, jwtService.getRoles(token))) .map(token -> chain.filter(exchange) .subscriberContext(context -> ReactiveSecurityContextHolder.withAuthentication(token))) .orElse(chain.filter(exchange) .subscriberContext(context -> ReactiveSecurityContextHolder.clearContext().apply(context))); } } The JWTAuthorizationFilter implements the WebFilter class' single method filter. It checks that the request's Authorization header for is prefixed with "Bearer " followed by a valid JWT.

If the header is not set, the filter simply passes the exchange to the next filter without doing anything.

If the token is valid, the filter sets the authentication into the security context.

Otherwise, the filter clears the security context.

Configure the SecurityWebFilterChain Finally, the SecurityWebFilterChain must be configured as a Spring bean.

@Configuration @EnableWebFluxSecurity public class WebSecurityConfig {

@Bean public SecurityWebFilterChain secureFilterChain(ServerHttpSecurity http, JWTAuthorizationFilter jwtAuthorizationFilter) { return http .formLogin(ServerHttpSecurity.FormLoginSpec::disable) .httpBasic(ServerHttpSecurity.HttpBasicSpec::disable) .csrf(ServerHttpSecurity.CsrfSpec::disable) .logout(ServerHttpSecurity.LogoutSpec::disable) .authorizeExchange(spec -> spec .pathMatchers("/actuator/health").permitAll() .pathMatchers("/actuator/**").hasAnyRole("ADMIN") .anyExchange().authenticated()) .addFilterAt(jwtAuthorizationFilter, SecurityWebFiltersOrder.AUTHORIZATION) .securityContextRepository(NoOpServerSecurityContextRepository.getInstance()) .build(); }

@Bean public AbstractJackson2Decoder jacksonDecoder(ObjectMapper mapper) { return new Jackson2JsonDecoder(mapper); }

} This configuration is set up to disable some features which aren't required (form login, http basic, csrf and logout). The two most important aspect are the authorizeExchange specification and addFilterAt method.

The authorizeExchange block specifies a number of path matchers that will determine what shall be done with requests that match those matchers.

The first matcher ensures any requests made to "/actuator/health" will be permitted, that is, no authorization token is required.

The second matcher will only allow requests made to "/actuator/**" which have the role "ADMIN". This implicitly requires the token to be valid too.

The last is a catch-all matcher which will ensure that any unspecified requests are authenticated, i.e. they have a valid token.

The addFilterAt method simply adds the JWTAuthorizationFilter that was created previously into the filter chain and places it at the AUTHORIZATION stage within the chain. This ensures the request is passed to the filter at the right time.

Testing To test the newly added configuration the ipf.security.jwt.secret property must be set. In this example the value secret will be used for simplicity.

ipf.security.jwt.secret=secret A tool to create valid JWTs is required, the tool used in this demonstration can be found in this GitHub repository.

Set the following environment variables.

JWT_SECRET=secret JWT_EXPIRY_SECONDS=180 # 3 minutes JWT_ROLE=USER Then create the JWT with the following command.

JWT=$(jwt encode \ --alg HS512 \ --sub user \ --exp $(expr $(date +%s) + ${JWT_EXPIRY_SECONDS}) \ --secret ${JWT_SECRET} \ '{"role": ["ROLE_${ROLE}"]}') At this point the service should be started, and the configuration can be tested using curl.

Verify the "/actuator/health" endpoint does not need to be authenticated curl localhost:8080/actuator/health # should return 200

Verify the "/transactions" endpoint needs to be authenticated curl localhost:8080/transactions -i # should return 401 curl -H "Authorization: Bearer $JWT" localhost:8080/transactions # Should return 200

Verify the "/actuator/info" endpoint needs to be authenticated and have role ADMIN curl localhost:8080/actuator/info -i # should return 401 curl -H "Authorization: Bearer $JWT" localhost:8080/actuator/info -i # should return 401 JWT_ROLE=ADMIN JWT=<same as before> curl -H "Authorization: Bearer $JWT" localhost:8080/actuator/info -i # should return 200 ---

---

---------- Using the persistent scheduler to support flows The ipf-flo-scheduler module provides an out of the box mechanism to support the usage of the ipf-flo-scheduler with a flo without having to add any additional configuration.

