ICON IPF Documentation - Business Functions

Complete section documentation with all subsections

Total subsections: 4

Total pages: 31

Table of Contents

Duplicate Check

6 pages in this subsection

Duplicate Check Floclient

Source: https://docs.ipfdev.co.uk/functions/current/ipf-duplicate-check-floclient/home.html

Duplicate Check Floclient

The Duplicate Check Floclient is an IPF component that enables the IPF process flows to check if a message being processed is a functional duplicate. The Floclient provides a domain function that can be included within the IPF processing flows.

A message is considered a functional duplicate when specified data elements within a message match with those in a message received previously and within a defined period of time.

Concepts

Source: https://docs.ipfdev.co.uk/functions/current/ipf-duplicate-check-floclient/concepts.html

Duplicate Check Keys - Single

When invoking `checkSingleDuplicate`, data elements from the message which are used as duplicate check parameters are mapped into a Duplicate Check Key. This key holds these elements as a list of strings and these keys are saved to the supporting transaction cache service.

A message will be considered a duplicate if more than one entry containing the same keys is present in the transaction cache at the time of checking.

```
public class DuplicateCheckKey implements Serializable {
   private List<String> data;
   private String transactionCacheEntryType;
}
```

Duplicate Check Keys - Multiple

To perform multiple duplicate checks in a single invocation (such as performing duplicate checks against all transactions contained within a message) the `checkMultipleDuplicate` function can be called. A duplicate check will be performed for each entry in the map provided and the responses will be collated into a single response which is sent back to the calling flow. If using this functionality, you should provide a Transaction Cache Entry Type in addition to the Duplicate Check Key Map. If one is not specified, a default of `CheckMultipleDuplicate` is used, which is taken from the action's name.

> The results of the multiple duplicate check are collected in a map, so consideration should be given to the expected number of entries in the resulting map, as this is stored against the flow aggregate. If there are more than 500 expected entries, an alternate approach to the multiple duplicate check flo-client might be more appropriate

```
public class DuplicateCheckMultipleRequest implements Serializable {
   String transactionCacheEntryType;
   Map<String, DuplicateCheckKey> duplicateCheckKeyMap;
}
```

Duplicate Check Key Multiple Response

If a multiple duplicate check is requested, results will be collated into a `Map`, which will be returned as business data to the flow. If any duplicates are contained in the result set, the response code returned by the `checkMultipleDuplicate` call will be `CONTAINSDUPLICATES`, otherwise the response code returned will be `NODUPLICATES`

```
public class DuplicateCheckMultipleResponse implements Serializable {
   Map<String, DuplicateResponse> duplicateCheckResponseMap;
}
```

Persistence

The transaction cache is provided by a PersistentTransactionCacheService. Refer transaction cache service. for more details on the transaction cache service.

This service is backed by MongoDB, so the necessary data to populate the cache and perform duplicate checks survives a service restart.

TransactionCacheEntryType

By default, entries associated with single duplicate check invocations will have a TransactionCacheEntryType of `CheckSingleDuplicate` whereas entries associated to the multiple duplicate check will have an entry type of `CheckMultipleDuplicate`, which is taken from the action's name. The transaction cache entry type is used at the time of saving and finding within the cache. Even if the same duplicate check key exists as an entry in the transaction cache, it will only be considered a duplicate if the `transactionCacheEntryType` is the same.

The Client Implementations are able to provide custom transaction cache entry types. This could be useful for when:

- There are two separate flows carrying out duplicate checks and want to restrict duplicate responses just to keys that flow has seen.
- There is a requirement to have different duration periods for different function calls. Custom purging schedules can be setup based on the type.

Refer Providing Custom TransactionCacheEntryTypes for more details on how a custom transaction entry type can be defined.

Duplicate Duration

The length of time a Duplicate Check Key entry exists within the cache effectively determines the period of time a message will be considered a duplicate. Duplicate Check Key entries are routinely purged from the supporting transaction cache service.

By default, purging will take place at midnight every day, and will remove all entries older than midnight. Effectively acting to clear the previous day's transactions. You can provide configuration if this schedule does not meet your needs.

Eager saving

Duplicate Check Keys are eargerly saved to the cache and then verified for any duplicates. If more than one entry is found then at least one previously existed, and the message will be considered a duplicate.

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.

It reduces the window for concurrent duplicates slipping through, at the cost of an extra record being stored.

Supported Message Types

The floclient has the ability to support any message type or message attributes for duplicate checking due to the use of client specific mapping functions for duplicate checking.

The following default mapping functions are provided.

Default Mapping Function Name	ISO 20022 Message Type	Fields mapped to Duplicate Check Key	Description

DuplicateSingleMapFromPain001	pain001	`.pmtInf[0].dbtrAcct.id.othr.id`
		`.pmtInf[0].cdtTrfTxInf[0].pmtId.endToEndId`
		`.pmtInf[0].cdtTrfTxInf[0].amt.instdAmt.value`
		`.pmtInf[0].cdtTrfTxInf[0].amt.instdAmt.ccy`
			Useful for messages containing a single transaction

DuplicateSingleMapFromPacs008	pacs008	`.cdtTrfTxInf[0].pmtId.endToEndId`
		`.cdtTrfTxInf[0].dbtrAcct.id.othr.id`
		`.cdtTrfTxInf[0].intrBkSttlmAmt.value`
		`.cdtTrfTxInf[0].intrBkSttlmAmt.ccy`	

DuplicateMultipleMapFromPain001	pain001	`.pmtInf[0].dbtrAcct.id.othr.id`
		`.pmtInf[0].cdtTrfTxInf[0].pmtId.endToEndId`
		`.pmtInf[0].cdtTrfTxInf[0].amt.instdAmt.value`
		`.pmtInf[0].cdtTrfTxInf[0].amt.instdAmt.ccy`
			Useful for messages which contain many transactions

DuplicateMultipleMapFromPacs008	pacs008	`.cdtTrfTxInf[0].pmtId.endToEndId`
		`.cdtTrfTxInf[0].dbtrAcct.id.othr.id`
		`.cdtTrfTxInf[0].intrBkSttlmAmt.value`
		`.cdtTrfTxInf[0].intrBkSttlmAmt.ccy`	

The Duplicate Check Keys provided by the default mapping functions have no knowledge of the message type. Different messages can share the same field values.

> To avoid misidentification of a duplicate between message types, use different transaction cache entry types. Alternatively, if implementing your own mappers, consider including a value within the key that ensures it is unique between message types.

Getting Started

Source: https://docs.ipfdev.co.uk/functions/current/ipf-duplicate-check-floclient/getting-started.html

Getting Started

Prerequisites

This starter guide assumes you have access to the following:

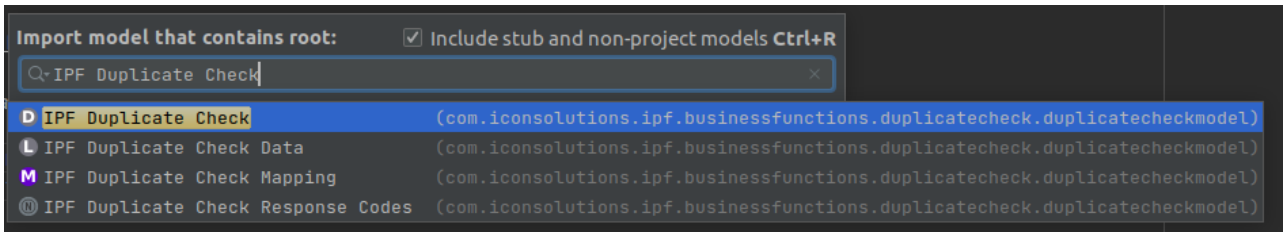
- A flow which is preloaded with the IPF Business Functions.

It demonstrates adding a Single Duplicate Check to a flow, you can also/alternatively add a Multiple Duplicate Check to a flow.

Integrating with a flow

1. Add The duplicate check solution and model to your MPS solution and model

From within the flow you want to add the duplicate check to, press `Ctrl+R` twice. Search for and select the IPF Duplicate Check model from the search bar that opens.

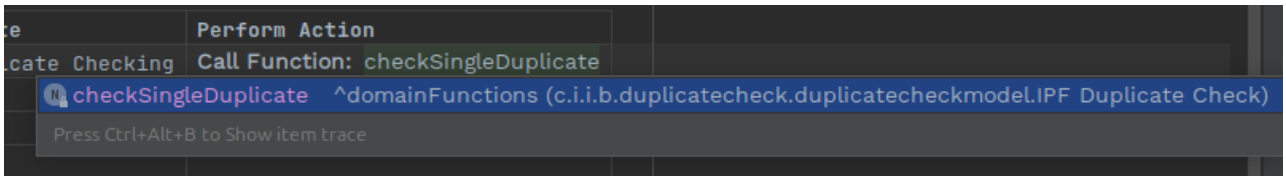


Select OK in the pop up window asking if you would like to import the modules automatically.

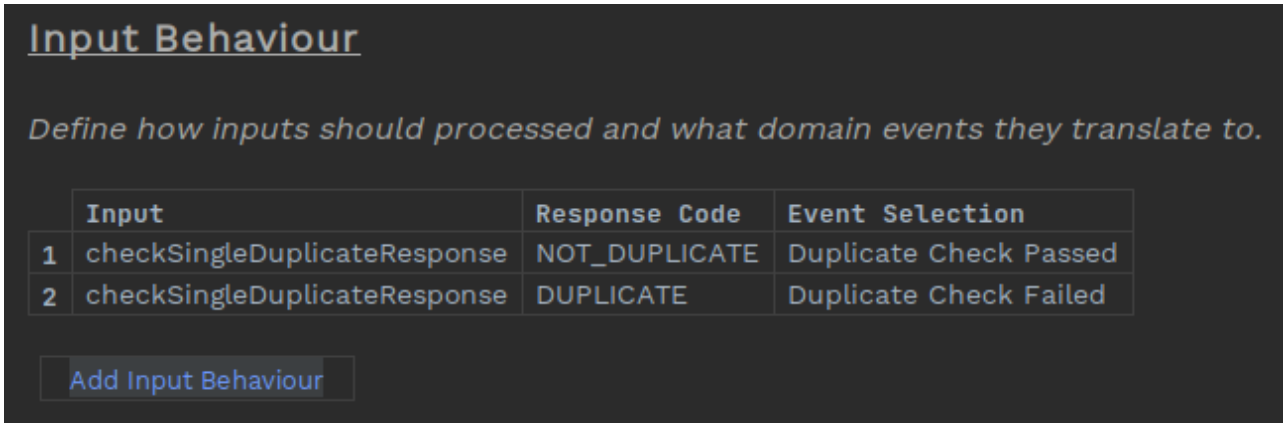
The `checkSingleDuplicate` domain function will now be available to use within your flow.

2. Use the check duplicate function within your flow

Use the `checkSingleDuplicate` function within your flow in the same way as you would for a domain function you have defined yourself.



You will need to define the input behaviour for all `checkSingleDuplicateResponse` response codes and map to your chosen event selection.



You can find more details on using domain functions at DSL 3 - Using a Domain Function. This shows all the associated MPS based changes to fully use a function within your flow.

3. Add the ipf-duplicate-check-floclient-service dependency

Add the ipf-duplicate-check-floclient-service as a dependency to the module that includes the bean for your domain declaration. If you generated your project from the icon archetype this will be named `<your-project-name>-service`

```
<dependency>
   <groupId>com.iconsolutions.ipf.businessfunctions.duplicatecheck</groupId>
   <artifactId>ipf-duplicate-check-floclient-service</artifactId>
```

```
    </dependency>
```

Providing Custom Purging Configuration

The Duplicate Check Floclient uses a TransactionCachePurgingScheduler with the following default configuration for single duplicate checks:

```
ipf.duplicate-check-floclient.single.purging {
  scheduling-specification = "0 0 0 ? * *"
  retain-from-time = "00:00:00"
  retain-from-offset = "0 days"
}
```

Default configuration for multiple duplicate checks:

```
ipf.duplicate-check-floclient.multiple.purging {
  scheduling-specification = "0 15 0 ? * *"
  retain-from-time = "00:00:00"
  retain-from-offset = "0 days"
}
```

Provide your own values in your `ipf-impl.conf` or `application.conf` in order to customise the schedule to your needs. From the above, purging for single transaction cache entries will be done at 12:00 AM each day, and multiple transaction cache entries will be performed at 12:15 AM each day.

You can find more details on providing configuration values for the TransactionCachePurgingScheduler in the transaction cache documentation.

Automated Retries

If the MongoDB instance providing the transaction cache fails to respond, or responds with a failure, no response to the `checkSingleDuplicate` function call will be received by your flow.

You can implement Action Retries within your flow to manage such circumstances. DSL 7 - Handling Timeouts provides an example of their use.

Defining a Custom Single Duplicate Check Key Mapping

Source: https://docs.ipfdev.co.uk/functions/current/ipf-duplicate-check-floclient/defining-a-custom-single-duplicate-check-key-mapping.html

```
```

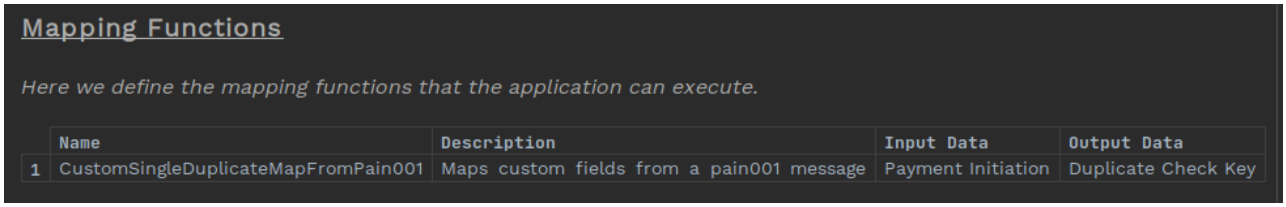
Defining a Custom Single Duplicate Check Key Mapping

This page explains how to define a mapping for a single duplicate check e.g. a message level duplicate check.

1. Add a mapping function

Within your MPS project, add a Mapping Library to your model. Right-click on your model and choose `new` → `v2Flo` → `Mapping Library`.

Add your function to this library and complete its details. The input data represents the message type you wish to duplicate check. The output data must be `Duplicate Check Key`.



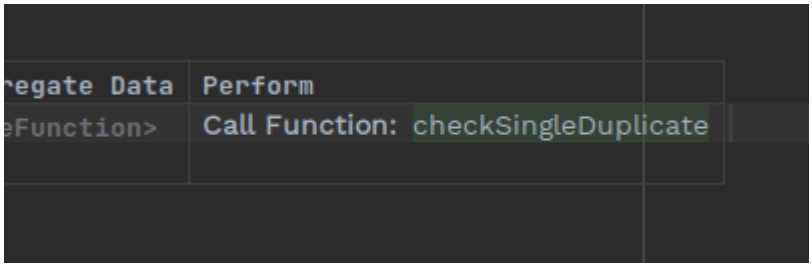
Mapping Functions

Here we define the mapping functions that the application can execute.

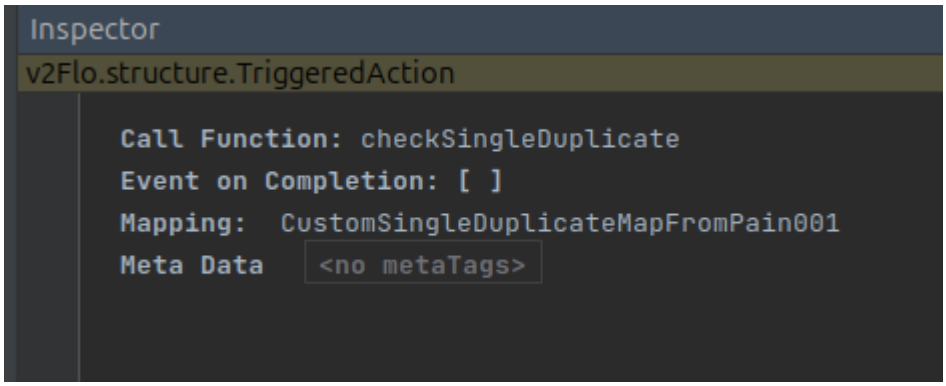
	Name	Description	Input Data	Output Data
1	CustomSingleDuplicateMapFromPain001	Maps custom fields from a pain001 message	Payment Initiation	Duplicate Check Key

2. Use the mapping function within your flow

Within your MPS flow, left-click on the `checkSingleDuplicate` action call you want to use the custom mapping with.



Press `Ctrl+Alt+I` to open the Inspector. Change the `Mapping` in the inspector to your mapping function.



3. Implement your mapping adapter

You need to provide a java implementation of your mapping function and provide it into your domain declaration as an adapter.

