Processing IPF contains a number of features within the core which are optionally used depending on your solution requirements. This section covers those

which are related to core processing.

IPF Cache - this module provides a simple Caffeine backed caching interface

that IPF solutions can use for in memory caching (also see Transaction Caching

for transaction specific caching, especially where a long lived persistent cache is

required).

Persistent Scheduler - scheduling future processing and jobs is covered by this

component and works with cron expressions in combination with calendars to

execute one-time or recurrent jobs. It also features a persistence layer whose

main role is to keep single source of truth.

Dynamic Settings - Dynamic Processing Settings provides a platform to manage

configurable settings and their lifecycle.

Bulk File Processing - the modules here provide the capability to process bulks

of transactions or records, streamed from or to files.

Message Logger - module and interface to allow the logging and/or publishing

of IPF processing data.

---------- IPF Cache

The ipf-cache module provides a simple caching interface that IPF products

can use for caching.

The current offering is a Caffeine backed cache baked into Spring or InfiniSpan.

Concepts

Features

Caffeine

InfiniSpan

Getting Started

---------- Concepts The

ipf-cache module provides a simple caching interface that IPF products can

use for caching.

The current offering is a Caffeine backed cache baked into Spring or Infinispan.

API The API is based upon common API interactions and comes in 2 flavours, asynchronous and synchronous.

```
package com.iconsolutions.ipf.core.platform.cache.api;

import com.iconsolutions.ipf.core.shared.domain.context.ProcessingContext;
```

```
import java.util.concurrent.CompletionStage; import java.util.function.Function;

/** * Asynchronous IPF cache adapter * * @param <K> key * @param <V>

Value */ public interface AsyncCacheAdapter<K,V> {

/** * Retrieve a future value from cache * @return future containing nullable

value */ CompletionStage<V> get(K key);

/** * Retrieve a cache entry or if missing provide a future to make the value

* @return future containing value cached */ CompletionStage<V> getOrDefault(K key, Function<K, CompletionStage<V» callback);

/** * Retrieve a cache entry, add to messageLog or if missing provide a future to make the value * @return future containing value cached */ CompletionStage<V> getThenLogOrDefault(ProcessingContext processingContext, K

key, Function<K, CompletionStage<V» callback);

/** * Pass a future which upon completion will add the cache entry */ CompletionStage<Void> put(K key, CompletionStage<V> value);

/** * Manually evict an entry */ CompletionStage<Void> evict(K key);

/** * Evict all entries */ CompletionStage<Void> clear(); } package

com.iconsolutions.ipf.core.platform.cache.api;

import com.iconsolutions.ipf.core.shared.domain.context.ProcessingContext;

import java.util.Optional;

/** * Synchronous IPF cache adapter * * @param <K> key * @param <V>

Value */ public interface CacheAdapter<K, V> {

/** * Retrieve from cache * * @param key * @return Optional<value> */ Optional<V> get(K key);

/** * Put an entry into cache * * @param key * @param value */ void put(K

key, V value);

/** * Manually evict the cached item for this key * * @param key */ void

evict(K key);

/** * Clear all cached entries. */ void clear();

/** * Retrieve an entry or if absent use the provided value * * @param key

* @param defaultValue * @return value cached */ default V getOrDefault(K

key, V defaultValue) { return get(key).orElseGet(() -> { put(key, defaultValue);

return defaultValue; }); }

/** * Retrieve an entry or if absent use the provided value * * * @param processingContext * @param key * @param defaultValue * @return value cached
```

*/ V getThenLogOrDefault(ProcessingContext processingContext, K key, V defaultValue); } Both implementations are created based on the Cache Factory

package com.iconsolutions.ipf.core.platform.cache.api;

/** * Factory to provide the cache implementation by name * * @param <K>

key * @param <V> value */ public interface CacheFactory<K, V> { CacheAdapter<K, V> createCacheAdapter(String name);

AsyncCacheAdapter<K, V> asyncCreateCacheAdapter(String name); }

--- Features The current IPF-Cache offering is a Caffeine

backed cache baked into Spring or InfiniSpan.

Here are the key features:

Caffeine

InfiniSpan --- Caffeine Introduction Caffeine is a high-performance caching library for

Java.

One fundamental difference between a cache and a Map is that a cache evicts

stored items.

An eviction policy decides which objects should be deleted at any given time.

This policy directly affects the cache’s hit rate — a crucial characteristic of

caching libraries.

Caffeine uses the Window TinyLfu eviction policy, which provides a nearoptimal hit rate.

Caffeine Configuration First, let’s create a Caffeine bean. This is the main

configuration that will control caching behavior such as expiration, cache size

limits, and more:

private CaffeineCache buildCaffeineCache(String name, CaffeineCacheSetting cacheSpec) { log.info(”Cache {} specified timeout of {} min, max

of {}”, name, cacheSpec.getTimeout(), cacheSpec.getMaxSize()); final Caffeine<Object, Object> caffeineBuilder = Caffeine.newBuilder() .expireAfterWrite(cacheSpec.getTimeout())

.maximumSize(cacheSpec.getMaxSize())

.recordStats(); return new CaffeineCache(name, caffeineBuilder.build()); }

Next, we need another bean using the Spring CacheManager interface. Caffeine

provides its implementation of this interface, which requires the Caffeine object

we created above:

@Bean(name = ”ipfCacheManager”) CacheManager ipfCaffeineCacheManager() { SimpleCacheManager manager = new SimpleCacheManager(); if (Objects.nonNull(settings)) { List<CaffeineCache> caches =

settings.entrySet().stream() .map(entry -> buildCaffeineCache(entry.getKey(),

entry.getValue())) .collect(Collectors.toList()); manager.setCaches(caches); }

return manager; } All the beans mentioned above we get for free when adding

the maven dependency mentioned before.