To use, it is simply necessary to add the dependency to the module:

<dependency> <groupId>com.iconsolutions.ipf.core.platform</groupId> <artifactId>ipf-flo-scheduler</artifactId> </dependency> This will inject all the necessary components via spring into your project. All that is then required is the standard configuration for determining which actions and states require scheduling information ---

---

---------- Sample Duplicate Check Function Implementation Duplicate check via transaction cache is modeled as a Domain Function implementation calling the configured Transaction Cache Service.

This uses PersistentTransactionCacheService backed by MongoDB, so the necessary data to populate the cache and perform duplicate checks survives a service restart. The definition of the Domain Function is (currently) done within the MPS project as the data points may be solution specific. See screenshots of an example Domain Function definition and its usage within a flow (calling the function and handling the response events) from a reference solution:

duplicate check flow duplicate check domain The corresponding implementation is as follows:

package com.iconsolutions.instantpayments.credittransfer.sample.config;

import com.iconsolutions.ipf.core.platform.txcache.service.PersistentTransactionCacheService;
import com.iconsolutions.ipf.core.platform.txcache.repository.TransactionCacheRepository;
import com.iconsolutions.ipf.core.platform.txcache.service.TransactionCacheService;
import com.iconsolutions.ipf.core.shared.retry.RepositoryRetryProvider; import com.iconsolutions.ipf.payments.domain.clearing_and_settlement.pacs008.FIToFICustomerCreditTransfer
import io.vavr.control.Try; import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.util.List; import java.util.Optional;

@Configuration public class TransactionCacheServiceConfig {

/** * * This definition builds a TransactionCacheService instances that handles entries for pacs.008 objects. * The following fields are being used to determine if the transaction is a duplicate and forms the key: * EndToEndId, DbtrAcct.Id, LclInstrm.Prtry/Cd, IntrBkSttlmAmt.Value, IntrBkSttlmAmt.Ccy * * @param transactionCacheRepository - by default this bean is provide by spring-data repository through the * com.icon.ipf.core.platform:ipf-transaction-cache dependency * * @param repositoryRetryProvider - by default this bean is provided by the SharedRepositoryConfiguration config * from the com.iconsolutions.ipf.core.shared:shared-application-common dependency. * Added below for completeness * * <pre> * @Bean * @ConditionalOnMissingBean * public RepositoryRetryProvider repositoryRetryProvider() { * return new RepositoryRetryProvider(* 0, * t -> false, * null); * }</pre> */ @Bean public TransactionCacheService<FIToFICustomerCreditTransfer> debtorCTTransactionCacheService(TransactionCacheRepository transactionCacheRepository, RepositoryRetryProvider repositoryRetryProvider) { return new PersistentTransactionCacheService<FIToFICustomerCreditTransfer>(fi2fi -> { var cdtTrfTxInf = fi2fi.getCdtTrfTxInf().get(0); return List.of(cdtTrfTxInf.getPmtId().getEndToEndId(), Try.of(() -> cdtTrfTxInf.getDbtrAcct().getId().getOthr().getId()).getOrElse(
-> cdtTrfTxInf.getDbtrAcct().getId().getIBAN()), Try.of(() -> Optional.ofNullable(cdtTrfTxInf.getPmtTpInf()

-> cdtTrfTxInf.getPmtTpInf().getLclInstrm().getCd()), cdtTrfTxInf.getIntrBkSttlmAmt().getValue().toString(
cdtTrfTxInf.getIntrBkSttlmAmt().getCcy()); }, transactionCacheRepository,
repositoryRetryProvider); } } package com.iconsolutions.instantpayments.credittransfer.sample.adapter.action

import com.iconsolutions.instantpayments.domain.credittransfer.actions.CheckFunctionalDuplicateAction;
import com.iconsolutions.instantpayments.domain.credittransfer.adapters.DuplicateCheckingActionPort;
import com.iconsolutions.instantpayments.domain.credittransfer.domain.CredittransferDomain;
import com.iconsolutions.instantpayments.domain.credittransfer.inputs.CheckFunctionalDuplicateResponseInp
import com.iconsolutions.instantpayments.domain.credittransfer.inputs.responsecodes.AcceptOrRejectCodes;
import com.iconsolutions.instantpayments.domain.credittransfer.reasoncodes.ISOReasonCodes;
import com.iconsolutions.ipf.core.platform.txcache.service.TransactionCacheService;
import com.iconsolutions.ipf.payments.domain.clearing_and_settlement.pacs008.FIToFICustomerCreditTrans
import lombok.AllArgsConstructor; import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.CompletionStage;