For example:

```
public class CustomDuplicateCheckMappingAdapter implements CustomDuplicateCheckMappingMappingPort {

    @Override
    public CustomSingleDuplicateMapFromPain001MappingOutput
performCustomSingleDuplicateMapFromPain001(CustomSingleDuplicateMapFromPain001MappingParameters inputParameters) {
        return new
CustomSingleDuplicateMapFromPain001MappingOutput(DuplicateCheckKey.builder().data(getDuplicateCheckFields(inputParameters.getPaymentIni

    }

    private List<String> getDuplicateCheckFields(CustomerCreditTransferInitiationV09 pain001) {
        List<String> data = new ArrayList<>();
        // your code here to populate data with the fields you want
        return data;
```

```
    }
}
```

```
     @Bean
    public DuplicatecheckexampleDomain duplicatecheckexampleDomain(ActorSystem actorSystem,
                                                                   CustomDuplicateCheckMappingMappingPort
customDuplicateCheckMappingAdapter,
                                                                   Dispatcher floDispatcher) {
        return new DuplicatecheckexampleDomain.Builder(actorSystem)
                .withCustomDuplicateCheckMappingMappingAdapter(customDuplicateCheckMappingAdapter)
                .withDispatcher(floDispatcher)
                .build();
    }
```

You can find more details on using mapping functions at DSL 6 - Mapping Functions.

Defining a Custom Multiple Duplicate Check Key Mapping

Source: https://docs.ipfdev.co.uk/functions/current/ipf-duplicate-check-floclient/defining-a-custom-multiple-duplicate-check-key-mapping.html

```
```

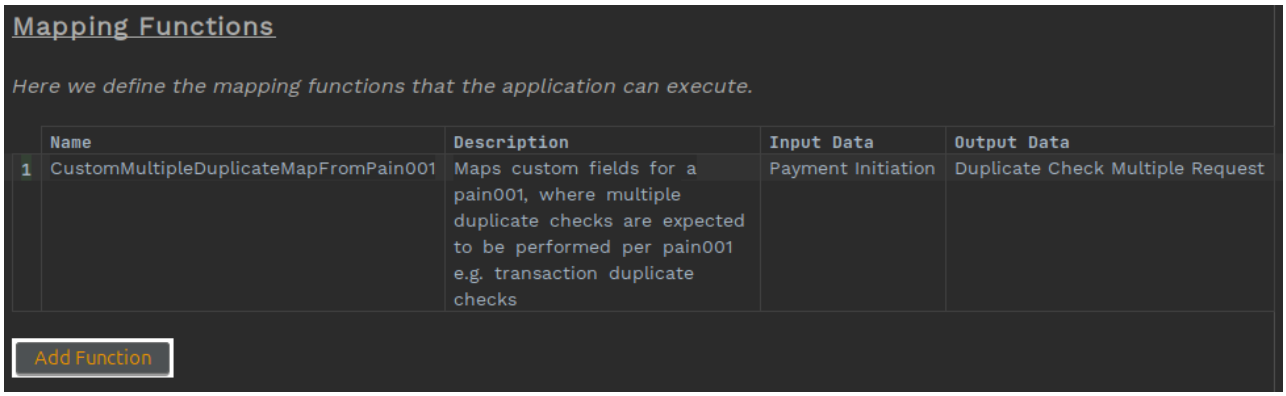
Defining a Custom Multiple Duplicate Check Key Mapping

This page explains how to define a mapping for a multiple duplicate check e.g. where you want to perform duplicate checks on a message which contains multiple transactions.

1. Add a mapping function

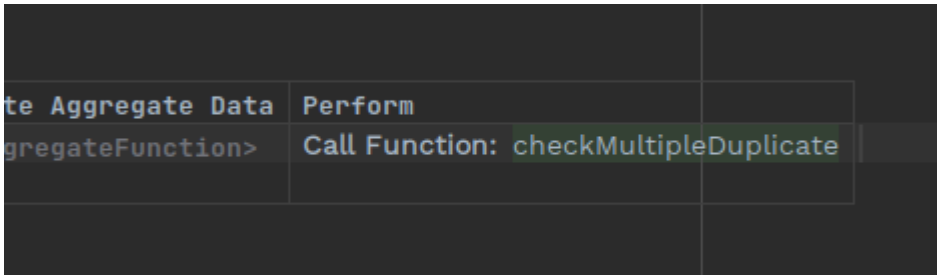
Within your MPS project, add a Mapping Library to your model. Right-click on your model and choose `new` → `v2Flo` → `Mapping Library`.

The output data must be Duplicate Check Multiple Request. Which should specify the `transactionCacheEntryType` and a map of entries for which you would like to perform duplicate checks.

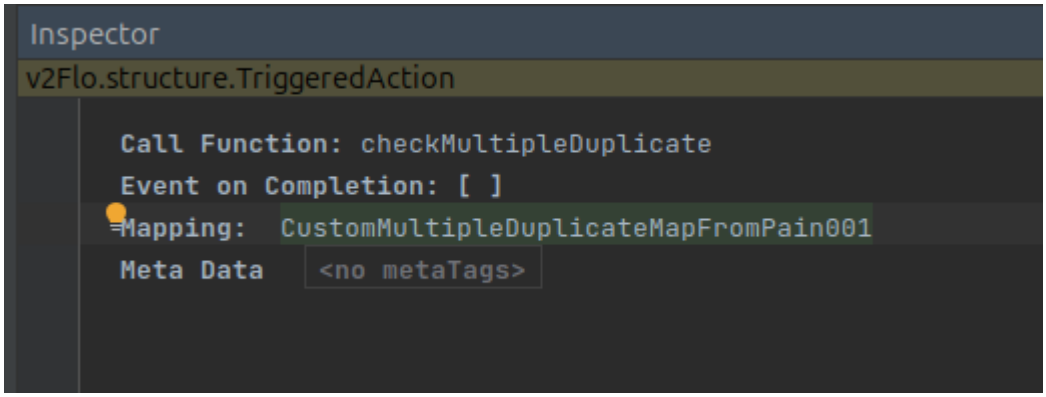


2. Use the mapping function within your flow

Within your MPS flow, left-click on the `checkMultipleDuplicate` action call you want to use the custom mapping with.



Press `Ctrl+Alt+I` to open the Inspector. Change the `Mapping` in the inspector to your mapping function.



3. Implement your mapping adapter

You need to provide a java implementation of your mapping function and provide it into your domain declaration as an adapter.

For example:

```
public class CustomDuplicateCheckMappingAdapterMultiple implements CustomDuplicateCheckMappingMappingPort {

    @Override
```

```
    public CustomMultipleDuplicateMapFromPain001MappingOutput
performCustomMultipleDuplicateMapFromPain001(CustomMultipleDuplicateMapFromPain001MappingParameters inputParameters) {
        // Logic to build a map of Duplicate Check Keys here
        var txnCounter = new AtomicInteger(0);
        Map<String, DuplicateCheckKey> duplicateCheckKeyMap = Optional.ofNullable(inputParameters.getPaymentInitiation())
                .map(CustomerCreditTransferInitiationV09::getPmtInf)
                .map(instruction -> instruction.stream()
                        .flatMap(CustomDuplicateCheckMappingAdapter::createCustomDuplicateCheckKeyForPain001Transaction)
                        // Using a linked hashmap to preserve order
                        .collect(Collectors.toMap(__ -> createIdForMultipleDuplicateCheck(inputParameters.getId(), txnCounter),
Function.identity(), (x, y) -> y, LinkedHashMap::new))))
                .orElseThrow(() -> new IconRuntimeException("No payment initiation found"));
        return new CustomMultipleDuplicateMapFromPain001MappingOutput(DuplicateCheckMultipleRequest.builder()
                .transactionCacheEntryType("PAIN_001_MULTIPLE_CUSTOM")
                .duplicateCheckKeyMap(duplicateCheckKeyMap)
                .build());
    }
}
```

The above, iterates through the transactions of the provided pain.001 message and builds a map of `DuplicateCheckKey` entries.

This map is wrapped in a Duplicate Check Key Multiple Request serves as input to the `checkMultipleDuplicate` function which will process each entry and perform a duplicate check, collating the results into a Duplicate Check Key Multiple Response. In addition to the map of Duplicate Check Keys, you can also specify the `transactionCacheEntryType` in the request (set to `CheckMultipleDuplicatePain001` above) - the action name will be used as a default if none is specified.

```
     @Bean
    public DuplicatecheckexampleDomain duplicatecheckexampleDomain(ActorSystem actorSystem,
                                                                   CustomDuplicateCheckMappingMappingPort
customDuplicateCheckMappingAdapter,
                                                                   Dispatcher floDispatcher) {
        return new DuplicatecheckexampleDomain.Builder(actorSystem)
                .withCustomDuplicateCheckMappingMappingAdapter(customDuplicateCheckMappingAdapter)
                .withDispatcher(floDispatcher)
                .build();
    }
```

You can find more details on using mapping functions at DSL 6 - Mapping Functions.

4. Handling multiple duplicate check responses

When handling `CONTAINS_DUPLICATES` response code in your flow, you should add the `Duplicate Check Multiple Response` business data element to the associated event as shown below:

Event Definitions

Define the events that are specific to this flow

	Name	Description	Business Data
1	Duplicate Check Passed	Duplicate Check Passed	<no business data>
2	Duplicate Check Failed	Duplicate Check Failed	Duplicate Check Multiple Response

Add Event

Aggregate Functions

Define the functions that should be applied on the aggregate after an event has been received

Add Aggregate Function

Input Behaviour

Define how inputs should processed and what domain events they translate to.

	Input	Response Code	Event Selection
1	checkMultipleDuplicateResponse	NO_DUPLICATES	Duplicate Check Passed
2	checkMultipleDuplicateResponse	CONTAINS_DUPLICATES	Duplicate Check Failed
3	checkMultipleDuplicateResponse	ERROR	Duplicate Check Failed

Add Input Behaviour

This will make the results available to downstream processing in your flow.

An example `Duplicate Check Failed` is shown below with the result map shown. The order of the entries is preserved from the source message. In the example below, the order in the response map is the same as the order of the transactions in the source message. The key can be used in the `Duplicate Check Multiple Request` map to retrieve the original duplicate key used to perform the duplicate check.

Viewing Domain Event


```
{
  "createdAt": "2024-11-27T19:35:27.502Z",
  "originalCommandId": "HandleCheckMultipleDuplicateResponse|f9ba7dd8-2a1c-41db-a549-89d803dd6c88|d8da8d6d-302e-4b90-8d70-85962b246
  "status": {
    "originatingStatus": "Checking Duplicate",
    "resultingStatus": "Rejected",
    "globalStatus": "REJECTED"
  },
  "eventId": "Pacs008MultipleDuplicateCheck|0be6cfec-1cc4-4cb5-b755-d778bdf04e3c|2",
  "processingContext": {
    "associationId": "Pacs008MultipleDuplicateCheck|0be6cfec-1cc4-4cb5-b755-d778bdf04e3c",
    "checkpoint": "PROCESS_FLOW_EVENT|Pacs008MultipleDuplicateCheck|0be6cfec-1cc4-4cb5-b755-d778bdf04e3c|1",
    "unitOfWorkId": "c316ac14-6e97-4025-ac80-afe043a90cc8",
    "clientRequestId": "9b66f4a1-9613-4656-be46-55add22b9ad9",
    "processingEntity": "Transaction Duplicate Check Flow"
  },
  "initiatingId": "0be6cfec-1cc4-4cb5-b755-d778bdf04e3c",
  "responseCode": "CONTAINSDUPLICATES",
  "hash": "-299209474",
  "failureResponse": true,
  "duplicateCheckMultipleResponse": {
    "duplicateCheckResponseMap": {
      "Pacs008MultipleDuplicateCheck|0be6cfec-1cc4-4cb5-b755-d778bdf04e3c|0": "NOT_DUPLICATE",
      "Pacs008MultipleDuplicateCheck|0be6cfec-1cc4-4cb5-b755-d778bdf04e3c|1": "DUPLICATE"
    }
  }
}
```

Providing Custom TransactionCacheEntryTypes

Source: https://docs.ipfdev.co.uk/functions/current/ipf-duplicate-check-floclient/providing-custom-transaction-cache-entry-types.html

Providing Custom TransactionCacheEntryTypes

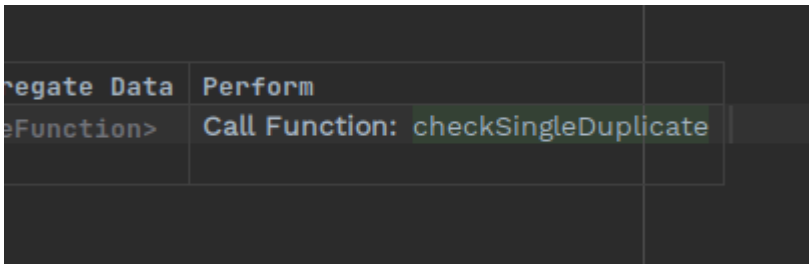
For a single duplicate check there are two ways to provide a `transactionCacheEntryType`, you can set the value using the metadata tag or set the field in your custom mapping function. The `transactionCacheEntryType` is set according to the following precedence for single duplicate checks:

1. Value set via mapping function
2. Value set via meta data tag
3. Otherwise, if 1 or 2 have not been set then the `transactionCacheEntryType` is defaulted to action name e.g. `CheckSingleDuplicatePacs008`

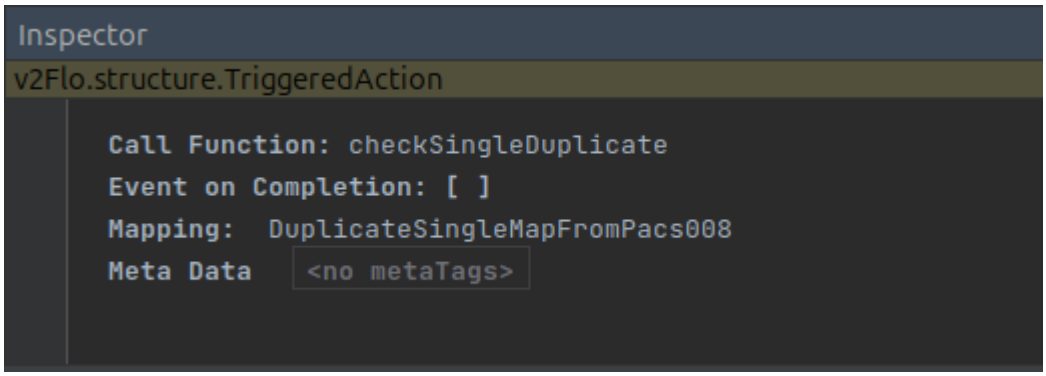
For multiple duplicate checks, the `transactionCacheEntryType` can only be set in the multiple duplicate mapping function. Additional details on how to set the `tranactionCacheEntryType` using the metadata tag are provided below.

Meta Data Tag

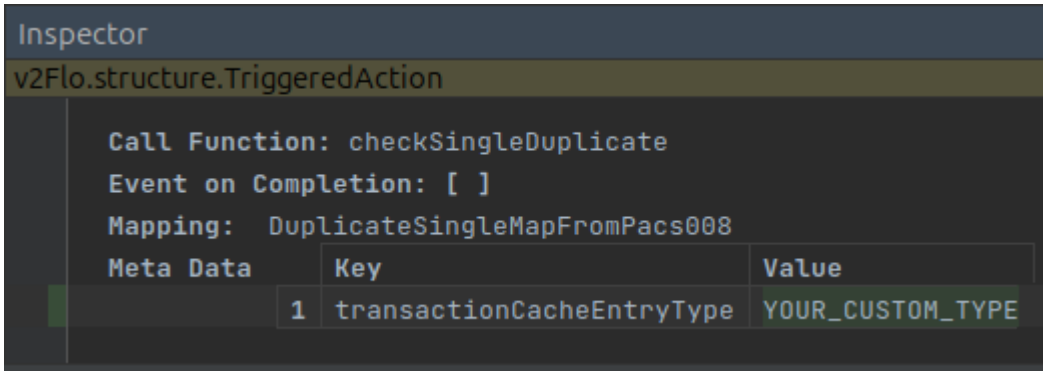
Within your MPS flow, left-click on the checkSingleDuplicate action call you want to use the custom mapping with.



Press `Ctrl+Alt+I` to open the Inspector. Left-click on `<no metaTags>` next to `Meta Data`.



Press `Enter` to create a new table. For the key enter `transactionCacheEntryType`. For the value, provide the string you want as your custom transaction cache entry type.



Add a suitable purging mechanism for your type

Your custom type will not be covered by the default purger provided by the flo-client. You will want to implement a purging mechanism so that keys are not considered duplicates forever. You are free to implement a mechanism of your choice.