However, the Caffeine cache requires the following configuration values for each

cache that is provided:

ipf.caching.caffeine.settings.”${cache_name}”.timeout=[Duration]
ipf.caching.caffeine.settings.”${cache_name}

size=[Long] cache_name - name of the cache being used

timeout - duration cache will remain in memory active before being evicted.

max-size - maximum cache size before the cache evicts entries that are less likely

to be used again see Caffeine

An Example:

ipf.caching.caffeine.settings.cache1.timeout=10m
ipf.caching.caffeine.settings.cache1.maxsize=10000 ipf.caching.caffeine.settings.cache2.timeout=20s
ipf.caching.caffeine.settings.cache2.maxsize=100000 Caffeine Implementation Implementation is simple, as this module

is spring boot aware it will wire up all the necessary beans and hook them into

the Spring CacheManager providing metrics.

Just add the maven dependency and then retrieve any caches by name.

We get the CacheFactory bean for free from ipf-cache-caffeine module and by

enabling caffeine caching.

@Bean(name = ”caffeineCacheFactory”) CacheFactory<?, ?> cacheFactory(CacheManager cacheManager, CacheLogger<Object, Object> cacheLogger) { return new CaffeineCacheFactory(cacheManager, cacheLogger); } Then,

you just need to use the CacheFactory to create either an AsyncCacheAdapter:

@Bean AsyncCacheAdapter<String, String> asyncCacheAdapter3(CacheFactory<String,

String> cacheFactory) { return cacheFactory.asyncCreateCacheAdapter(CACHE_3);

} Or a CacheAdapter:

@Bean CacheAdapter<?, ?> cacheAdapter1(CacheFactory<?, ?> cacheFactory) { return cacheFactory.createCacheAdapter(CACHE_1); } Dependencies

The dependency on ipf-cache-api module requires some supplied dependencies

to read and write to the message log.

These can be added to your application (if not present) as follows:

@Bean ObjectMapper objectMapper() { return new ObjectMapper(); }

@Bean MessageLogger messageLogger() { return messageLogEntry ->

log.info(”log entry: {}”, messageLogEntry); }

----------InfiniSpan

Introduction Infinispan is an in-memory key/value data store that ships with a

more robust set of features than other tools of the same niche.

It provides a flexible, in-memory data stores that you can configure to suit use

cases such as:

Boosting application performance with high-speed local caches.

Optimising databases by decreasing the volume of write operations.

Providing resiliency and durability for consistent data across clusters.

Infinispan Configuration The CacheManager is the foundation of the majority of features that we'll use. It acts as a container for all declared caches, controlling their lifecycle, and is responsible for the global configuration.

Infinispan ships with a really easy way to build the CacheManager:

```
@Bean InfinispanCacheProvider infinispanCacheProvider(final Marshaller marshaller) { var configuration = kubernetesStack ? buildClusteredConfigurationForKubernetes(marshaller) : buildDefaultClusteredConfiguration(marshaller); var cacheManager = new DefaultCacheManager(configuration); return new InfinispanCacheProvider(cacheManager, settings); } private GlobalConfiguration buildClusteredConfigurationForKubernetes(Marshaller marshaller) { return GlobalConfigurationBuilder.defaultClusteredBuilder() .cacheManagerName(cacheManagerName) .transport() .addProperty("stack", "kubernetes") .addProperty("configurationFile", "default-configs/defaultjgroups-kubernetes.xml") .initialClusterSize(initialClusterSize) .initialClusterTimeout(initialClusterTimeout.getSeconds(), TimeUnit.SECONDS) .serialization() .marshaller(marshaller) .build(); } private GlobalConfiguration buildDefaultClusteredConfiguration(Marshaller marshaller) { return GlobalConfigurationBuilder.defaultClusteredBuilder() .cacheManagerName(cacheManagerName) .transport() .initialClusterSize(initialClusterSize) .initialClusterTimeout(initialClusterTimeout.getSeconds(), TimeUnit.SECONDS) .serialization() .marshaller(marshaller) .build(); } A
```

cache is defined by a name and a configuration. The necessary configuration can be built using the class ConfigurationBuilder, already available in our classpath.

The ConfigurationBuilder is provided with the following method:

private Cache<Object, Object> buildInfinispanCache(final String name, final InfinispanCacheSetting infinispanCacheSetting) { log.info(”Cache {} specified timeout of {} min, max of {}”, name, infinispanCacheSetting.getTimeout(), infinispanCacheSetting.getMaxSize());

var configBuilder = new ConfigurationBuilder(); var cacheMode CacheMode.valueOf(infinispanCacheSetting.getCacheMode());

=

configBuilder.clustering() .cacheMode(cacheMode) .encoding().mediaType(”application/json”) .memory() .maxCount(infinispanCacheSetting.getMaxSize()) .whenFull(EvictionStrategy.REMOVE) .expiration() .lifespan(infinispanCacheSetting.getTimeout().toMillis(), TimeUnit.MILLISECONDS);

5

if (isRemote(cacheMode)) { configBuilder.clustering()
.stateTransfer().fetchInMemoryState(infinispanCacheSet

.awaitInitialTransfer(infinispanCacheSetting.getAwaitInitialStateTransfer())

.timeout(infinispanCacheSetting.getStateTransferTimeout().toMillis()); }

final Cache<Object, Object> cache =
cacheManager.administration().withFlags(CacheContainerAdmin.Admin

.getOrCreateCache(name, configBuilder.build());

setCacheLevelLogging(cache, name, cacheMode, infinispanCacheSetting);
setClusterLevelLogging(cacheMode, infinispanCacheSetting); return cache; }

private void setCacheLevelLogging(final Cache<Object, Object> cache, final

String cacheName, final CacheMode cacheMode, final InfinispanCacheSetting

infinispanCacheSetting) {

if (isRemote(cacheMode) && infinispanCacheSetting.getClusterLogging())

{ cache.addListener(new ClusterCacheLoggingListener(cacheName)); } if

(infinispanCacheSetting.getLocalLogging()) { cache.addListener(new
LocalCacheLoggingListener(cacheName)); } }

private void setClusterLevelLogging(final CacheMode cacheMode, final InfinispanCacheSetting
infinispanCacheSetting) {

if (isRemote(cacheMode) && isLoggingEnabled(infinispanCacheSetting)) {

cacheManager.addListener(new ClusterLoggingListener()); } }

private boolean isLoggingEnabled(final InfinispanCacheSetting infinispanCacheSetting) { return
infinispanCacheSetting.getClusterLogging() || infinispanCacheSetting.getLocalLogging(); }

private boolean isRemote(final CacheMode cacheMode) { return cacheMode.isDistributed()

|| cacheMode.isReplicated(); } All the documentation on how to configure an

Infinispan cache is available here.