/** * This class is the implementation of an external domain action adapter that calls the transaction cache in order to perform * a duplicate check. * * It depends on an instance of TransactionCacheService that has been defined in TransactionCacheServiceConfig */ @Slf4j @AllArgsConstructor public class SampleDuplicateCheckingActionAdapter implements DuplicateCheckingActionPort {

private final TransactionCacheService<FIToFICustomerCreditTransfer> transactionCacheService;

@Override public CompletionStage<Void> execute(CheckFunctionalDuplicateAction action) { return saveAndVerify(transactionCacheService, action) .thenCompose(CredittransferDomain.duplicateChecking()::handle) .thenAccept(result -> log.debug("DuplicateCheckingActionAdapter completed with {}", result.getResult())); }

/** * In this example we eagerly save the payment and then verify if it has any duplicates * - save it to the cache * - re-read it from the cache using the derived key * - if more than one entry is found then at least one previously existed and therefore it IS a duplicate * - return the appropriate response to the process response Input * * Note: * This "eager" save is a preferable alternative to the process of: * - read from cache with derived key * - if there is a result then flag a duplicate else save to the cache * * As it reduces the window for concurrent duplicates slipping through, at the cost of an extra record being stored * * @param cacheService * @param action */ public CompletionStage<CheckFunctionalDuplicateResponseInput> saveAndVerify(TransactionCacheService<FIToFICustomerCreditTransfer> cacheService, CheckFunctionalDuplicateAction action) { return cacheService.saveToCache(action::getFlowType, action.getCustomerCreditTransfer()) .thenCompose(entry -> cacheService.findInCache(action::getFlowType, action.getCustomerCreditTransfer())) .thenApply(entries -> entries.size() == 1 ? accepted(action.getId()) : rejected(action.getId())); }

private CheckFunctionalDuplicateResponseInput accepted(String aggregateId)

{ return new CheckFunctionalDuplicateResponseInput.Builder(aggregateId, AcceptOrRejectCodes.Accepted) .build(); }

private CheckFunctionalDuplicateResponseInput rejected(String aggregateId) { return new CheckFunctionalDuplicateResponseInput.Builder(aggregateId, AcceptOrRejectCodes.Rejected) .withReasonCode(ISOReasonCodes.AM05) .build(); }

} ---

---------- Get Aggregate Data for Use in an External Call Getting the aggregate data for use in an external domain function can be done with the help of the domain functions as mentioned in Domain Operations.

All domain operations are asynchronous and return a CompletionStage. So if we want to use the result of the getAggregate we need to chain calls with thenCompose/thenApply to ensure we are not blocking anywhere.

Below is an example of sending some data out to a SendConnector after we have retrieved the aggregate data, as it is required to populate some data in the request message of the SendConnector.

public class SampleFraudActionAdapter implements FraudActionPort {

private final SendConnector<FraudRequest, OlafRequest> fraudSendConnector;

@Override public CompletionStage<Void> execute(CheckFraudAction action) { FraudRequest fraudRequest = new FraudRequest(); fraudRequest.fiToFICustomerCreditTransfer = action.getCustomerCreditTransfer(); return CredittransferDomain.getAggregate(action.getId()) .thenCompose(aggregate -> { fraudRequest.customFields.put("mykey", aggregate.getStatus().getStatus()); return fraudSendConnector.send(action.getProcessingContext(), fraudRequest) .thenAccept(deliveryOutcome -> log.debug("FraudAdapter completed with {}", deliveryOutcome.getDeliveryReport().getOutcome())); }); }

private class FraudRequest { public FIToFICustomerCreditTransfer fiToFICustomerCreditTransfer; public Map<String, String> customFields; } }

---------- IPF Archetype The IPF archetype is a Maven Archetype that provides engineers with a quick and efficient way to boostrap a new project using the IPF SDK. The archetype will generate a new IPF project that includes all the necessary dependencies and configuration to be able to run as a standalone project.

To understand more about the key concepts and configuration options, we suggest you start with Archetype Concepts. Alternatively you can just get going and create a quick IPF project using the Getting Started guides.

-- Archetype Concepts Output Once the archetype has run, it will generate a new project. This key configuration option to consider here is the 'artifactId'. This will be used to determine the folder name of the generated project. So if running with an 'artifactId' for example of 'my-test' then this will lead to a new folder being generated called my-test relative to the directory the archetype is run from.