Some options you may want to use:

- Implement your own TransactionCachePurgingScheduler bean as per the instructions in the transaction cache docs.
- If this is the only type within the transaction cache, you can implement a MongoDB TTL index on the creationDate.

\n

Human Task Manager

13 pages in this subsection

Human Task Manager Floclient

Source: https://docs.ipfdev.co.uk/functions/current/htm-floclient/home.html

Human Task Manager Floclient

IPF's HTM Floclient provides a reusable DSL component that aims to make integration with the Human Task Manager simpler. This page introduces the core concepts used in the floclient, enabling clients to quickly introduce human tasks into their flows.

The IPF Human Task Manager

The IPF Human Task Manager provides the ability to manage tasks that cannot be fulfilled automatically and requires some form of human interaction. This is typically used for exception type processing.

Features

The htm floclient provides a number of out-of-the-box features to support integrating with the Human Task Manager application from within a payment flow.

- HTM Requests
- HTM Processor
- Meta Data Values

Features

Source: https://docs.ipfdev.co.uk/functions/current/htm-floclient/features/features.html

Features

The HTM Floclient provides a number of features to support interaction with the HTM Services.

The following pages describe the features provided by the service.

- HTM Requests
- HTM Processor
- Meta Data Values

HTM Requests

Source: https://docs.ipfdev.co.uk/functions/current/htm-floclient/features/htmrequests.html

HTM Requests

An 'HTM Request' is the DSL's representation of a single HTM task definition. It provides the ability to define all the aspects of an HTM task request so that it can then be invoked from within any payment flow.

The HTM Request will automatically create two responses that need to be handled. These are:

- <RequestName> Response - this is the main response from the HTM Server and details the outcome of the task.
- <RequestName> Technical Response - this is the technical ack that comes back from the HTM Server and defines whether the task registration was successful.

The HTM request holds a number of properties that can be used to setup the definition of the task:

- Name - this will become the 'task type' within the HTM definition.
- Description - used for informational purposes only and not sent to HTM itself.
- Priority - this integer value is sent to HTM as the priority value of the task.
- Required Approvals - this integer value is sent to HTM to specify how many approvals are required for the task.
- Ignore Tech Response - when an HTM task is created within HTM, an acceptance response is returned containing the task id. By default, this is returned to the flow as a "Technical Response" to the request. If ignore tech response is enabled, the technical response will be logged only and not sent to the flow for processing.
- Response Codes - these define the different options that the user will be able to select in HTM for an outcome of the task.
- Business Data - this specifies the business data that is made available to the HTM service. If a single business data element is defined, then it is sent as the primary element in HTM. If multiple elements are defined then it is necessary to configure a mapping to the HTM input data type.
- Meta Data - this specifies static meta tags that are applicable to all the requests of this type. It is typically used to define known groups, for example that a request is related to "Sanctions".

Each HTM request is stored within an 'HTM Request Library'. The library in this case is simply a grouping container for logically displaying HTM Requests and has in itself no functional impact.

HTM Processor

Source: https://docs.ipfdev.co.uk/functions/current/htm-floclient/features/htmprocessor.html

HTM Processor

The HTM floclient allows for custom configuration of each property that can be sent to the HTM service. This is done through the 'HTM Processor' interface. It is possible to register a processor for each individual task type.

Default HTM Processor

The default HTM processor is used as a fallback for any task type where a custom processor has not been provided. It has the following properties:

Property	Default Implementation
Idempotency	Checkpoint + "_" + HTM Request Name
Due Date	Current Time + 1 Day
Supporting Data	If no business elements provided, this is empty. If 1 business element provided, this is used as the payload. If 2 or more business elements are provided, this throws an exception.
Meta Tags	Returns the static data list only.
SupportedTypes	Returns the generic fallback entry. == Idempotency The HTM application requires a unique key to be passed that uniquely identifies a given task. This is used to ensure that it is not possible to create multiple identical tasks (through retries etc). The Floclient application default uses a combination of the current checkpoint in the flow and the task type to generate this key.

Meta Data Values

Source: https://docs.ipfdev.co.uk/functions/current/htm-floclient/features/metatags.html

Meta Data Values

The HTM application supports the provision of 'MetaDataTags' to provide enriched information about a particular task. This information can then be used in activities within HTM, such as searching for given types.

There are two methods of setting meta data tags within the flo-client.

Static Meta Tags

If the tag is not dependent on the payment’s runtime details, then it is possible to specify it as a static entry.

Runtime Meta Tags

It is also possible to specify runtime tags. This can be used for example if you want to group tasks together by currency. To specify a runtime tag, it is necessary to inject an instance of the 'HtmProcessor' instance into the spring context that supports your task type. These tags are provided by code implementation.

Error Handling

Source: https://docs.ipfdev.co.uk/functions/current/htm-floclient/features/error-handler.html

Error Handling

The HTM floclient also offers an implementation of the flo-lang 'FlowErrorExtensions' interface. This extension provides the ability to hook errors that occur in flow processing into HTM so that users can handle them.

The extension provides the following capability when activated:

- When unexpected errors occur during flow processing, the flow will move to the 'In Error' state and a new HTM task of type 'ERROR_PROCESS' is raised.
- The new task will be available in HTM and have three possible outcomes:
    - ***Abort*** - this will abort the flow.
    - ***Force Complete*** - this will force the flow to the special 'Force Completed' state.
    - ***Resume*** - this will tell the flow to attempt to retry the previous actions; this is particularly useful, for example, if the issue has been resolved and a retry is the logical next step.

Details on how to use the extension are provided here.

Getting Started with Human Task Manager (HTM)

Source: https://docs.ipfdev.co.uk/functions/current/htm-floclient/htm-getting-started.html

Getting Started with Human Task Manager (HTM)

Pre-Requisites

This starter guide to using Human Task Manager assumes you have access to the following:

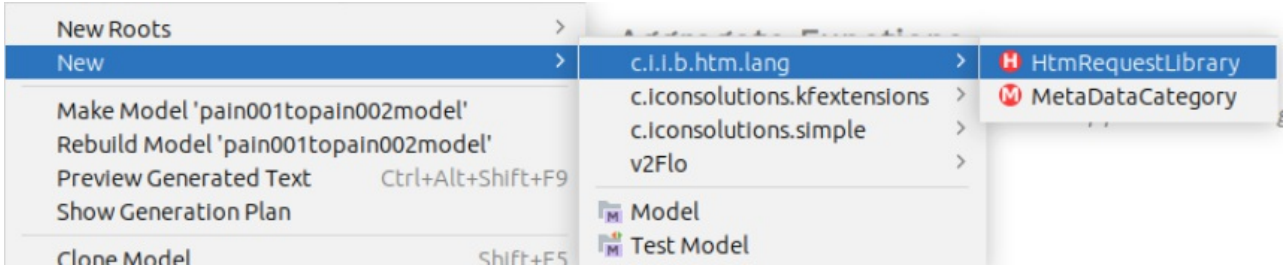
- An HTM Server - this is available as a docker image from Icon.
- The Operational Dashboard - this is available as a docker image from Icon.
- A flow which is pre-loaded with the IPF Business Functions. You can find more details about how to add business functions to your flow here.

Integrating with a flow

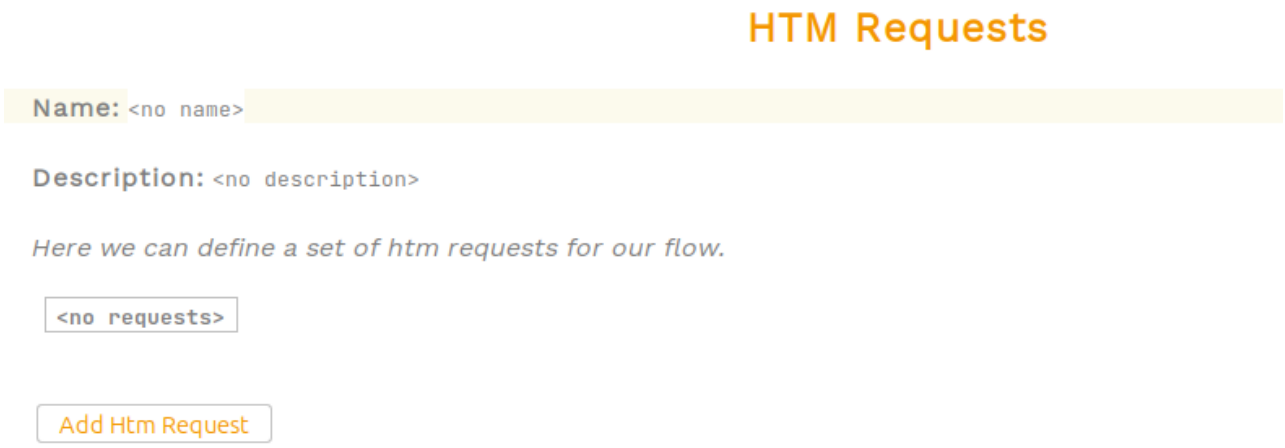
The HTM business function is designed to allow easy interaction between an IPF process flow and the HTM application. It provides the capability to define the core characteristics of a task within the DSL such that tasks can be created and bespoke response codes returned from any point in your flow.

Defining an HTM Task

To define a task for HTM, we need to define a 'HTM Request' within our flow. To do this we use the bespoke HTM language - this works just like the core language except that it will appear in the folder for that language:



The core component here is the 'HTM Request Library'. It behaves in a similar way to the external domain component as creating one will provide both the request and responses that you need to interact with the HTM system. Let's create a new library now nad you should see:



Now we can define the name and give a description to our library. Then we can add a new HTM Request - each HTM request represents a different task type that we want to send to the HTM Server. There are a number of properties to provide in defining the request:

- Name - this will be provided to HTM as the 'task type'.
- Description - informational only.
- Priority - this will be provided to HTM as the 'task priority'. It is an integer value.
- Required Approvals - this will be provided to HTM as the |required approvals for the task|.
- Ignore Tech Response - this allows the flow to hook into the HTTP response of the initial task creation request. If this is not required, it can be ignored and only the final completion result will be returned to the flow.
- Response Codes - this is the result outcome that will be available within the HTM application.

- Business Data - this is the business data that will be packaged and sent to HTM.
- MetaData Tags - these are custom, optional tags which can be attached to the HTM Request.

	Name	Description	Priority	Required Approvals	Ignore Tech Response	Response Codes	BusinessData
1	HTM Request One	Testing	1	0	[x]	AcceptOrReject	Customer Credit Transfer
2	HTM Request Two	Testing	2	1	[ ]	HTM Bespoke Codes	Customer Credit Transfer

Using an HTM Task

An HTM Task is used exactly as a more traditional external domain request / response pair. The task is created simply by calling the HTM Request as an action at any point in the flow.

	With Current States	When	For Event	Move to State	Perform Action
1	In HTM One	On	Request One Accepted	In HTM Two Pending	**Perform Action :** HTM Request Two

Then we can just use the resulting matching response input.

	Input	Response Code	Perform Enrichment	Event Selection
1	HTM Request One Response	Accepted	<no mappingFunction>	Request One Accepted
2	HTM Request One Response	Rejected	<no mappingFunction>	Request One Rejected
3	HTM Request Two Response	HTM Outcome One	<no mappingFunction>	Request Two Outcome One
4	HTM Request Two Response	HTM Outcome Two	<no mappingFunction>	Request Two Outcome Two
5	HTM Request Two Technical Response	<Any Response Code>	<no mappingFunction>	Technical Acceptance

Providing an implementation

Once we've set up our flow integration, we need to provide the implementation. We do this simply by adding the following dependency:

```
<dependency>
  <groupId>com.iconsolutions.ipf.businessfunctions.htm</groupId>
  <artifactId>ipf-human-task-manager-floclient-service</artifactId>
</dependency>
```

From a configuration standpoint, we simply need to specify where our HTM server implementation is running. This is done by setting the following properties:

```
ipf.htm.request-reply.starter {
  http.client {
    host = "localhost"
    port = 8083
  }
  register-task.enabled = true
  cancel-task.enabled = false
}
```

How do I use the floclient at runtime?

Source: https://docs.ipfdev.co.uk/functions/current/htm-floclient/guides/adddependency.html

How do I use the floclient at runtime?

Once a task has been added to your flow, in order to use the floclient you need to ensure the implementation dependency has been added to your project.

This dependency is:

```
<dependency>
   <groupId>com.iconsolutions.ipf.businessfunctions.htm</groupId>
   <artifactId>ipf-human-task-manager-floclient-service</artifactId>
</dependency>
```

> Failure to include this dependency will result in the following error when building the domain structures within your application:
>
> ```
> Application validation errors occurred during startup:
>    An action processor has not been provided for external domain IPF_DEBULKER_DOMAIN in model IPFDEBULKMODEL
>    A decision processor has not been provided for library IPF_DEBULKER_DECISIONS in model IPFDEBULKMODEL
>    A mapping processor has not been provided for library IPF_DEBULKER_MAPPING_FUNCTIONS in model IPFDEBULKMODEL
> ```

How do I provide a custom htm processor?

Source: https://docs.ipfdev.co.uk/functions/current/htm-floclient/guides/customhtmprocessor.html

How do I provide a custom htm processor?

As discussed in HTM Processor, a set of default behaviours are provided for all task types. If you require to change these, then you simply need to create a new instance of the HTM processor and inject it into the spring context.

It's important to note that when doing this, it's necessary to implement the "supportedTypes" method and provide a list of all the HTM Requests that you wish this processor to be used for instead of the default one. For example, if you want to override the behaviour for 'HTM Request One' you'd add:

```
 @Override
public List<String> supportedTypes() {
    return List.of("HTM Request One");
}
```

> A custom HTM processor can support multiple different HTM requests; simply add them to the list!

How do I use meta tags using the payment data?

Source: https://docs.ipfdev.co.uk/functions/current/htm-floclient/guides/runtimemetatags.html

How do I use meta tags using the payment data?

It's possible to define meta tags using the runtime data through provision of a custom HTM Processor and then overriding the 'buildMetaDataTags' method.

This method provides all the business data that has been passed to the HTM request and returns a list of the meta data tags that are required. For example, creating a meta tag for currency of a pacs008 may look like:

```
 @Override
public List<MetaDataTag> buildMetaDataTags(List<MetaDataTag> staticTags, Map<String, Object> businessDataElements) {
    var metaTags = new ArrayList<>(staticTags);
    metaTags.add(MetaDataTag.builder()
            .category("Currency")

.value(((FIToFICustomerCreditTransferV08)businessDataElements.get("CustomerCreditTransfer")).getCdtTrfTxInf().get(0).getIntrBkSttlmAmt()

    return metaTags;
}
```

How do I ensure that my task is processed by the right application?

Source: https://docs.ipfdev.co.uk/functions/current/htm-floclient/guides/filtering.html

How do I ensure that my task is processed by the right application?

When a HTM task is completed, a notification is published to the completion Kafka topic. Depending on how your application is deployed, you may need to consider filtering tasks to ensure that the appropriate response is only processed once by the appropriate system.

Filtering by flow

By default, an application using the HTM Flo Client will process all tasks where it has an instance of the target flow available. This ensures that responses are only processed by those flows that are expected to process them - other HTM responses are ignored.

This uses the default property configuration:

```
ipf.htm.receiver-type = default
```

Filtering by flow & running instance

When a flow is deployed in multiple places (for example a common reusable flow that is deployed alongside it's main calling flow) it may be necessary to further filter the flow from the default behaviour. This is because although the default flow filtering will match, the actual instance of the flow will potentially be on a different deployment. To do that, we can switch the filtering modes by setting the property:

```
ipf.htm.receiver-type = receive-when-non-initial
```

Providing a custom implementation

It is also possible to provide a completely custom implementation of the handling of HTM completion responses. To do this we set the receiver type to custom:

```
ipf.htm.receiver-type = custom
```

When using this option, it is also then necessary to provide a custom implemenation of the 'HtmClientReceiveNotificationPort' that is provided by the core HTM API.

> ## How Do I Use the Pre-Built HTM Error Handler for Flo-Lang?