All the beans mentioned above we get for free when adding the maven dependency mentioned before.

However, the Infinispan cache requires the following configuration values for

each cache that is provided;

ipf.caching.infinispan.settings.”${cache_name}”.cache-mode=[CacheMode]

ipf.caching.infinispan.settings.”${cache_name}”.timeout=[Duration]
ipf.caching.infinispan.settings.”${cache_na

size=[Long] ipf.caching.infinispan.settings.”${cache_name}”.cluster-logging=[Boolean]

ipf.caching.infinispan.settings.”${cache_name}”.local-logging=[Boolean]

cache_name - name of the cache being used

cache-mode - Infinispan cache managers can create and control multiple caches that use different modes. For example, you can use the same cache manager for local caches, distributed caches, and caches with invalidation mode.

timeout - duration cache will remain in memory active before being evicted.

max-size - maximum cache size before the cache evicts entries that are less likely to be used again

cluster-logging - instantiates a ClusterCacheLoggingListener

local-logging - instantiates a LocalCacheLoggingListener

An Example:

ipf.caching.infinispan.settings.cache1.cache-mode=REPL_ASYNC
ipf.caching.infinispan.settings.cache1.timeou

ipf.caching.infinispan.settings.cache1.max-size=15000
ipf.caching.infinispan.settings.paymentdata.cluster-logging=true

ipf.caching.infinispan.settings.payment-data.locallogging=true cache-mode - can be set to one of following:

LOCAL - Data is not replicated

REPL_ASYNC - Data replicated asynchronously

REPL_SYNC - Data replicated synchronously

DIST_SYNC

DIST_ASYNC

In case the cache mode is distributed or replicated, the following additional

configuration is required:

ipf.caching.infinispan.settings.”${cache_name}”.fetch-in-memory-state=[Boolean]

ipf.caching.infinispan.settings.”${cache_name}”.await-initial-state-transfer=[Boolean]

ipf.caching.infinispan.settings.”${cache_name}”.state-transfer-timeout=[Duration]

More detail on the fields:

fetch-in-memory-state - If true, the cache will fetch data from the neighboring

caches when it starts up, so the cache starts ’warm’, although it will impact

startup time. In distributed mode, state is transferred between running caches

as well, as the ownership of keys changes (e.g. because a cache left the cluster).

Disabling this setting means a key will sometimes have less than numOwner

owners.

await-initial-state-transfer - If true, this will cause the first call to method CacheManager.getCache() on the joiner node to block and wait until the joining is

complete and the cache has finished receiving state from neighboring caches (if

fetchInMemoryState is enabled). This option applies to distributed and replicated caches only and is enabled by default. Please note that setting this to

false will make the cache object available immediately but any access to keys

that should be available locally but are not yet transferred will actually cause a (transparent) remote access. While this will not have any impact on the logic of your application it might impact performance.

state-transfer-timeout - This is the maximum amount of time - in milliseconds - to wait for state from neighboring caches, before throwing an exception and aborting startup.

An Example:

ipf.caching.infinispan.settings.cache1.fetch-in-memory-state=true
ipf.caching.infinispan.settings.cache1.awaitinitial-state-transfer=true
ipf.caching.infinispan.settings.cache1.state-transfertimeout=6m Infinispan Implementation We get the CacheFactory bean for free

from ipf-cache-infinispan module and by enabling infinispan caching.

@Bean(name = ”infinispanCacheFactory”) CacheFactory<?, ?>
infinispanCacheFactory(InfinispanCacheProvider infinispanCacheProvider,

CacheLogger<Object, Object> cacheLogger) { return new
InfinispanCacheFactory(infinispanCacheProvider, cacheLogger); } Then, you just need to use the

CacheFactory to create either an AsyncCacheAdapter:

@Bean(name

=

”paymentInfinispanDataCacheAdapter1”)

AsyncCacheAdapter<Object, Object> paymentInfinispanDataCacheAdapter1(CacheFactory<Object,

Object> infinispanCacheFactory) { return
infinispanCacheFactory.asyncCreateCacheAdapter(”cache1”);

}

---------- Getting Started

Modules There are 3 separate modules:

ipf-cache-api - all the API contracts

ipf-cache-caffeine - using spring backed cache provides caffeine backed implementation

ipf-cache-infinispan - provides our implementation around famous InfiniSpan

caching technology.

Dependency Declare the api dependency if a custom implementation is to be

used.

<dependency>

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

<artifactId>ipf-cache-api</artifactId> <version>${project-version}</version>

</dependency> If you want a pre-prepared Caffeine flavour then simply add

the following;

<dependency>

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

<artifactId>ipf-cache-caffeine</artifactId> <version>${project-version}</version>

</dependency> If you want a pre-prepared InfiniSpan flavour then simply add

the following;

<dependency>

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

<artifactId>ipf-cache-infinispan</artifactId> <version>${project-version}</version>

</dependency>

IPF Persistent Scheduler

Introduction IPF's Persistent Scheduler allows you to schedule jobs of any kind.