Inside this folder will be a fully functional IPF application which can be used as the basis from which to build your IPF application"

The Icon BOM The generated project uses the Icon BOM (a Maven Bill of Materials) as its parent. This provides all the dependencies and their versions for the generated application to use. If the project you are building needs to extend for your own parent, this can be updated post generation.

Run Modes The archetype can be run in either:

interactive mode - this is where the user specifies a minimal set of configuration on initial creation command and then the archetype will prompt the user to complete each of the remaining config items in turn.

batch mode - this is where the user is responsible for supplying all the configuration options to the archetype on the initial creation command. There is then no ability to further enrich or edit the commands.

Configuration Options The following table defines the different configuration options available for use within the archetype.

Parameter

Description

Type

Example

archetypeGroupId

Group of the archetype.

String

com.iconsolutions

archetypeArtifactId

The archetype id.

String

icon-archetype

archetypeVersion

Version to use of the archetype.

String

1.3.1

ipfVersion

Version of IPF to use

String

2022.3

groupId

Group id, also used as package.

String

my.test

artifactId

The name of the new archetype.

String

my-test

version

The version to start with.

String

1.0.0-RELEASE

solutionName

Name of the solution.

Alpha first uppercase, rest lower

Solution

modelName

Name of the model being used.

Alpha all lowercase

model

flowName

Name of the initial flow.

Alpha first uppercase, rest lower

MyFlow

includeApplication

Include application modules.

(yn)

y

includeTest

Include test modules

(yn)

y

useDocker

Generate project using docker

(yn)

y

dockerRegistry

Docker registry for docker images

String

registry.ipf.iconsolutions.com

You must supply the target version of both the archetype (archetypeVersion) and the 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!

-- Getting Started This guide is aimed to get a user from never having used IPF before to having a running application in under 5 minutes! It assumes that you already have access to the appropriate maven repository containing an IPF release.

We're going to build the simplest IPF application possible:

quick start basic

Let's look a little deeper at what is actually making up the IPF application here:

quick start ipf

Here you can see that the application is built using a number of IPF components.

We'll use the IPF Archetype to build a skeleton application that contains a single processing flow. We can then use a simple http controller to invoke the flow and see our results on the IPF Developer Application.

Running the Archetype Firstly navigate to the directory that you want your generated project to be installed to. Whilst this directory does not need to be empty it must not contain a folder with the name of the project to generate and most not have a pom.xml file within it.

mvn archetype:generate -B -DarchetypeGroupId=com.iconsolutions.ipf.core.archetype -DarchetypeArtifactId=icon-archetype -DarchetypeVersion=1.*.* -DgroupId=quick.start -DartifactId=quick-start -Dversion=0.0.1-SNAPSHOT -DipfVersion=202*.*.* -DsolutionName=Quickstartsolution -DmodelName=quickstartmodel -DflowName=Quickstartflow -DincludeApplication=y -DincludeTest=y You must supply the target version of both the archetype (archetypeVersion) and the 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, the project will be generated. You can see this in the output:

[INFO] Project created from Archetype in dir: /build/archdocs/quick-start [INFO] -- [INFO] BUILD SUCCESS [INFO] -- [INFO] Total time: 8.914 s [INFO] Finished at: 2023-01-05T17:01:08Z [INFO] -- Here you can see that our project now resides in /build/archdocs/quick-start. Note that in this case, the archetype was run from the /build/archdocs directory.

Here you need to supply the target version of both the archetype and the IPF version. At the time of writing, the latest versions are 2022.2.17.172 and 1.5.0 but please check this for your target environment! If you want to learn more about the config options provided and their meaning please check here.

Let's now look at the /build/archdocs/quick-start directory and you'll see:

mytest ls -ltr total 44 drwxrwxr-x 3 bensavage bensavage 4096 Jan 5 17:01 quick-start-app drwxrwxr-x 8 bensavage bensavage 4096 Jan 5 17:01 quick-start-domain drwxrwxr-x 3 bensavage bensavage 4096 Jan 5 17:01 quick-start-e2e -rw-rw-r-- 1 bensavage bensavage 2490 Jan 5 17:01 Readme.md -rw-rw-r-- 1 bensavage bensavage 3893 Jan 5 17:01 pom.xml The key folders to note here are the:

quick-start-app - this is the home for the application runtime code, the code that will be deployed and use the IPF generated domain to process transactions. quick-start-domain - this contains all the artefacts relating to the DSL and flow definitions. quick-start-e2e - this contains all the docker application code Readme.md - this contains information on how to run the generated project,

have a look at that now. Having generated the project, you can build it. Move to the base directory of the project and run:

mvn clean install This will build the project, generating all the artifacts and run both the DSL and application tests. Note that in the application tests, the base implementation uses TestContainers to provide the mongo database. If you're not using docker, you can update this and use your own mongo database.