Source: https://docs.ipfdev.co.uk/functions/current/htm-floclient/guides/use-errors.html

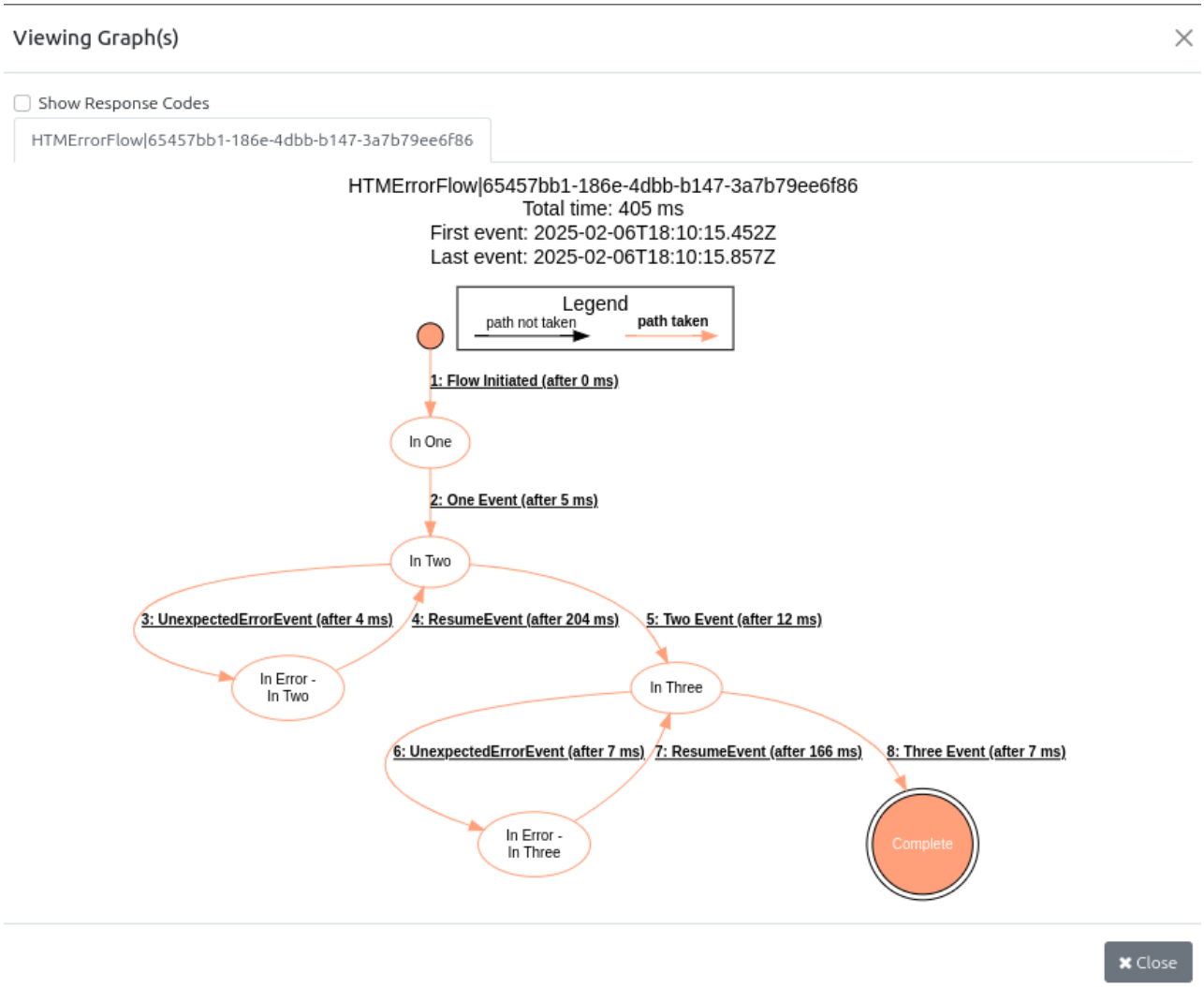
```
```

How Do I Use the Pre-Built HTM Error Handler for Flo-Lang?

The HTM Error handler can be used by simply importing the standard HTM floclient dependency, and then registering the error handler with your domain. For example:

```
 @Bean
public HtmexampleDomain htmexampleDomain(ActorSystem actorSystem, Dispatcher floDispatcher,
                                         HtmErrorHandlerExtensions<Aggregate> htmErrorHandlerExtension) {
    // All adapters should be added to the domain model
    return new HtmexampleDomain.Builder(actorSystem)
            .withDispatcher(floDispatcher)
            .withFallbackExtensionProvider(ExtensionProvider.builder().flowErrorExtensions(htmErrorHandlerExtension).build())
(1)
            .build();
}
```

1	Here we can see that we've told our domain configuration to use the "HtmErrorHandlerExtension" as the flow error extension for the whole model domain. That's everything you need to do! Now, any unexpected error that occurs during flow processing will activate error handling and process the error accordingly. For example, if we were to resume the flow, and that fixed the problem, we'd see something like this in our graphs:



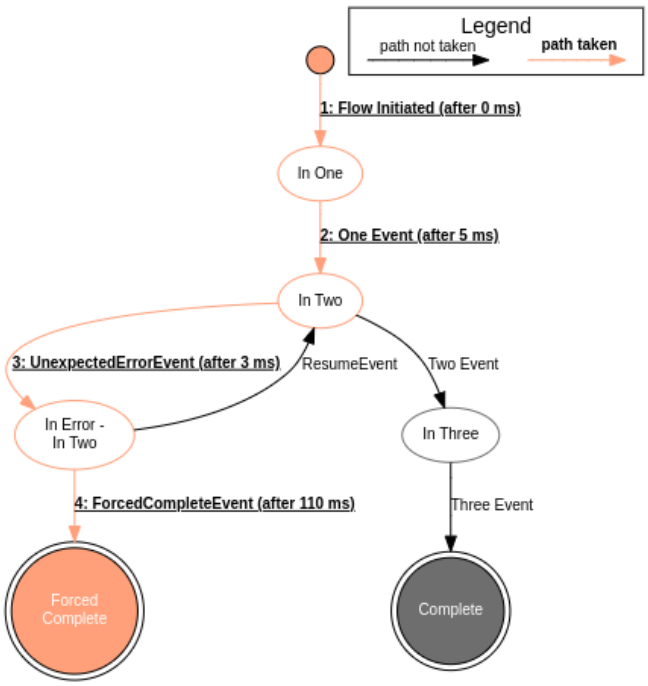
Here we can see the routing of the flow to the error state and the resumption and eventual completion of the flow. Alternatively, you may choose to force complete the flow instead:

Viewing Graph(s)

Show Response Codes

HTMErrorFlow|85534ca5-cb31-4b10-be2d-52721d6b3b44

HTMErrorFlow|85534ca5-cb31-4b10-be2d-52721d6b3b44
Total time: 118 ms
First event: 2025-02-06T18:10:15.099Z
Last event: 2025-02-06T18:10:15.217Z



Close

How Do I Modify HTM Request from HTM Error Handler for Flo-Lang?

Source: https://docs.ipfdev.co.uk/functions/current/htm-floclient/guides/modify-htm-request-error-handler.html

How Do I Modify HTM Request from HTM Error Handler for Flo-Lang?

The custom HTM Error handler, based on default implementation can be created for sending additional data (supporting data and parameters) to HTM. For example:

```
 public class CustomHtmErrorHandlerExtensions<T extends Aggregate> extends HtmErrorHandlerExtensions<T> {

    public CustomHtmErrorHandlerExtensions(SendingConnector<RegisterTaskRequest, Response<RegisterTaskResponse>> taskConnector,
HtmProcessorRegistry htmProcessorRegistry) {
        super(taskConnector, htmProcessorRegistry);
    }

    @Override
    protected ValueWrapper getSupportingData(T aggregate, Throwable t) {
        return ValueWrapper.builder().build(); // custom implementation
    }

    @Override
    protected ValueWrapper getParameters(T aggregate, Throwable t) {
        return ValueWrapper.builder().build(); // custom implementation
    }
}
```

Spring configuration example:

```
 @Bean
public HtmErrorHandlerExtensions<Aggregate> defaultHtmErrorExtensions(SendingConnector<RegisterTaskRequest,
Response<RegisterTaskResponse>> taskConnector, HtmProcessorRegistry htmProcessorRegistry) {
    return new CustomHtmErrorHandlerExtensions<>(taskConnector, htmProcessorRegistry);
}

@Bean
public HtmexampleDomain htmexampleDomain(ActorSystem actorSystem, Dispatcher floDispatcher,
                                         HtmErrorHandlerExtensions<Aggregate> htmErrorHandlerExtension) {
    // All adapters should be added to the domain model
    return new HtmexampleDomain.Builder(actorSystem)
            .withDispatcher(floDispatcher)
            .withFallbackExtensionProvider(ExtensionProvider.builder().flowErrorExtensions(htmErrorHandlerExtension).build())
(1)
            .build();
}
```

\n

Debulker

11 pages in this subsection

Debulker Floclient

Source: https://docs.ipfdev.co.uk/functions/current/debulker-floclient/home.html

Debulker Floclient

IPF's Debulker Floclient provides a reusable DSL component that aims to make integration with the IPF Debulker simpler. This page introduces the core concepts used in the floclient, enabling clients to quickly introduce bulk based flows.

The IPF Debulker

The IPF Debulker provides the ability to process large files which contain multiple messages and transactions. Full details of it's capabilities can be found here.

Features

The debulker floclient implements a number of features out of the box to provide support integrating with the debulker.

- Business Data Elements
- Call Bulk Flows
- The IPF Debulker Domain
- Debulker Reason Codes
- Threshold Checks

Features

Source: https://docs.ipfdev.co.uk/functions/current/debulker-floclient/features/features.html

Features

The Debulker floclient provides a number of features to support debulking of files.

The following pages describe the features provided by the service.

- Business Data Elements
- Call Bulk Flows
- The IPF Debulker Domain
- Debulker Reason Codes
- Threshold Checks

Business Data Elements

Source: https://docs.ipfdev.co.uk/functions/current/debulker-floclient/features/businessdataelements.html

Business Data Elements

The debulker floclient provides a number of default business data elements that can be used within the flow:

	Name	Description	Data Type
1	Debulk Identifier	A collection of identifiers related to the bulk	DebulkerIdentifier
2	Debulk Counter	The counter of number of returned results	DebulkerRecordCounter
3	Debulk Component Data	Data Received from the component store	String
4	Debulk Config Name	The config name for debulker	String
5	Debulk ID	The ID of the debulk flow	String
6	Debulk Source	The Source of the debulk	DebulkerFileSource
7	Debulk Default Marker	<no description>	String

These are explained in more detail in the sections below:

Data Elements Used Throughout Processing

These elements are used and updated by the processing of the debulking file:

The Debulker Identifier

This element contains all the information needed to uniquely identify the current flow within the bulking output. It contains key information such as the bulkId, componentId and the parent call information.

The Debulk Counter

This element is used by the Threshold Checks to track the outcome of records as they are received.

The Debulk Component Data

This element is returned by calls to the component store. It contains the string representation of the data retreived.

Data Elements Used In Initiation

These elements are used in order to initiate a new debulk.

Debulk ID

This is the unique ID which will be assigned to the debulking process.

Debulk Source

This tells the debulker where to find the source file information.

Debulk Config Name

This tells the debulker which config set up to use when initiating the debulking of the file.

Debulk Default Marker

This tells the debulker what marker to use when none is present.

Call Bulk Flows

Source: https://docs.ipfdev.co.uk/functions/current/debulker-floclient/features/callbulkflow.html

Call Bulk Flows

As part of the standard IPF DSL it is possible to call other flows using the 'Call Flow' action. With the debulker floclient, we now introduce the 'Call Bulk Flow' capability.

Action Behaviour

When using 'Call Bulk Flow' it is necessary to provide two arguments:

1. The flow which is being targeted
2. The 'Marker' within the debulked data to be used.

When invoked therefore, the flow will call the debulker and ask for all records for the current bulkId and the given marker. For each of these records it will then initiate a new instance of the targeted flow, providing the data retrieved from the component store to the child flow.

Data Transfer

The call bulk flow capability expects the 'Debulk Identifier' data element to be available to it. It uses this in order to determine the bulk Id that should be processed at runtime.

When a child flow is called, it will be sent two key data fields:

1. The Debulk Identifier
2. The Debulk Component Data

Acknowledgement

When a flow is called via the bulk process, the immediate response is a 'Bulk Acknowledgement'. This tells the caller that the component store has successfully accepted the message and begun to trigger the child flows.

Usage

An example usage of call bulk flow is given below, firstly we invoke the call:

On Received Data	Move To State	Perform Enrichment	Generate Aggregate Data	Perform
Debulk Identifier Debulk Component Data Payment Journey Type Related Unit Of Work	Call Component Store	Derive Pain001 Instructions	<no aggregateFunction>	**For each** Document.CstmrCdtTrfInitn.PmtInf.CdtTrfTxInf **call flow** Transaction Flow

And then we handle the responses in the same way as child flows.

	With Current States	When	For Event	Move to State	Perform Action
1	Call Component Store	On	Bulk Acknowledgement	Awaiting Transaction Results	
2	Awaiting Transaction Results	On any of	**When** Transaction Flow **reaches** Complete **When** Transaction Flow **reaches** Rejected	Awaiting Transaction Results	**Call Function:** Threshold Check
3	Awaiting Transaction Results	On any of	Threshold Passed Threshold Passed With Errors	Complete	
4	Awaiting Transaction Results	On	Threshold Failed	Failed	

The IPF Debulker Domain

Source: https://docs.ipfdev.co.uk/functions/current/debulker-floclient/features/debulkerdomain.html

The IPF Debulker Domain

The IPF Debulker Domain is an external library representation of basic interaction with the component store.

It has two predefined functions for use:

Initiate Debulk

1	Initiate Debulk	<no description>	Debulk ID Debulk Source Debulk Config Name	**Name:** Initiate Debulk Response **Description:** <no description> **Business Data:** <no business data> **Response Codes:** AcceptOrReject **Reason Codes:** IPF Debulker Reason Codes **Completing:** [x] **Default Behaviour:** <no default behaviour>

The initiate debulk function provides support for starting a brand new debulk process. You provide it with the debulk ID, source, and config Name and then it will go and attempt to debulk the provided source information.

Once the debulking has completed the response will be returned, noting a rejected reason code will be returned if for any reason the debulk process fails.

Find Component Data

2	Find Component Data	<no description>	Debulk Identifier	**Name:** Find Component Data Response **Description:** <no description> **Business Data:** Debulk Component Data Debulk Identifier **Response Codes:** AcceptOrReject **Reason Codes:** <no reason codes> **Completing:** [x] **Default Behaviour:** <no default behaviour>

This function is used to retrieve the component data for any given debulk identifier.

Debulker Reason Codes

Source: https://docs.ipfdev.co.uk/functions/current/debulker-floclient/features/debulkerreasoncodes.html

Debulker Reason Codes

When debulking operations fail, the floclient returns specific reason codes to help identify the cause of the failure. These codes provide essential diagnostic information for troubleshooting issues during the debulking process.

For more details of all available reason codes, their meanings, and the system events that trigger them see: Debulker System Events

---

Threshold Checks

Source: https://docs.ipfdev.co.uk/functions/current/debulker-floclient/features/thresholdchecks.html

Threshold Checks

When a bulk file is split into multiple interacting levels of flows, it is often important to be able to determine when one level of flow completes.

For example, imaging a simple file with two top level records each containing 5 sub records. From a flow processing perspective, we may represent this as 2 "parent" flows and 10 "child flows". Here it can be important for a "parent" flow to know when all it's "children" are completed.

For this we introduce the concept of a threshold check. It's job is to track the interaction between parent flows and their children and determine when each has completed.

When implemented within a flow, the threshold check will track the outcomes of all children and then each time a child returns it's result status, it will determine whether the threshold for completion has passed.

Configuration

The threshold check by default assumes that to pass it requires 100% of the child transactions to return successfully. This figure can be altered through the property:

```
ipf.debulker.threshold.percentage=100
```

Outcomes

There are four outcomes possible from the threshold check:

1. Threshold Passed - this occurs when the number of records returned successfully exceeds or equals the threshold value without any failures occurring.
2. Threshold Failed - this occurs when the number of records that returns unsuccessfully means it is no longer possible to pass the threshold value.
3. Threshold Passed With Errors - this occurs when the number of records returned successfully exceeds or equals the threshold value but errors have occurred in some of the records.
4. In Progress- the threshold check will wait for more records as it is still waiting enough results to determine an outcome.

When any of the top three outcomes occurs an equivalent event is emitted which can then be handled as normal in the flo. When the fourth outcome occurs, the flow takes no further action and awaits for the next record to return.

Using the Threshold Check

We use the threshold check just as we would a normal action, calling it within the flow will automatically cause the three events to be emitted. These will then need to be handled as per standard flow processing, for example:

	With Current States	When	For Event	Move to State	Perform Action
1	Call Component Store	On	Bulk Acknowledgement	Awaiting Transaction Results	
2	Awaiting Transaction Results	On any of	**When** Transaction Flow **reaches** Complete **When** Transaction Flow **reaches** Rejected	Awaiting Transaction Results	**Call Function:** Threshold Check
3	Awaiting Transaction Results	On any of	Threshold Passed Threshold Passed With Errors	Complete	
4	Awaiting Transaction Results	On	Threshold Failed	Failed	

For a detailed guide to using the threshold checks, see the quick-start.