It is based on the Quartz scheduler, and works with cron expressions in combination with calendars in order to execute one-time or recurrent jobs. It also

features a persistence layer whose main role is to keep single source of truth with

regards to the definition of tasks and also persists job history in an append-only

journal.

The Scheduler also features failsafes such as a rescheduling module which runs

at startup and restores all the scheduled jobs back into Quartz after a previous

failure. To use the Scheduler, start with SchedulingModuleInterface.

Finally, it is designed to be run in a cluster. It uses Akka Cluster and Cluster

Singleton to ensure that jobs are only scheduled in one place in the cluster and

can survive any number of node failures (including a total outage).

The architecture for the scheduler is shown below:

service architecture ---

--- Concepts The Action

Helper

Calendars

Failed Jobs

Job Specification and Status

Scheduling Module

Scheduling Status

--- The Action Helper This can be thought of as

the engine that receives the scheduled commands to run. It awaits messages

from the scheduler which - when the time comes to run a job as per the

cron expression - will hand the command to the SchedulingHelper and being

executing

Interface The SchedulingHelper has two methods:

execute(String, Command): Run a task This is what the scheduler will call at

the right time. The ”String” is the triggerIdentifier from the JobSpecification,

and the Command is the triggerCommand from the same specification.

supports(Command): Does this SchedulingHelper support this command?

When defining multiple SchedulingHelpers, the scheduler needs to know which

SchedulingHelper can support which command. ---------- ---------- ---------- Calendars A Calendar is a mechanism that

is used for excluding blocks of time regardless of the time specification that

was supplied. For example, if a job is set to run once a day, it is possible to

augment this specification with a calendar to say that it should only run once

a day on weekdays, for example. More information is available here.

If a calendar is not provided as part of the JobSpecification, then the default

calendar is used. To define a default calendar, the configuration file has

to contain ipf.persistent-scheduler.quartz.calendars.default and ipf.persistentscheduler.quartz.calendars as in example below.

If no default calendar is configured, then the scheduler will not use calendars

and the cron expression will be honoured without exclusions.

ipf.persistent-scheduler.quartz.calendars.default = ”DefaultCalendar”

ipf.persistent-scheduler { quartz { calendars { DefaultCalendar { type = Annual

description = ”Default calendar” exclude-dates = [”12-25”, ”01-01”] } } } }

Calendar Types Quartz calendars can be of the following types:

Calendar type name Description Example Daily

Exclude blocks of time from a day with a timezone. Use UTC or the list of

timezones here.

exclude { start-time = ”03:00” end-time = ”05:00” } timezone = UTC Monthly

Exclude days from a month

exclude-days = [1, 3, 5, 7]

Weekly

Exclude days of the week from a week. Days are 1-indexed and start on Sunday,

i.e. Sunday = 1, Monday = 2, etc.

exclude-days = [1, 7] //will exclude the weekend

Cron

Exclude by a custom cron expression

exclude-expression = * * 0-7,18-23 ? * *

Holiday

Exclude explicit dates (useful for moveable feasts like Easter or UK bank holidays) in ISO 8601 yyyy-MM-dd format

exclude-dates = [”2024-03-31”, ”2025-04-20”] //excludes Easter 2024 and 2025

Annual

10

Exclude calendar dates from each year in MM-DD format

exclude-dates = [”01-01”, ”25-12”] //excludes New Year’s Day and Christmas

Day every year

A job can only use one calendar.

Failed Jobs A failed job is defined as a job that matches the

following criteria:

It has missed its regular execution slot with its specification unchanged, OR;

It was scheduled to be in the past

Its specification was updated in a way to stop any further execution

The Failed Jobs Processor When the system starts, it schedules an internal job

to find other failed scheduled jobs. This job behaves like any other scheduled

job, but has a special task of identifying other jobs that have failed according

to the above criteria, and processes them (see below for details).

The frequency of execution is configurable, and it can be set using the following

properties file:

package com.iconsolutions.ipf.core.platform.scheduler.persistent.job;

import lombok.Data; import org.springframework.boot.context.properties.ConfigurationProperties;

@Data @ConfigurationProperties(prefix = ”ipf.persistent.scheduler.processfailed-jobs”) public class ProcessFailedJobsProperties { private Boolean active;

private String cronExpression; } active: a flag to set if the job is active or not

cronExpression: a cron expression to describe the frequency the job will run.

For help with building a cron expression, use an online cron expression builder

such as this one.

The example below sets it to run once a day at 0:15:59:

ipf.persistent.scheduler.process-failed-jobs { active = true cron-expression = ”59

15 0 */1 ? *” } Being notified of failed jobs When configuring a JobSpecification,

it is possible to be notified of failures for that specific job by specifying the failure

identifier and command:

JobSpecificationDto.builder() .jobRequestor(JOB_REQUESTOR) .triggerCommand(TEST_COMMAND)

.schedulingSpecification(cronExpression)

.failureIdentifier(failureIdentifier) .failureCommand(failureCommand) .build();

If this job has failed since the last checkpoint, then the relevant SchedulingHelper will be notified of this. Note that the SchedulingHelper will need to

have the supports method updated to support the failure command too.

---11

---------- Job Specification

and Status This page explains the job specification and status:

jobspec status Job Specification The job specification is the way to tell the IPF

Persistent Scheduler how to run a scheduled job. It consists of the following

parts:

Name Mandatory? Description jobSpecificationKey

Yes

a unique identifier for this job

jobRequestor

Yes

A unique identifier for the requestor (a requestor can have multiple jobs)

schedulingSpecification

Yes

The cron expression for this job

triggerCommand

Yes

The Command to send to the SchedulingHelper at the scheduled time(s)

triggerIdentifier

Yes

An ID to use to send to the SchedulingHelper at the scheduled time(s)

failureCommand

Yes

When a failed execution is detected, the SchedulingHelper will receive this message.

failureIdentifier

Yes

An ID to use to send to the SchedulingHelper at the scheduled time(s)

calendar

No

Calendar to use for this job (see Calendars)

Execution Status The JobExecutionStatus collection is linked to the JobSpecification collection by the jobSpecificationId field.