Let's now run the application in it's containerised form. To do this, open the quick-start-e2e module and try running the E2EComposeLauncher java class. This will spin up a docker environment containing:

An instance of our application code.

An instance of a mongo database.

An instance of the ipf-developer-app, which can be used to quickly view payments.

It should take a minute or so to fully start the environment, you can see when it's ready from the logs:

22-01-2023 17:39:52.889 [main] INFO c.i.t.c.c.runner.env.ComposeLauncher - background compose started successfully Once it's ready, we can fire a dummy payment through by calling the "submit" endpoint of the provided controller. A simple way to do this using curl is:

curl -X POST localhost:8080/submit If successful this should return you the payment identification details:

{ "requestId": "4cece8f7-1a7f-4eb8-a189-a0de5523175e", "uowId": "478a35ce-5e34-4e7b-a58b-060368fab758", "aggregateId": "QuickStartFlow|39437cdb-4612-42a8-b503-8963a211391a" } The key id to consider now is the uowId - this is the unique identifier for our transaction.

Now let's open the Developer Application. Once open, click search and we should see:

developer app From here you can click the 'View' button to start digging into the details of your transaction!

How do you build a project without docker dependencies? When the archetype runs, it's default mode will generate a set of docker scripts that can be used to run the application. These docker scripts provide containers for mongodb. The docker based mongo containers are required in order for the generated tests to work. To stop them from being generated entirely, we simply run the archetype with the "includeTests" config option set to "N".

--- How do you run the archetype in the different modes? To run the archetype, we simply run in one of the two modes as detailed below. This will generate a new IPF project in a directory relative to that the command is run from. The project name will be set to the artifactId provided.

In order to run the application, you'll need to supply the IPF version you want to run against.

Running in interactive mode The application can be run interactively, as follows;

mvn archetype:generate \ -Preleases \ -DarchetypeGroupId=com.iconsolutions.ipf.core.archetype \ -DarchetypeArtifactId=icon-archetype \ -DipfVersion=___IPF_VERSION___ \ -DarchetypeVersion=${project.version} This will then prompt the user to complete the remaining configuration options.

Running in batch (non-interactive) mode mvn archetype:generate -B \ -DarchetypeGroupId=com.iconsolutions.ipf.core.archetype \ -DarchetypeArtifactId=icon-archetype \ -DarchetypeVersion=${project.version} \ -DipfVersion=___IPF_VERSION___ \ -DgroupId=my.test \ -DartifactId=my-test \ -Dversion=1.0.0 \ -DsolutionName=Mytest \ -DmodelName=model \ -DflowName=Myflow \ -DincludeApplication=y -DincludeTest=y

----

--- How do you use your own parent dependency? Projects created by this archetype will use the IPF Bill Of Materials (BOM) as a parent, if you wish to use a custom parent POM, then make the following changes after project creation to ensure that the correct ipf dependencies are included:

<dependencyManagement> <dependencies> <dependency> <groupId>com.iconsolutions.ipf</groupId> <artifactId>ipf-release-bom</artifactId> <version>___IPF_VERSION___</version> <type>pom</type> <scope>import</scope> </dependency> </dependencies> </dependencyManagement>

--- How do you create a reusable flow module? Most IPF projects are traditional applications that contain one or more flows together with the implementation details to make them executable and able to process payments. This is done through the many different types of component available with the Icon DSL.

However, you may wish to build a project that only includes the DSL code, i.e. to create a re-usable pieces of DSL that is not implemented directly within the generated project but is expected to be included within other projects, for example see: Using shared models.

To do this, you simply run the archetype with the 'includeApplication' flag set to "N". This will mean the archetype will only build the project in a reusable component format. Note there's no reason why you can't subsequently turn it into a standard self-contained application project by adding the relevant application modules yourself subsequently.