Getting Started

Source: https://docs.ipfdev.co.uk/functions/current/debulker-floclient/gettingstarted.html

Getting Started

The debulker floclient comes as one of the standard business functions available within any new IPF application that has been bootstrapped by the 'IPF Scaffolder'.

To use it, simply press `CTRL+R` twice and then select one of the 'IPF Debulker' components.

> If using an existing project that has not been configured to use the business functions, then it is simple to import them. Simply follow the instructions here.

How to implement a basic debulk

Source: https://docs.ipfdev.co.uk/functions/current/debulker-floclient/guides/yourfirstdebulk.html

How to implement a basic debulk

To complete this guide, you'll need the following:

1. To know your current IPF Version
2. To know the appropriate IPF Scaffolder Version

Creating the project

We'll start by creating a new project just like any other.

```
mvn com.iconsolutions.ipf.build:ipf-project-scaffolder-maven:<your-scaffold-version>:scaffold \
-DgroupId=com.icon.ipf \
-DartifactId=debulk-example-1 \
-DsolutionName=DebulkSolution \
-DprojectName=DebulkProject \
-DmodelName=DebulkModel \
-DincludeApplication=y \
-DflowName=BulkFlow \
-DipfVersion=<your-ipf-version> \
-DoutputDir=/build/debulk-example-1
```

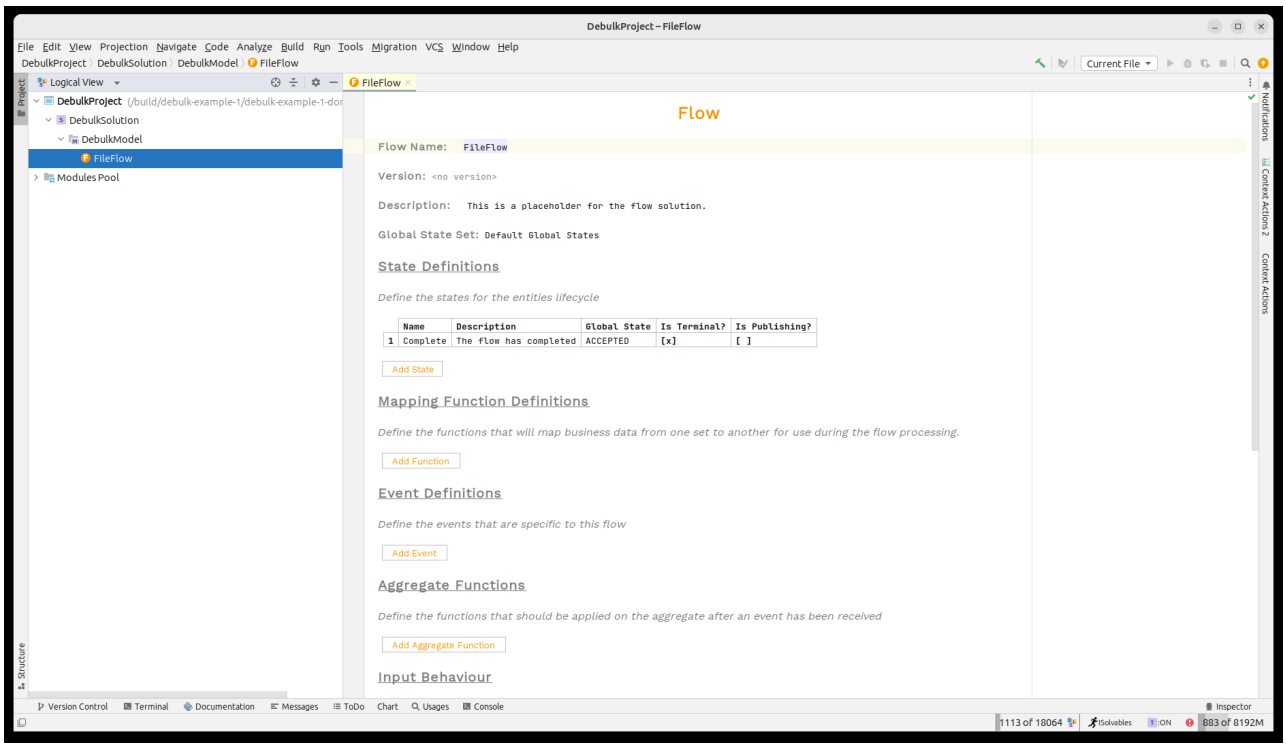
Once built, we'll build the project.

```
mvn clean install -DskipTests
```

We'll structure our debulker example using a standard pain001 as our input. We'll treat it as a set of four flows:

1. A 'File' flow to represent the handling of the file that arrives.
2. A 'Bulk' flow to represent the top level envelope of the pain001 itself.
3. A 'Batch' flow to represent the instruction part of the pain001.
4. A 'Transaction' flow to represent the transaction part of the pain001.

Now let's open the project in MPS. We'll see that our 'File Flow' has already been provided for us.



A brief reminder on the IPF Component Store

The IPF component store is used to debulk large files into components that can then be used within the normal flo system. In our case, we are

going to use pain001 files and split them into the different levels of the file.

The component store requires the configuration to tell it how to split up any given file structure it receives. This configuration is provided in the form of hocon, in the pain001 case an example is given below:

```
ipf.debulker {
 archiving.path = "/tmp/bulk_archive"
 configurations = [
   {
     name = "pain.001.001.09"
     splitter = "xml"
     processing-entity = "BANK_ENTITY_1"
     archive-path = "/tmp/bulk_archive"
     component-hierarchy {
       marker = "Document"
       children = [
         {
           marker = "CstmrCdtTrfInitn.PmtInf"
           children = [
             {
               marker = "CdtTrfTxInf"
             }
           ]
         }
       ]
     }
   }
 ]
}
```

The key points to note here:

1. The name 'pain.001.001.09' when an instruction to process a file is received, this name must then be provided to tell the component store to use this configuration.
2. Each 'marker' level represents the different levels of our pain001 as described above.

Modelling the Process

Creating the 'Bulk' flow.

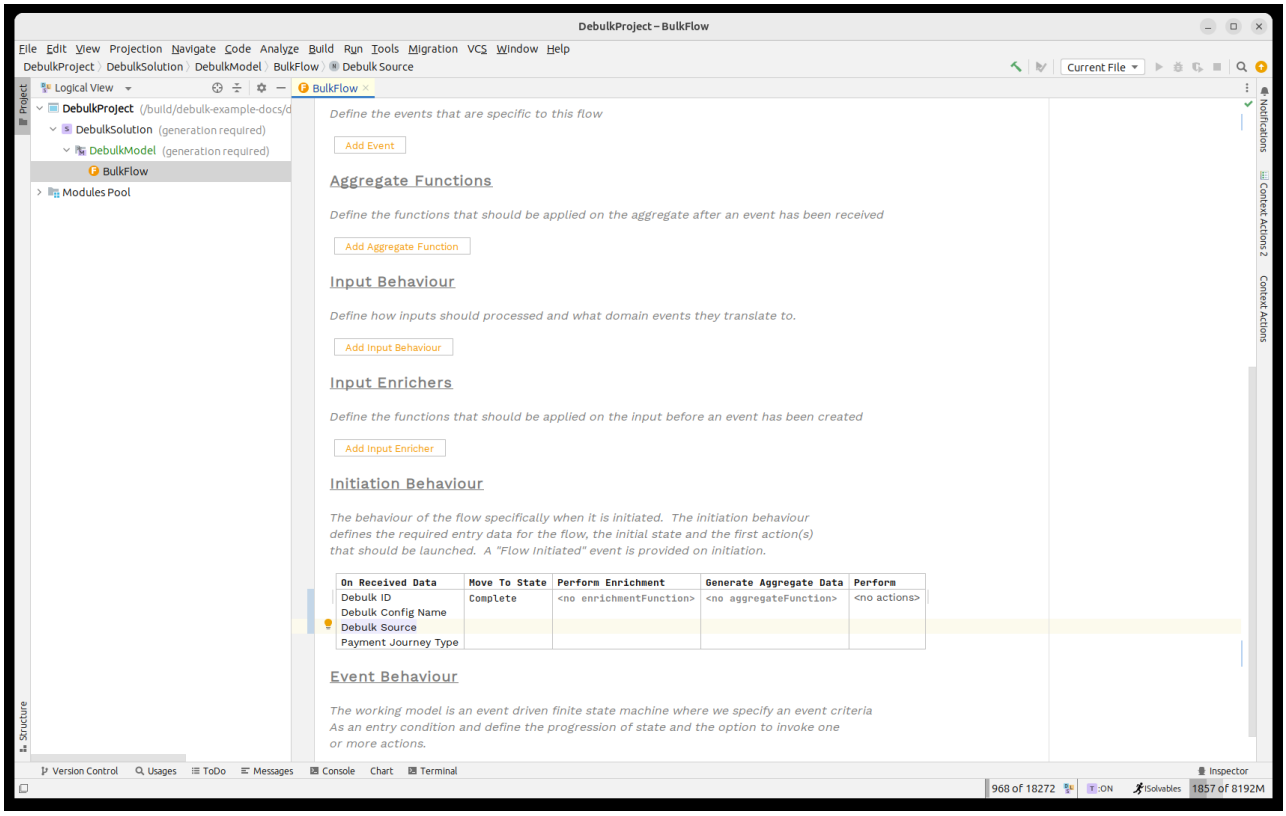
The bulk flow will be responsible for receiving some kind of instruction including details of where the pain001 file is stored. It will then interact with IPF's component store to read the file and debulk it into the appropriate component before kicking off the processing of top level flows.

From an instruction viewpoint, the component store needs the following key information to be provided:

- A unique ID for the debulk - this is modelled by the ' ***Debulk ID***' data element.
- The name of the config to use - this is modelled by the ' ***Debulk Config Name***' data element.
- The file location of data - this is modelled by the ' ***Debulk Source***' element.

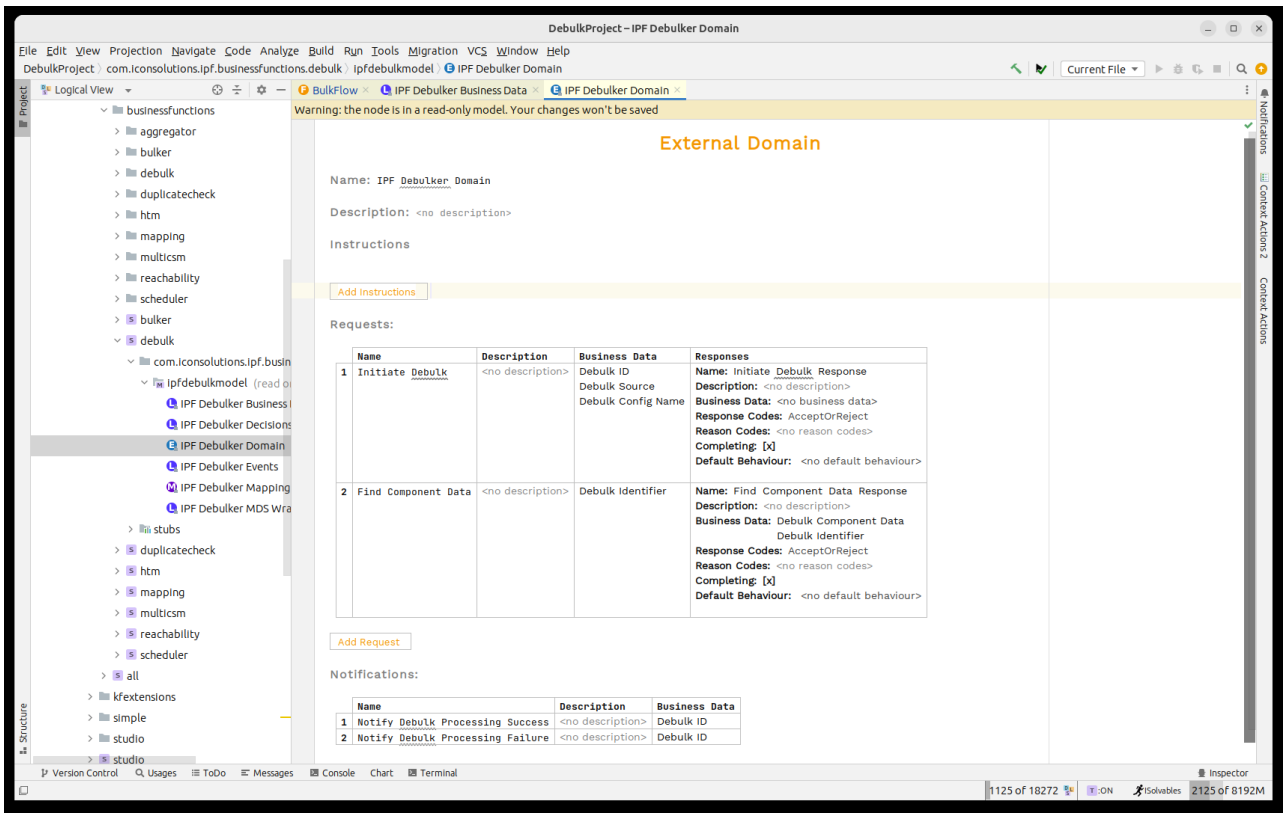
So let's start by adding those three data elements as initiation data into our flow. To do so we first need to import the debulker business data library. This can be done by pressing `CTRL`+`R` twice and then searching for 'IPF Debulker Business Data'.

Once done we should look like:



> Here we are using the data points directly, but it is perfectly reasonable to absorb your own file type and then use an aggregate function to extract the IPF fields required.

Next up we need to invoke the 'IPF Component Store' to process the bulk file. The component store has been modelled as the 'IPF Debulker Domain'. This has a function available 'Initiate Debulk' which takes the three fields we've made available and will process the file as a result.



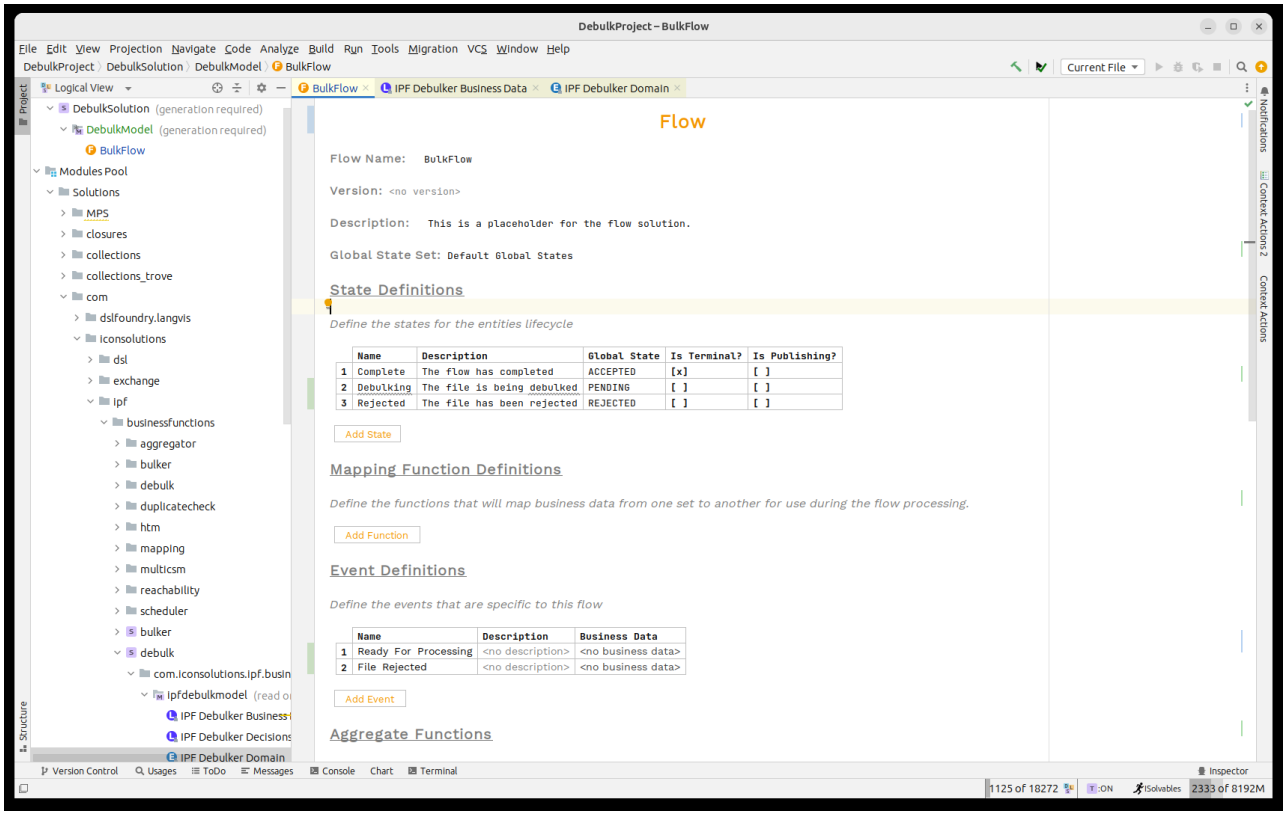
So just like any normal process we need to call this domain. To do so we'll think through the standard integration points:

1. We'll need a state to handle whilst the component store is being called, we'll call this 'Debulking'. We'll also need a state to handle failure, we'll call this 'Rejected'.