The JobExecutionStatus collection is going to change more frequently, because

the status of a job changes more frequently than its specification.

It contains the jobSpecificationId, the executionStatus and the updatedTime. It

is an append only collection and in order to get the current job execution status

we just need to order by updatedTime and get the latest more recent entry.

Status definitions The different statuses are:

SCHEDULED: Is scheduled to run (and may have run in the past)

TRIGGERED: Has already run and will not run again

CANCELED: Was previously SCHEDULED but was not triggred

FAILED: Failed to execute

Scheduling Module The scheduling

module is how to interact with the IPF Persistent scheduler. It offers typical

scheduling operations that you would expect to see on such a tool. They are

documented below.

scheduleJob and updateJob - [re]schedule a job This method takes a JobSpecificationDto, persists the job specification and enqueues the job to run in Quartz

with the relevant scheduler cron expression (”specification”) and calendar if

present.

Note that if the key for this job already exists, then the existing job will be

updated to this new specification.

cancelJob - cancel a job Takes the JobSpecificationKey only, and any future

executions of this job - if any - will not be executed. Please note that the

execution history of the job will be retained.

findJobById - get a job’s latest status Takes the JobSpecificationKey of a job

and returns its specification and latest run status. ---

--- Scheduling Status The purpose of this page is to clarify the possible

execution statuses for the two different types of jobs as well how these statuses

should be handled in the context of rehydration as well as failure handling.

Execution Statuses are stored in a separate collection which is keyed on JobId

and Timestamp, with the JobId corresponding to a valid Job Specification in the Job Specification collection. Determining if a job is one time or recurrent, can be done looking at the schedulingSpecification (in Job Specification)
One Time Scheduled Job For a one time scheduled job, the following Execution Statuses apply for a one time scheduled job and the transition between the statuses is as follows:

one time scheduled Rehydration - for one time scheduled job Rehydration occurs

in the event of a failure which causes Quartz to restart. Since there is no

persistence in Quartz, it needs to be rehydrated with jobs which have been

persisted in the Job Repository.

Candidate statuses for rehydration would be SCHEDULED

Job Failure Handling - for one time scheduled job When determining if jobs

have failed we would need to check for the following statuses: SCHEDULED

Recurrent Scheduled Job The following Execution Statuses apply for a recurrent

scheduled job, the following Execution Statuses apply for a recurrent between

the statuses is as follows:

recurrent scheduled Rehydration - for recurrent scheduled job Rehydration occurs in the event of a failure which causes Quartz to restart. Since there is

no persistence in Quartz, it needs to be rehydrated with jobs which have been

persisted in the Job Repository.

Candidate statuses for rehydration would be :

SCHEDULED

TRIGGERED

FAILED

Job Failure Handling - for recurrent scheduled job When determining if jobs

have failed we would need to check for the following statuses:

SCHEDULED

TRIGGERED

And also the updatedTime from the JobExecutionStatus collection

In addition to considering the Execution Status we also need to consider the

following from JobSpecification

latestUpdate

schedulingSpecification

The latestUpdate should be compared to the updatedTime in the JobExecutionStatus if latestUpdate > updatedTime then the schedulingSpecification should

be considered as pending execution, you can use the getNextValidTimeAfter

using latestUpdate comparing to see if this is consistent with the expected execution.

Otherwise, it should be considered as an existing job i.e. the updateTime should

be compared to the cronExpression - getNextValidTimeAfter this in conjunction with specifying calling the method against the current time can be used to determine failed job execution by comparing the two dates.

For example:

//Successful job //Daily expression with last job run time of 2022-1118T08:24:00 // ”Now”: 2022-11-18T08:26:00 @Test void shouldHaveAfterNowAndAfterLastJobRunResultSame() throws ParseException { String

dailyExpressionString = ”0 24 08 * * ? *”; CronExpression dailyExpresssionJobRun = new CronExpression(dailyExpressionString); LocalDateTime jobLastRunTime = LocalDateTime.parse(”2022-11-18T08:24:00”); LocalDateTime

nowTime = LocalDateTime.parse(”2022-11-18T08:26:00”); Date nowDate

= Date.from(nowTime.atZone(ZoneId.systemDefault()).toInstant());

Date

nextValidTimeAfterNow = dailyExpresssionJobRun.getNextValidTimeAfter(nowDate);

Date jobLastRunDate = Date.from(jobLastRunTime.atZone(ZoneId.systemDefault()).toInstant());

Date nextValidTimeAfterJobRun = dailyExpresssionJobRun.getNextValidTimeAfter(jobLastRunDate);

String nextValidTimeAfterJobRunString = nextValidTimeAfterJobRun.toString();

System.out.println(”next valid time after last run: ” + nextValidTimeAfterJobRunString);

String nextValidTimeAfterNowString = nextValidTimeAfterNow.toString();

System.out.println(”next valid time after now: ” + nextValidTimeAfterNowString); assertThat(nextValidTimeAfterJobRunString, is(nextValidTimeAfterNowString));

// next valid time after last run: Sat Nov 19 08:24:00 GMT 2022 // next valid

time after now: Sat Nov 19 08:24:00 GMT 2022 }

//Failed job //Daily expression with last job run time of 2022-11-17T08:24:00

// ”Now”: 2022-11-18T08:26:00 @Test void shouldHaveAfterNowAndAfterLastJobRunResultDifferent() throws ParseException { String dailyExpressionString = ”0 24 08 * * ? *”; CronExpression dailyExpresssionJobRun

= new CronExpression(dailyExpressionString); LocalDateTime jobLastRunTime = LocalDateTime.parse(”2022-11-17T08:24:00”); LocalDateTime

nowTime = LocalDateTime.parse(”2022-11-18T08:26:00”); Date nowDate

= Date.from(nowTime.atZone(ZoneId.systemDefault()).toInstant());