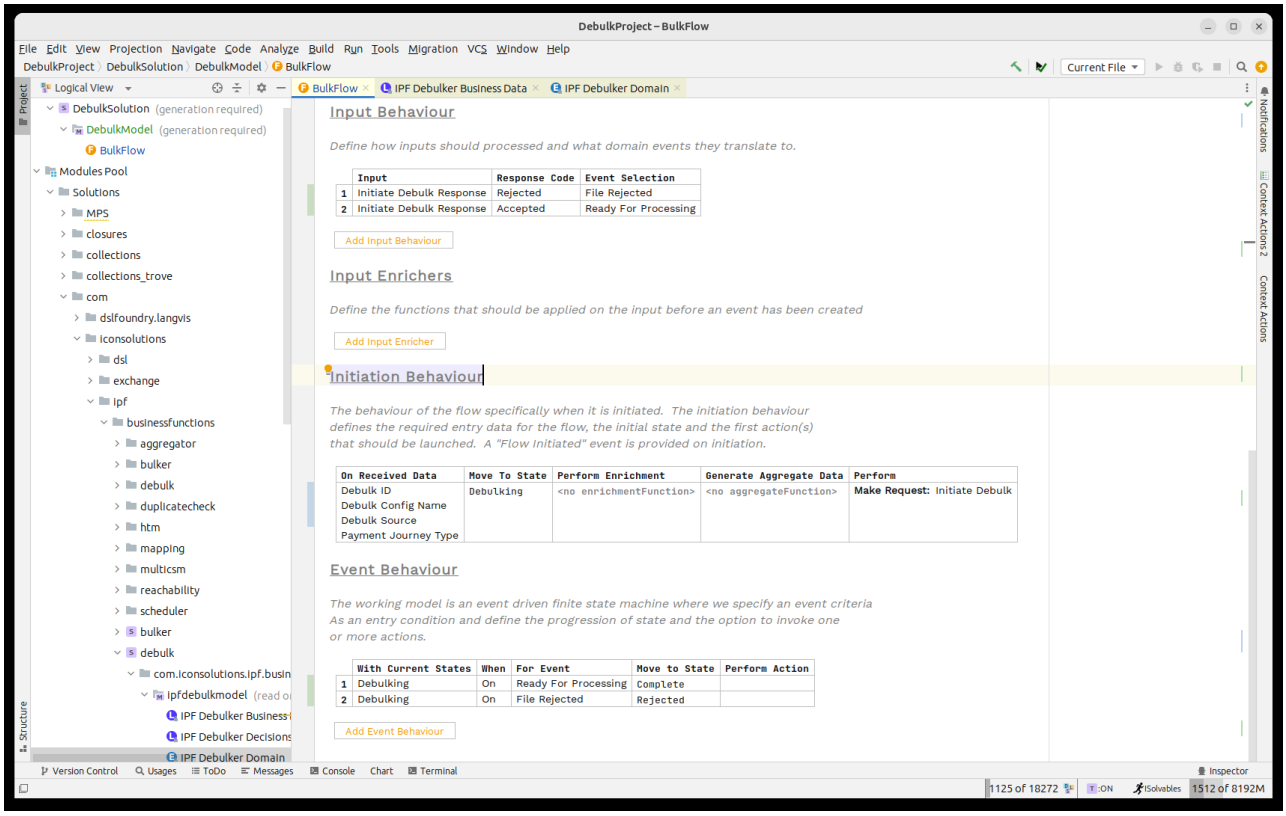
2. The file can debulk successfully or fail validation, so we'll need two events 'Ready for Processing' and 'File Rejected'.
3. We'll use the 'Ready for Processing' event on receipt of the 'Initiate Debulk Response' with the 'Accepted' code and similarly use the 'File Rejected' on the 'Rejected' code.
4. We'll call the component store on initiation.
5. When the successful event is raised we'll move to Complete, on failure we'll move to Rejected.

Let's go ahead and add all these capabilities in.

Firstly the states and events:



And then the behaviours:



Now we need to think about how to process the debulked file in the component store.

Initiating flows via the component store is very similar to initiating standard flows. The only difference here is that we may be initiating more than one flow at a time depending on how many records are in the component store entry.

So to use this we need to know two things:

1. The marker from the component store that the child records will be under.
2. The flow we wish to call.

In our case, the marker is 'Document.CstmrCdtTrfInitn.PmtInf' - this represents the point in the pain001 that the batch transactions are held.

For the flow, we'll create a new 'Batch' flow.

Creating the 'Batch' flow

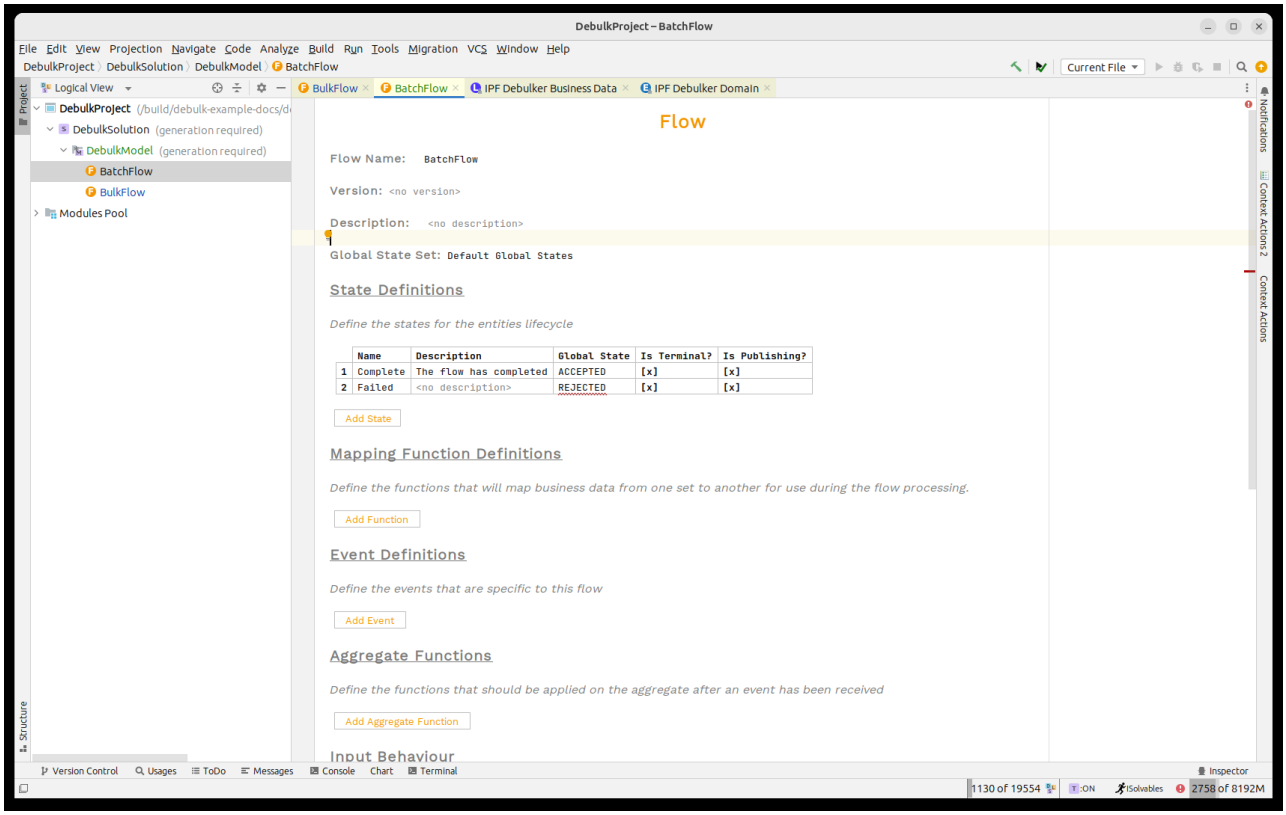
So before we start editing the bulk flow, let's create the shell for the batch flow.

To do this, we'll create a flow just like we did previously creating the bulk flow.

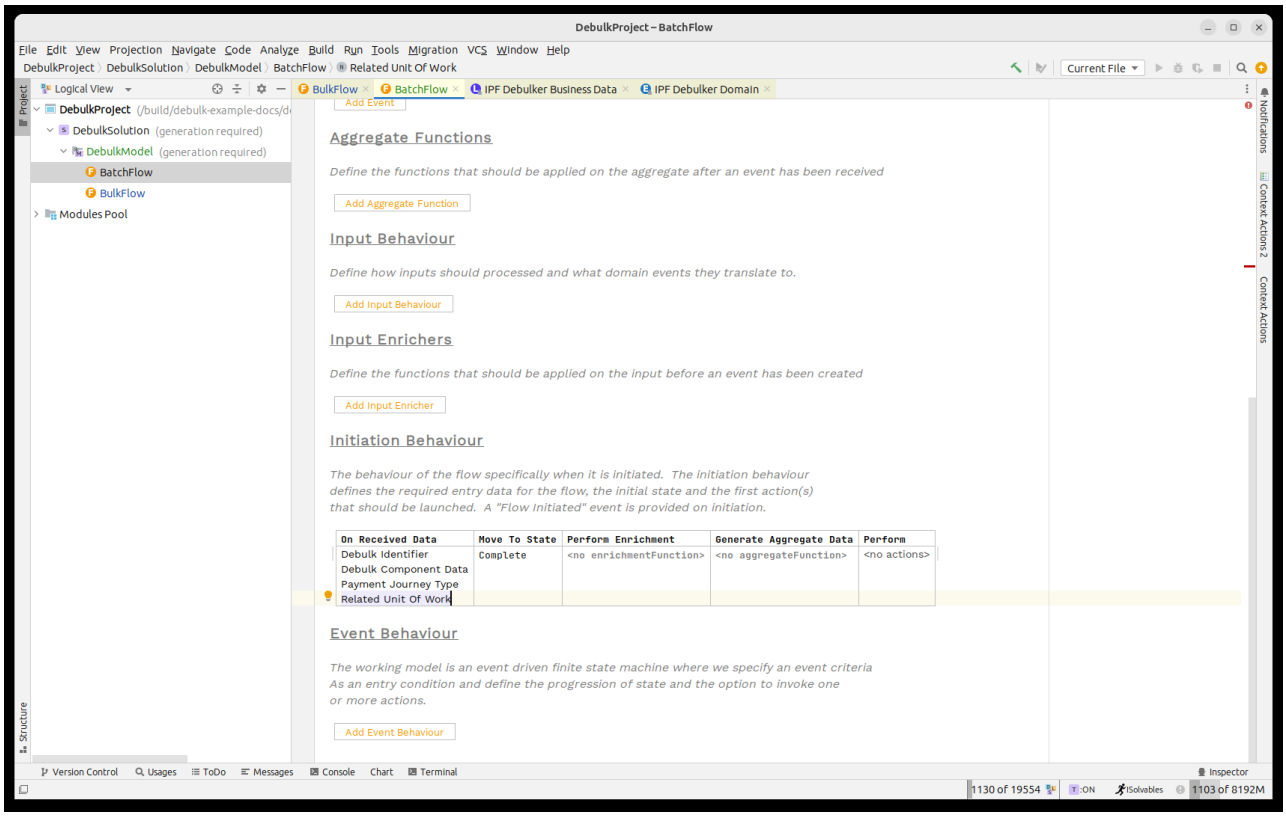
- It will be called 'Batch Flow'
- It will take in:
    - Debulk Identifier
    - Debulk Component Data
    - Payment Journey Type
    - Related Unit of Work
- It will have terminal, publishing states for:
    - Complete
    - Failed

Before we create it's worth mentioning the new business data element here 'Related Unit of Work'. This field is used to tie the different work streams of the bulk together - i.e. to establish the relationship between child and parent flows. From a processing perspective it is not required, and will be published automatically by the process. If it's missing however, you won't be able to navigate between the flows using the operational UI.

Let's now add in all the elements to our flow:



with the initiation setup:



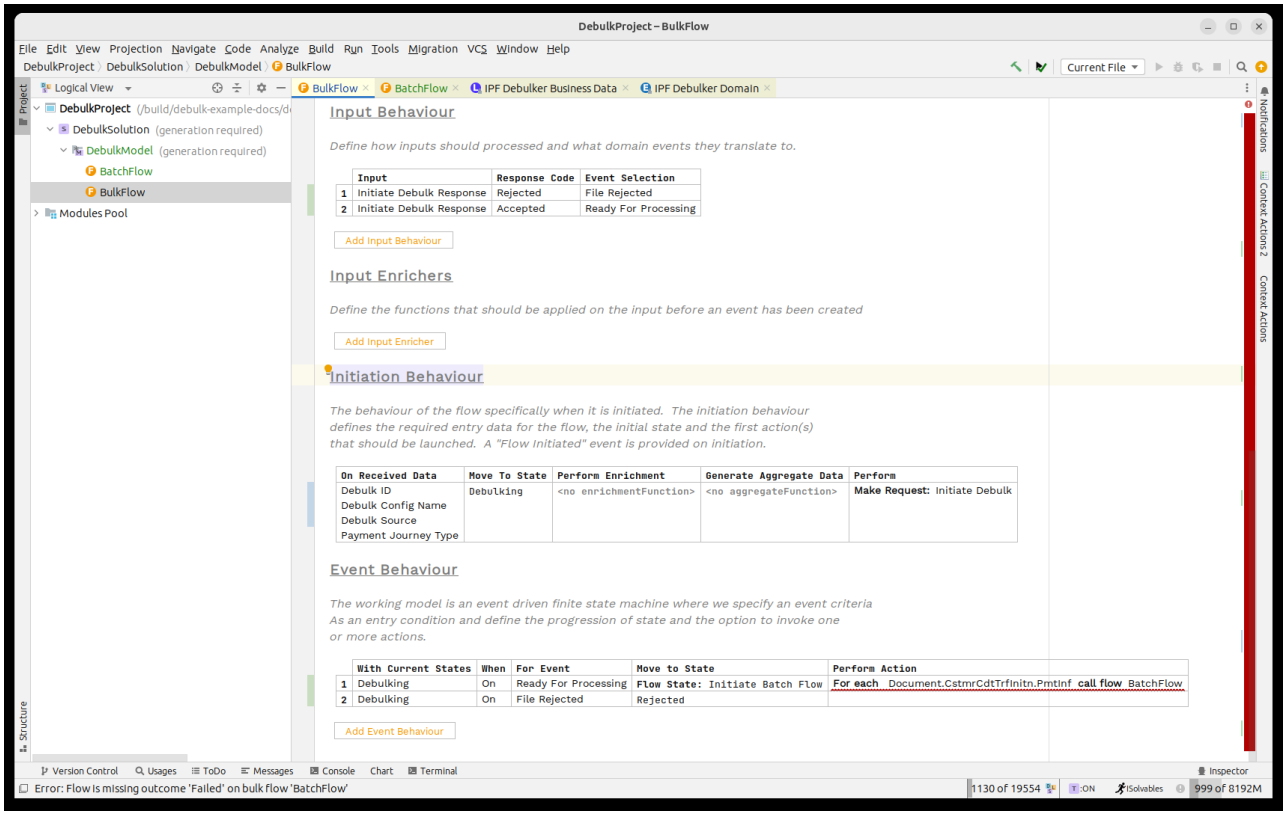
That's our batch flow shell defined, so let's return to the bulk flow and add in the start of our debulking process.

Calling the File Flow

So for this we use the concept of a 'Bulk Flow Call'. It works just like a normal flow call except also requires us to state which level of the bulk it needs to process on.

So we'll start by changing our flow so that on receipt of the 'Ready for Processing' event we'll call the 'Batch Flow' for each 'Document.CstmrCdtTrfInitn.PmtInf' element in the bulk.