Date

nextValidTimeAfterNow = dailyExpresssionJobRun.getNextValidTimeAfter(nowDate);

Date jobLastRunDate = Date.from(jobLastRunTime.atZone(ZoneId.systemDefault()).toInstant());

Date nextValidTimeAfterJobRun = dailyExpresssionJobRun.getNextValidTimeAfter(jobLastRunDate);

String nextValidTimeAfterJobRunString = nextValidTimeAfterJobRun.toString();

System.out.println(”next valid time after last run: ” + nextValidTimeAfterJobRunString);

String nextValidTimeAfterNowString = nextValidTimeAfterNow.toString();

System.out.println(”next valid time after now: ” + nextValidTimeAfterNowString);
assertThat(nextValidTimeAfterJobRunString, not(is(nextValidTimeAfterNowString)));

// next valid time after last run: Fri Nov 18 08:24:00 GMT 2022 // next valid

time after now: Sat Nov 19 08:24:00 GMT 2022 }

---------- Features Scheduling Metrics ----------

---------- Scheduling Metrics

This page discusses the metrics exposed by IPF’s Persistent Scheduler:

Metric Reference Metric Type Description scheduling_registered_jobs_total

Counter

Total number of new scheduled jobs registered with the scheduling module. Each

unique job ID counts only once irrespective of number of updates or number of

re-hydrations

scheduling_updated_jobs_total

Counter

Total number of updates to existing scheduled jobs. Each update to the same

job ID increments the counter

scheduling_cancelled_jobs_total

Counter

Total number of explicitly cancelled jobs

scheduling_failed_jobs_total

Counter

Total number of failed jobs due to expiration

scheduling_rehydrations_total

Counter

Total number of re-hydrations due to crash of scheduling module

Enabling Persistent Scheduler Metrics Metrics are enabled by default and will

be present alongside other IPF metrics for connectors, flows, etc. No extra

work is required to enable Persistent Scheduler metrics. ----------
---
---------- Getting

Started Getting started guides for starting with the IPF Persistent Scheduler.

Scheduling Your First Job ----------

Scheduling Your First Job There are

a few things to set up to start scheduling your first IPF Scheduler job. We'll

do them in the most sensible order possible.

You will create a command, a JobSpecification that uses that command to define

the job to run, and an SchedulingHelper which gets given that command at the

scheduled run time(s).

Step 0: Add dependency You will need to add this to pom.xml:

<dependency>

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

<artifactId>scheduler-core</artifactId> <version>${ipf-persistent-scheduler.version}</version>

</dependency> To find the latest version, you can use this Nexus query.

Step 1: Create a command This is a command that will be sent to your that

has to extend Command.

Here's an example of one:

public static class MyCommand implements Command {

@Override public CommandId

dId.from("A|B|C"); }

getCommandId()

{

return

Comman-

@Override public Instant getCreatedAt() { return Instant.now(); } } Step 2:

Create an SchedulingHelper and define it as a Spring bean This is the thing

that will run your job at a specific time, with the given command from step 1:

public static class MySchedulingHelper implements SchedulingHelper { @Override public CompletionStage<Void> execute(String id, Command command) {

//do some really important work here that can possibly take a long time...or

not? log.info("Look I'm being scheduled! The ID was: {}", id); return CompletableFuture.completedFuture(null); }

@Override public boolean supports(Command command) { return command

instanceof MyCommand; } } You will also need to define it as a bean:

@Bean public SchedulingHelper mySchedulingHelper() { return new MySchedulingHelper(); } Step 3: Schedule the job Now we tell the SchedulingModuleInterface to schedule our job with our command at a specific time.

In the below example we are running our job every 5 seconds.

public void scheduleJob() { schedulingModuleInterface.scheduleJob(JobSpecificationDto.builder()

.jobRequestor("test-requestor") .jobSpecificationKey(new JobSpecificationKeyDto("myspecial-job-wow")) .triggerCommand(new MyCommand()) .triggerIdentifier("mytrigger-id") .schedulingSpecification("*/5 * * ? * *") .build()); } Note that

if you want a non-repeating job, you can use singleSchedule and pass in a Calendar instance representing the desired trigger time, instead of supplying a cron-style schedulingSpecification.

Step 4: Run it! If we run this application we can see that every 5 seconds our log message is printed out:

28-02-2023 15:29:00.002 [DefaultQuartzScheduler_Worker-1] INFO c.i.i.c.p.s.persistent.DocsExamples.execute

- Look I'm being scheduled! The ID was: my-trigger-id 28-02-2023 15:29:05.001

[DefaultQuartzScheduler_Worker-2] INFO c.i.i.c.p.s.persistent.DocsExamples.execute

- Look I'm being scheduled! The ID was: my-trigger-id 28-02-2023 15:29:10.000

[DefaultQuartzScheduler_Worker-3] INFO c.i.i.c.p.s.persistent.DocsExamples.execute

- Look I'm being scheduled! The ID was: my-trigger-id 28-02-2023 15:29:15.001

[DefaultQuartzScheduler_Worker-4] INFO c.i.i.c.p.s.persistent.DocsExamples.execute

- Look I'm being scheduled! The ID was: my-trigger-id 28-02-2023 15:29:20.000

[DefaultQuartzScheduler_Worker-5] INFO c.i.i.c.p.s.persistent.DocsExamples.execute

- Look I'm being scheduled! The ID was: my-trigger-id

Dynamic Processing Settings Background

Dynamic Processing Settings provides a platform to manage configurable settings and their lifecycle. This platform provides for defining, maintaining and exposing configurable settings that can be referenced from an external application such as IPF.

The initial use case which has been used to prove out the framework in the first instance is CSM Reachability.