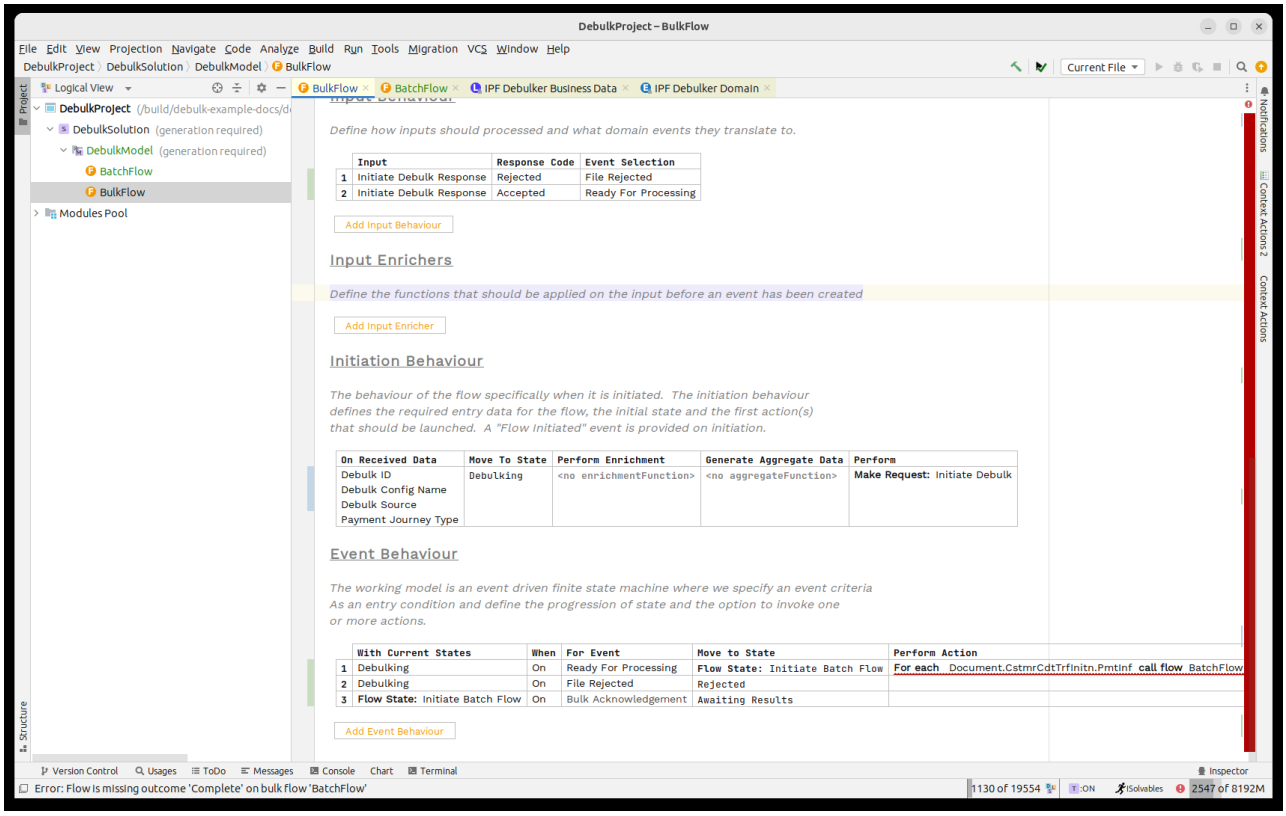
Let's change the processing now so that we move to a new 'Flow State'. We'll give this the identifier of 'Initiate Batch Flow' and then in the 'Perform Action' box we'll call our 'Bulk Flow' using the call bulk flow capability. Finally we'll specify the marker as 'Document'.



By using the 'Call Bulk Flow' capability we'll invoke the component store and then it will fire of as many flows as we need that match the provided marker.

Once it has been invoked, it will return an acknowledgement back to say that the request has been accepted.

So the first thing we need to do is handle that acknowledgement. It comes back as a special 'Bulk Acknowledgment' event type. To handle this we'll create a new state 'Awaiting Results' that we move to on receipt of the acknowledgement, we'll do nothing as an action here as we'll then wait for our bulk flow results to come back.

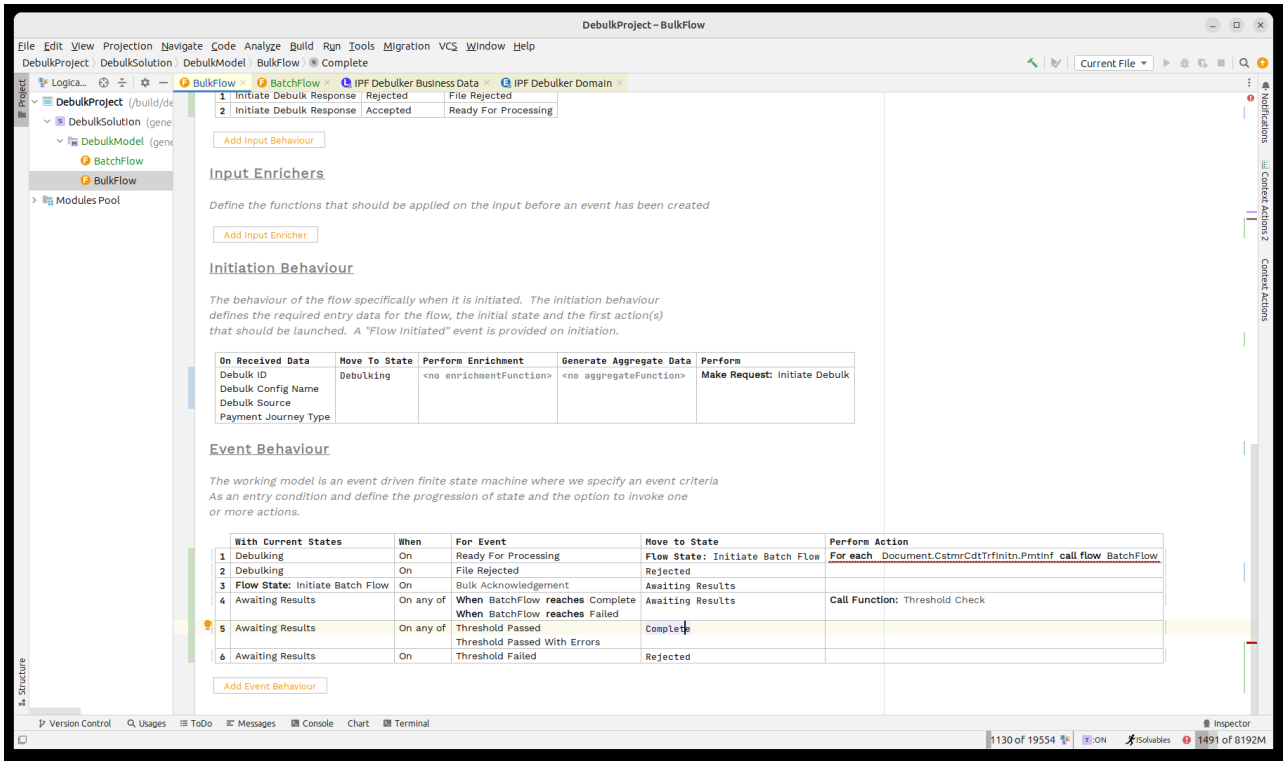


Next we need to consider the outcomes of our child batch flows coming back. For this we will use the threshold check to monitor when they have all returned.

So first, on receipt of a child result we stay in the same state and call the threshold check.

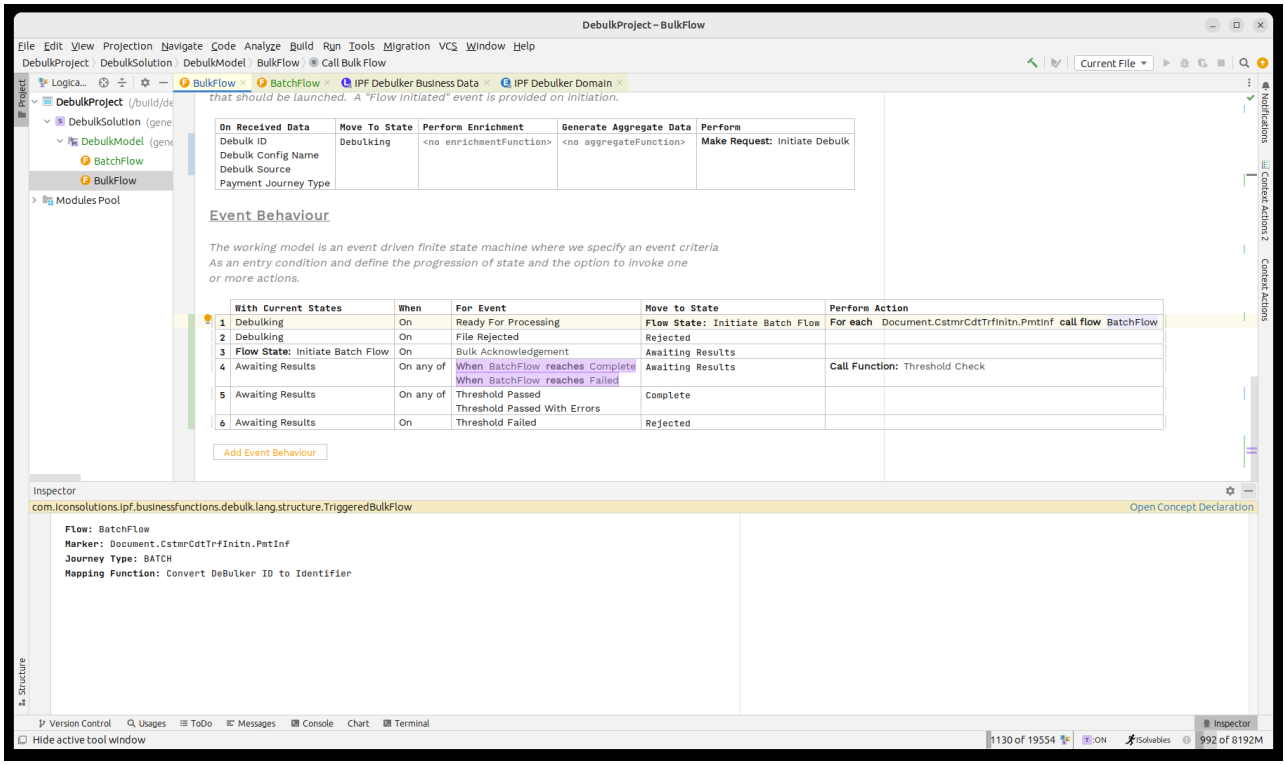
The Threshold Check function maintains a count of the results that have come back the child flows, it's configurable but by default will assume that it has completed when ALL flows have returned successfully and has failed when a SINGLE element has failed.

Then we have to handle the three possible results, in our case when one of the passed events is returned we'll move to complete and if the failure event arrives we'll move to rejected.



Finally, we note that the call to the batch flow is still showing in red. Let's bring up the inspector for it by pressing `CTRL+ALT+I`.

Here we need to set the journey type for our child flow, in this case 'BATCH' and then we'll use the pre-built mapping function 'Convert Debulker ID to Identifier' which will map our 'Debulk ID' into the relevant identifier format.



That's it we've now completed basic integration between our bulk and batch flows.

Now we think about how the batch flow will work. It needs to call the next level down, the 'Payment' level. So again we need to create a new flow.

Creating the 'Payment' flow

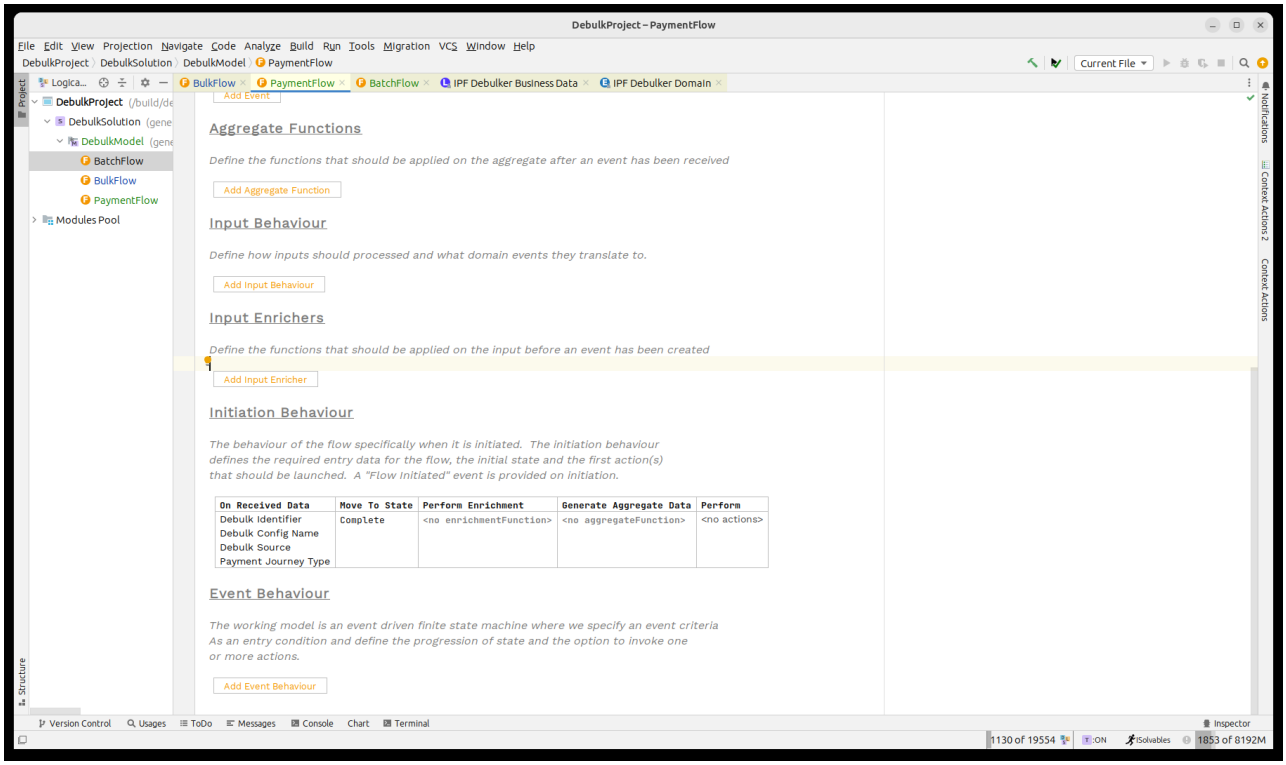
Next up we'll create a payment flow to handle the final level of our pain001.

In this case our payment flow we'll leave for now as just a simple default flow that will immediately complete.

From a data perspective it has the same elements as the batch flow:

- Debulk Identifier
- Debulk Component Data
- Payment Journey Type
- Related Unit of Work

Let's create this flow now.

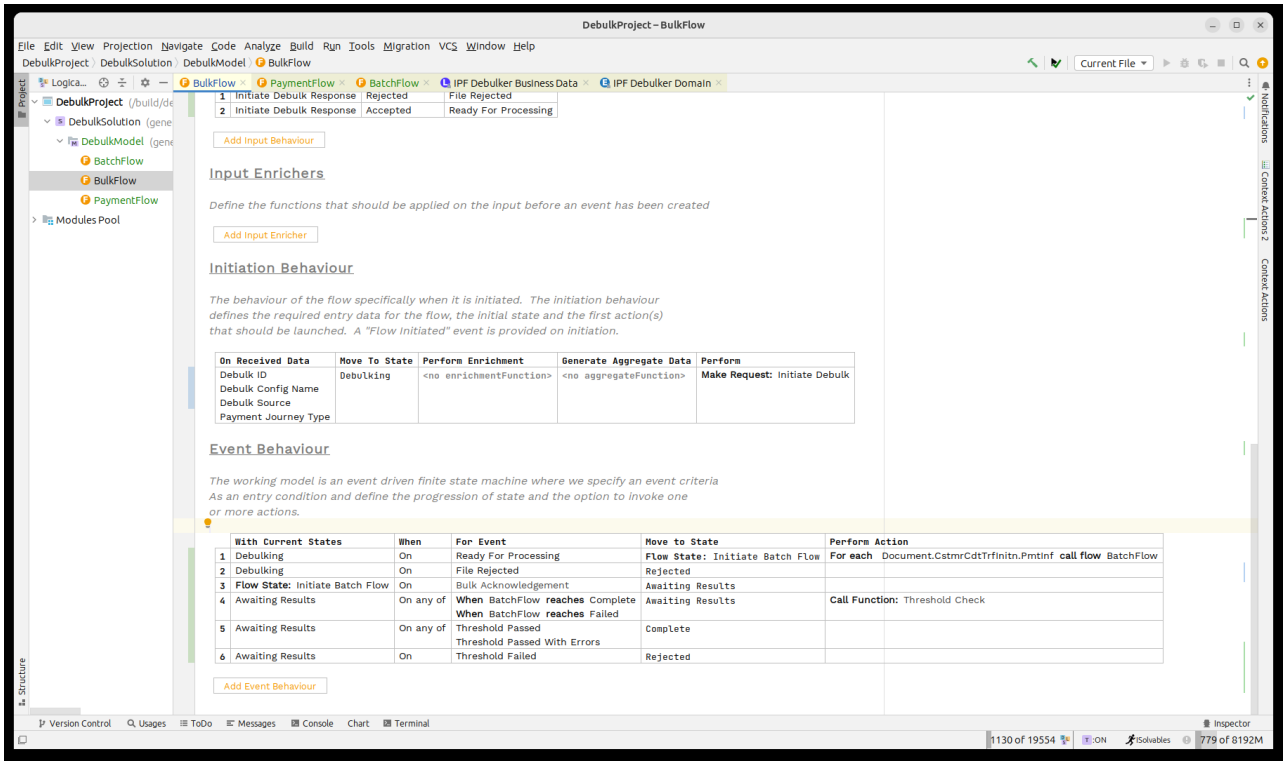


Implementing the 'Batch' flow

Now we can call the payment flow from the batch flow. The process is identical to that we set up for the bulk flow. As a reminder:

- On initiation we'll make a bulk flow call to the payment flow and use marker 'Document.CstmrCdtTrfInitn.PmtInf.CdtTrfTxInf'.
- When the bulk acknowledgement is received we'll move to an 'Awaiting results' state.
- When results from the payment flow are received we'll perform the Threshold Check.
- When the threshold check completes, we'll move to Complete.