High Level Project Structure The following diagram shows an example of the high level structure of components involved in Dynamic Processing Settings.

structure Platform Projects: Project Description file-ingestion-service

Contains the required infrastructure to consume settings from a source e.g. a file and propagate the consumed settings through to the setting management

API via a Process Manager.

setting-domain

Contains generic setting domain and model objects e.g. CreateSetting, UpdateSetting, Setting

setting-management

Contains the API framework which is used to manage the settings (CRUD API)

setting-workflow

Contains the MPS generated artefacts to support lifecycle management via

Event Sourced Behaviours

setting-catalogue

Specific settings which have been defined to be managed by the Dynamic Processing Settings framework. Contains supporting infrastructure such as read

side models and settings definitions

CSM Reachability Solution Projects: The CSM Reachability Solution leverages

the Platform Projects and defines additional solution specific projects

Project Description csm-reachability

Pulls in the relevant platform projects, and contains the validate csm reachability business service, which invokes the setting management APIs in order to

determine CSM reachability given a specific input

csm-reachability-app

Wrapper project which pulls in all the relevant modules and runs as a single

sprint boot application. It also builds a docker image of the same.

csm-reachability-service

Contains the orchestration service - csm-reachability-service

csm-reachability-setting-management

Leverages setting-management and specific setting definitions relevant to the

CSM Reachability Solution

participant-file-handling

Leverages file-ingestion-service and specific setting definitions relevant to the

CSM Reachability Solution

csm-reachability-e2e-test

Verifies the CSM Reachability App assembles and runs successfully as a docker

image

Documentation How to create a project based on Dynamic Settings - An

overview of modules needed to assemble an application built on top of

dynamic-settings-workflow

Creating a Setting - Example of adding a setting to the dynamic processing

settings framework

Workflow Documentation - An overview of all the concepts relating to the dynamic settings workflow

---

---------- Features Settings

Model ----------

---------- Settings Model This section describes

the Dynamic settings flow and its attributes.

Flows Dynamic Settings Flow Properties Flow Name:

Dynamic Settings

Version:

0

Description:

This is a placeholder for the flow solution.

Global State Set:

Default Global States

Flow Graph dynamicsettings settings flow graph States Name Description

Global State Is Terminal Inactive Approval Pending

Approval Pending for Setting currently inactive

none

No

Active

Active Setting state

none

No

Active Approval Pending

Approval pending state for active setting after update

none

No

Delete Approval Pending

Delete Approval Pending for setting deletion

none

No

Terminal State

State added because it’s required to have terminal state

none

Yes

Events Name Description Business Data Flow Initiated

The flow has been successfully started.

Provisional Setting

Requires Approval

Approval Received

Raised after setting approved

Active Setting

Provisional Setting

Requires Approval

Approval Rejected

Emited after approval rejected

Provisional Setting

Requires Approval

Setting Deactivated

Setting moved to Inactive state

Active Setting

Requires Approval

Setting Created

Setting has been created

Active Setting

Requires Approval

Setting Created Needs Approval

Setting has been created but needs approval

Provisional Setting

Requires Approval

Setting Updated

Setting has been updated

Provisional Setting

Active Setting

Setting Updated Requires Approval

Update requested requires approval

Provisional Setting

Setting Deactivated Requires Approval

Deactivation request requires approval

Active Setting

Functions No aggregate functions defined. Input Behaviour Input Response

Code Event Selection Initiate Dynamic Settings

none

Flow Initiated

Update Setting

none

Decision: Needs Approval

On On*YES*raiseSetting Updated Requires Approval On*NO*raiseSetting Updated

Approval Response

Accepted

Approval Received

Approval Response

Rejected

Approval Rejected

Deactivate Setting

none

Decision: Needs Approval

On On*YES*raiseSetting Deactivated Requires Approval On*NO*raiseSetting

Deactivated

Create Setting

none

Decision: Needs Approval

On On*NO*raiseSetting Created On*YES*raiseSetting Created Needs Approval

Event Behaviour Given State Criteria Events New State Perform Actions Initial

On

Flow Initiated

Inactive Approval Pending

Initial

On

Setting Created

Active

Initial

On

Setting Created Needs Approval

Inactive Approval Pending

Call Request: Approve Setting

Inactive Approval Pending

On

Approval Received

Active

Inactive Approval Pending

On

Approval Rejected

Initial

Active

On

Setting Updated Requires Approval

Active Approval Pending

Call Request: Approve Setting

Active

On

Setting Updated

Active

Active Approval Pending

On

Approval Received

Active

Active Approval Pending

On

Approval Rejected

Active

Delete Approval Pending

On

Approval Rejected

Active

Delete Approval Pending

On

Approval Received

Initial

Active

On

Setting Deactivated

Initial

Active

On

Setting Deactivated Requires Approval

Delete Approval Pending

Call Request: Approve Setting

Terminal State

On

Setting Deactivated

Terminal State

Initial

On

Setting Updated

Active

Initial

On

Setting Updated Requires Approval

Inactive Approval Pending

Call Request: Approve Setting

Flow BDD DynamicSettings-Aborted.story Meta:

Narrative: Dynamic Settings This is a placeholder for the flow solution. Paths

ending in state: Aborted DynamicSettings-TerminalState.story Meta:

Narrative: Dynamic Settings This is a placeholder for the flow solution. Paths

ending in state: Terminal State External Domains Settings Settings workflow

domain

Requests No requests defined. Notifications No notifications defined. Instructions Name Description Business Data Deactivate Setting

deactivate

Provisional Setting

Active Setting

24

Requires Approval

Update Setting

Update setting

Active Setting

Provisional Setting

Requires Approval

Create Setting

Initial command

Provisional Setting

Active Setting

Requires Approval

Approver This domain is used to handle the approval process

Requests Name Description Business Data Response Approve Setting

Submit this setting for approval

Active Setting

Provisional Setting

Name:

Approval Response

Description:

description

Business Data:

Active Setting

Provisional Setting

Requires Approval

ResponseCodes:

AcceptOrReject

ReasonCodes:

none

Completing:

Yes

Notifications No notifications defined. Instructions No instructions defined. Initiation A system generation domain representation of this domain to allow initiate from external sources.