Let's add all that in now:



Finally, just as with the previous flows we need to define the journey type of the child flow. We do this by bringing up the inspector (pressing

`CTRL`+`ALT`+`I`) and then setting the value to 'PAYMENT'.

That's it, we've now completed our DSL implementation for the debulking of our pain001.

Java Implementation

The next step would be to complete the implementation side of our flow.

Adding dependencies

Now that we've finished modelling our bulk process, it's time to start working on the implementation side.

Firstly we'll need to add in the dependency for the debulker client into our service project's pom.xml.

```
 <dependency>
    <groupId>com.iconsolutions.ipf.businessfunctions.debulk</groupId>
    <artifactId>ipf-debulker-floclient-service</artifactId>
</dependency>
```

We'll also delete the 'SampleController' from the application project as this is intended for use with normal single transaction files. We'll add a replacement in later.

Having done that, lets rebuild the project.

```
 maven clean install -DskipTests
```

Adding sample controllers

That's our example ready to run, but before that we need to work out how to initiate a new flow. In a real world there are many different ways we might want to feed the initiation process but here we'll simply make a sample controller that allows us to fire in transactions.

To do this we'll start by creating a simple bean to represent our initial data fields:

```
 @AllArgsConstructor
@NoArgsConstructor
@Data
@Builder
public class SampleDebulkInstruction {

    private String filePath;
    private String fileName;
    private String configName;
    private String bulkId;

}
```

Then we'll use this in a standard spring controller function that will call the initiation of our file flow:

```
 @RestController
public class FileController {

    @RequestMapping(value = "/submit", method = RequestMethod.POST)
    public Mono<InitiationResponse> submit(@RequestBody final SampleDebulkInstruction request) throws IOException {

        final String processingEntity = "BANK_ENTITY_1";
        final String unitOfWorkId = UUID.randomUUID().toString();
        final String filePath = Optional.ofNullable(request.getFilePath()).orElse("/tmp/");
        final String fileName = Optional.ofNullable(request.getFileName()).orElse(UUID.randomUUID() + ".xml");
        final String configName = Optional.ofNullable(request.getConfigName()).orElse("pain.001.001.09");
        final String debulkId = Optional.ofNullable(request.getBulkId()).orElse(UUID.randomUUID().toString());
        if (request.getFileName() == null) {
            // file not provided, so generate a dummy one.
            generateFile(filePath, fileName, 2, 10);
        }

        return Mono.fromCompletionStage(DebulkModelDomain.initiation().handle(new InitiateBulkFlowInput.Builder()
                        .withDebulkID(debulkId)
                        .withDebulkConfigName(configName)

.withDebulkSource(DebulkerFileSource.builder().fileProvider("local").filePath(filePath).fileName(fileName).build())
                        .withPaymentJourneyType("BULK")

.withProcessingContext(ProcessingContext.builder().unitOfWorkId(unitOfWorkId).clientRequestId(debulkId).processingEntity(processingEnti

                                .build())
                        .build())
```

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

    private void generateFile(String filePath, String fileName, int batches, int transactionPerBatch) throws IOException {
        var xmlMapper = ISO20022MessageModel.init().xmlMapper();
        xmlMapper.setPruneWhitespaceOnSerialization(true);
        var generator = new Pain001Generator();
        var doc = new Document();
        doc.setCstmrCdtTrfInitn(generator.generate(batches, transactionPerBatch));
        var xml = xmlMapper.toXML(doc);
        File file = new File(filePath + fileName);
        FileUtils.writeStringToFile(file, xml, StandardCharsets.UTF_8);
    }
}
```

The key points to note here are that we call our Debulk Domain and just pass it the relevant data just as we do on any normal initiation flow. It also provides a file generator for pain001's so that if required a new file is generated for testing purposes.

Finally as the component store uses Kafka, we need to add our normal default kafka properties.

```
 common-kafka-client-settings {
  bootstrap.servers = "localhost:9093"
}

akka.kafka {
  producer {
    kafka-clients = ${common-kafka-client-settings}
    restart-settings = ${default-restart-settings}
    kafka-clients {
      client.id = ipf-tutorial-client
    }
  }
  consumer {
    kafka-clients = ${common-kafka-client-settings}
    restart-settings = ${default-restart-settings}
    kafka-clients {
      group.id = ipf-tutorial-group
    }
  }
}

default-restart-settings {
  min-backoff = 1s
  max-backoff = 5s
  random-factor = 0.25
  max-restarts = 5
  max-restarts-within = 10m
}
```

You'll also need to include the pain001 debulking configuration.

That's our implementation done, time to try and run our bulk process!

Running the process

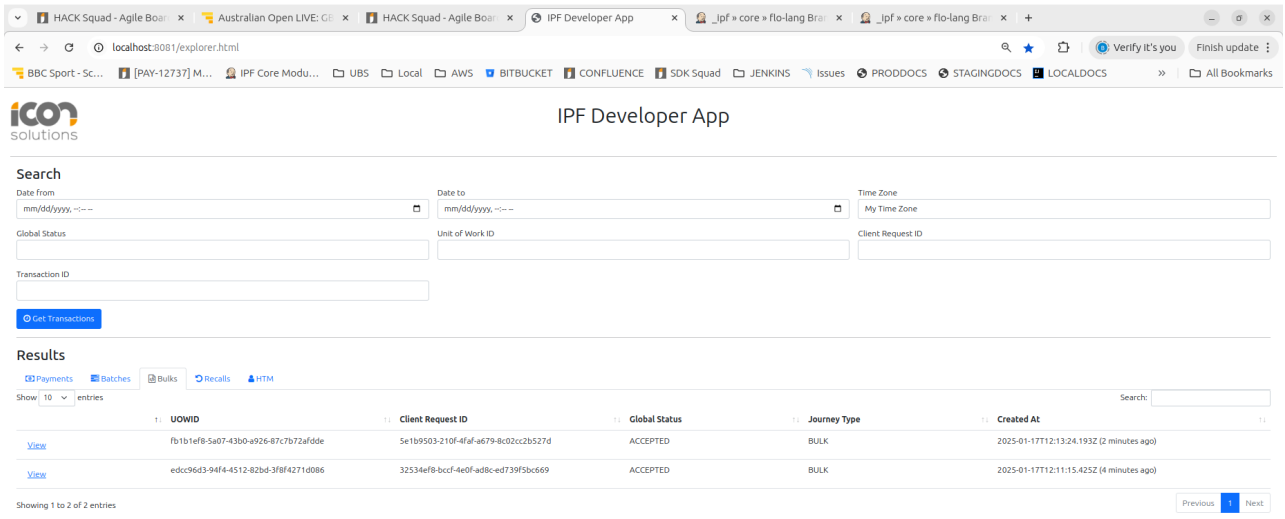
For the process to run we need access to Kafka and Mongo. We'll use the standard setup for this and run:

```
docker compose -f minimal.yml -f kafka.yml up -d
```

Then once up we can simply start our application as a standard springboot one. Once it's up the simplest thing is to fire a new record using the little sample controller we created:

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

Then we can check our output by looking in the developer app:



Here we can see the different levels of the processing: 'Payments', 'Batches' and Bulks.

How to specify which states are successful in a threshold check

Source: https://docs.ipfdev.co.uk/functions/current/debulker-floclient/guides/specifycustomstates.html

How to specify which states are successful in a threshold check

When running a threshold check, by default the the threshold check will look for a state called 'Complete'. If it receives an outcome with this value, then it will recognise the outcome as successful. Otherwise, it will assume the outcome is failure. There are two approaches to do this:

By Java Class Injection

It may be that you require to specify your own state matching mechanism for this. In order to do this we simply inject into the spring context a new 'DebulkerThresholdStateOutcome' for example adding this to a config file:

```
 @Bean
public DebulkerThresholdStateOutcome mySuccessMatcher() {
    return new DebulkerThresholdStateOutcome("myFlow", "mySuccessState", true);
}
```

Here for the flow 'myFlow' when it receives the state 'mySuccessState' then it will result in a positive match.

By Configuration

You can define your state outcomes by configuration using the following example:

```
 ipf.debulker.threshold.outcomes  = [
  {
    flowName = "ABC"
    stateName = "MatchedOne"
    success = false
  },
  {
    flowName = "ABC"
    stateName = "MatchedTwo"
    success = true
  },
]
```

Here we are setting up two outcomes that will then be used just like the java one defined previously.

Retrieving parent data from a child

Source: https://docs.ipfdev.co.uk/functions/current/debulker-floclient/guides/parentchildprocessingdata.html

```

```

Retrieving parent data from a child

By default, when a parent flow calls into a child flow, the data structures related to the parent flow are not available in the child flow.

One of the situations where we might wish to retrieve the parent data structures is in bulking/debulking; processing is split across levels with many C-Levels (creditor information) required per B-Level (debtor information). When processing C-Level it is necessary to access debtor information for booking/settlement.

Accessing the Parent Flow

In the absence of other alternatives, to access these structures (such as the MDS and PDS), a call would be required through an external domain to ODS to retrieve the information.

However, there are better and more efficient ways to retrieve this information:

1) If the parent is deployed on a separate processing service then instead we need to use transactionOperation API deployed on the parent service, which contains an endpoint which allows the aggregate to be retrieved.

2) If the parent is present in the same processing service. In this situation, the data structures can be retrieved from the flo-domain API.

Using the flo-domain API

> This assumes the parent flow is available in the same processing service.

An alternative to a call to ODS to retrieve the data is using a call to `getAggregate` from the `flo-domain` API, where both the parent and the child flow are on the same processing service.

This makes use of the fact that a child action knows the flow Id that called it, and as such the data can be retrieved.

The Aggregate class specified as a parameter in the call belongs to the parent flow, although of course the retrieval is occurring during processing within the action of the child flow.

In the situation below, the `TransactionFlow` is being called from the `BatchFlow`, so the aggregate being retrieved is the `BatchFlowAggregate`.

```
 import debulkexamplemodel.domain.DebulkexamplemodelModelOperations;
[...]
debulkexamplemodelModelOperations.getAggregate(parentId, BatchFlowAggregate.class)
```

Processing the values

MDS and PDS values are filtered from the parent aggregate for use:

```
return debulkexamplemodelModelOperations.getAggregate(parentId, BatchFlowAggregate.class)
               .thenAccept(batchFlowAggregate -> {
                   MdsWrapper<PaymentInstruction30> parentMdsValues = batchFlowAggregate.getPain001Instruction();
                   List<? extends DataElementWrapper<?>> parentPdsValues = batchFlowAggregate.businessData().values()
                           .stream()
                           .filter(dataElementWrapper -> dataElementWrapper.getCategory().equals("PROCESSING_DATA_STRUCTURE"))
                           .toList();
               });
```

Using the cache

In addition to this, if the same parent id is being retrieved multiple times, caching would be useful. ipf-cache should be used for this, to reduce the repeated use of getAggregate.

To facilitate this, the following entry can be added to the `pom.xml`

```
<dependency>
    <groupId>com.iconsolutions.ipf.core.platform</groupId>
    <artifactId>ipf-cache-api</artifactId>
</dependency>
<dependency>
    <groupId>com.iconsolutions.ipf.core.platform</groupId>
    <artifactId>ipf-cache-caffeine</artifactId>
</dependency>
```

Further documentation on how to set up the cache for use is here.

By doing this, an implementation can be used that avoids the pitfalls associated with re-reading from the aggregate. Further details here.

Implementing the cache

When this is all brought together in the `ActionAdapter` for the child flow, we can have something like the following:

```
     private final DebulkexamplemodelModelOperations debulkexamplemodelModelOperations;
    private final CacheAdapter<String, MdsPdsWrapper> cacheAdapter;
```

```
private Optional<MdsPdsWrapper> getFromCache(String parentId) {
    var mdsPdsWrapper = cacheAdapter.get(parentId);
    if (mdsPdsWrapper.isPresent()) {
        log.debug("Existing MdsPdsWrapper retrieved from cache; parentId:{} mdsPdsWrapper:{}", parentId, mdsPdsWrapper);
    }
    return mdsPdsWrapper;
}
```

```
CompletionStage<Void> getFromCacheOrLoad(String parentId) {
    return getFromCache(parentId).isPresent() ?
            CompletableFuture.completedFuture(null) :
            retrieveParentMdsPdsWrapper(parentId);
}
```

```
private CompletionStage<Void> retrieveParentMdsPdsWrapper(String parentId) {
    return debulkexamplemodelModelOperations.getAggregate(parentId, BatchFlowAggregate.class)
            .thenAccept(batchFlowAggregate -> {
                MdsWrapper<PaymentInstruction30> parentMdsValues = batchFlowAggregate.getPain001Instruction();
                List<? extends DataElementWrapper<?>> parentPdsValues = batchFlowAggregate.businessData().values()
                        .stream()
                        .filter(dataElementWrapper -> dataElementWrapper.getCategory().equals("PROCESSING_DATA_STRUCTURE"))
                        .toList();
                MdsPdsWrapper mdsPdsWrapper = new MdsPdsWrapper(parentMdsValues, parentPdsValues);
                cacheAdapter.put(parentId, mdsPdsWrapper);
                log.debug("Loaded into cache -- parentId:{} mdsPdsWrapper:{}", parentId, mdsPdsWrapper);
            });
}
```

> The `MdsWrapper` class is a POJO which was added to simplify the values stored in the cache.

\n

How to Add Business Functions to an existing IPF Solution

1 pages in this subsection

How to Add Business Functions to an existing IPF Solution

How to Add Business Functions to an existing IPF Solution

Prerequisites

Your project must be using IPF release 2024.1.0 or later.

Update your maven dependency plugin executions

`Add the `unpack-ipf-business-functions-plugin` execution to the `maven-dependency-plugin` build plugin included within your `mps` module. If your project was generated using the IPF Archetype this will be a submodule of `<your-project-name>-domain`. See below:`

```
<execution>
   <id>unpack-ipf-business-functions-plugin</id>
   <phase>initialize</phase>
   <goals>
       <goal>unpack</goal>
   </goals>
   <configuration>
       <artifactItems>
           <artifactItem>
               <groupId>com.iconsolutions.ipf.businessfunctions.aggregator.domain</groupId>
               <artifactId>mps</artifactId>
               <version>${icon-business-functions-aggregator.version}</version>
               <type>zip</type>
               <overWrite>true</overWrite>
               <outputDirectory>${plugin_home}</outputDirectory>
           </artifactItem>
       </artifactItems>
   </configuration>
</execution>
```

Add the aggregator's external libraries as a dependency

Add the below dependency to the pom.xml for your `domain` module. If your project was generated using the IPF Archetype this will be a submodule of `<your-project-name>-domain`.

```
 <dependency>
    <groupId>com.iconsolutions.ipf.businessfunctions.aggregator.domain</groupId>
    <artifactId>external-libraries</artifactId>
</dependency>
```

Rebuild your MPS project

Rebuild your project with `mvn clean install`.

You will see the business functions listed within your MPS project. They can be found at the package path `com.iconsolutions.ipf.businessfunctions` in the `Solutions` and `DevKits` folders of your `Modules Pool`.