Requests No requests defined. Notifications No notifications defined. Instructions Name Description Business Data Initiate Dynamic Settings

Flow Initiation

Provisional Setting

Requires Approval

Supporting Libraries Business Data Libraries Business Data Library Business

Data Library for Dynamic Processing Settings

Name Description Data Type Data Category Active Setting

Currently Active Setting

com.iconsolutions.ipf.dynamicsettings.domain.Setting

Provisional Setting

Setting to be approved

com.iconsolutions.ipf.dynamicsettings.domain.Setting

Requires Approval

Requires Approval

java.lang.Boolean

Common Event Libraries No event libraries have been defined Decision Libraries

Decision Library No description provided.

Name Description Business Data Outcomes Needs Approval

Does this action need approval

Requires Approval

YES

NO

Domain Function Libraries No domain functions have been defined. Response

Code Libraries No response codes have been defined. Reason Code Libraries

No reason codes have been defined. Placeholder Libraries No placeholders have

been defined. Global States No global states have been defined.

---------- How to guides Create

a Project Based on Dynamic Settings Workflow

Create Settings ----------

Create a Project Based on

Dynamic Settings Workflow Dynamic settings workflow contains the building

blocks that are meant to be used as a starting point to create an application

that suits your needs. Such an application will manage the lifecycle of the

settings which you will then use depending on your use case.

Two typical reasons to manage a setting in your application are:

to expose the setting to other services

to use it as part of the business API that you want to expose (e.g. CSM

Reachability)

Creating a Setting covers details around adding a new setting.

The structure of your Application Your application will typically contain the

following modules: file-ingestion module

data-management module

one or more modules that expose business APIs

an application module that assembles the application from the previous modules

File ingestion module File ingestion module is used to feed the setting data

from a source (local directory, REST API…) through a data management API

exposed by the data-management module. This module is built on top of the

generic file-ingestion-service framework. The framework expects you to configure connectors that would ingest the settings from its source (currently supported transports are local directory/REST API).

Depending on the specification of the source, specific mappers are needed in

order to convert the settings into the canonical format.

Data management module Data management module manages the lifecycle of

each supported type of the setting. It encapsulates the write-side of the application. Module should be built on top of the setting-management framework.

Setting-management framework is a generic API framework that exposes the

data management API for each of the settings found on the classpath. You

need to provide the relevant settings from the settings-catalogue as dependencies.

Business API module(s) Depending on the use case, you may need one or more

modules that expose business APIs. Such a module would typically query read

side collections for one or more settings in order to provide a specific functionality.

CSM Reachability is a good starting point.

---27

---------- Create Settings The following is an example of how to add a

setting, in this case a CsmAgent Setting that will be managed by the platform.

You need to configure a domain project and a repository project

Domain Project Setup In order to add a setting to be managed by the Dynamic

Processing Settings Platform you need to create the following:

Setting Definition

Domain Object

Search Fields for the setting

Setting Definition Specifies how to calculate the logical unique key for the setting

and associates all the other components (domain object and search fields) to the

setting concept

@Bean SettingDefinition csmAgentSettingDefinition(final Notifier systemEventSender) { return SettingDefinition.<CsmAgent>builder() .name(”csmagent”)

.clazz(CsmAgent.class) .idFunction(setting -> setting.getProcessingEntity() +

”-” + setting.getPayload().getCsmAgentId()) .approvalFunction((requiresApproval,

persistanceId, inputSetting) -> CompletableFuture.completedStage(requiresApproval))

.searchableFields(CsmAgentSearchableFields.class) .notificationFunction(systemEventSender::notify)

.build(); } Domain Object This will be the payload of a setting object and

should contain all the relevant attributes for the setting you wish to define

@Data @Builder(toBuilder = true) public class CsmAgent { @NotNull private

String csmAgentId; private String csmAgentBic; @Size(max = 70) private

String csmAgentName; @NotNull @Size(max = 35) private String csmAgentType; @NotNull @Size(max = 15) private String csmParticipantIdentifierType;

@NotNull @Size(max = 35) private String csmAgentConnector; @Size(max

= 70) private String csmAgentConnectorAddress; @Size(min=1) @Valid private List<CsmAgentMessageStandard> csmAgentMessageStandards; private

Boolean onUsCSM; private Boolean higherParticipantLimitNotAllowed; private

Boolean instantPayments;

@Data @Builder public static class CsmAgentMessageStandard { @NotNull

@Size(max = 35) private String messageStandard; @NotNull @Size(max = 35)

private String messageStandardVersion; @NotNull private Instant activeFrom;

```
}

public boolean isHigherParticipantLimitNotAllowed() { return
BooleanUtils.isTrue(higherParticipantLimitNotAllowed); }

public boolean isOnUsCSM() { return BooleanUtils.isTrue(onUsCSM); }

public boolean isInstantPayments() { return BooleanUtils.isTrue(instantPayments);

}

} Setting class:
```

@Data @AllArgsConstructor @NoArgsConstructor @Builder public class Setting <T> implements Serializable { private String id; @Size(max = 15, min

= 1) private String processingEntity; private Instant activeFromDate; private

String source; private String status; private int version; private String createdBy;

private String rejectedBy; private String approvedBy; @JsonTypeInfo(use =

JsonTypeInfo.Id.CLASS, property = ”className”) private T payload;

@JsonIgnore public boolean isActive() { return ”ACTIVE”.equalsIgnoreCase(status);

} } Search Fields Define the fields which are searchable on the setting, in this

case the CsmAgent can be searched by CsmAgentId.

package com.iconsolutions.ipf.dynamicsettings.search;

public enum CsmAgentSettingSearchFields

CSM_AGENT_ID;

implements

SearchField

{

@Override public String getName() { return this.name(); } } In addition to the

search fields you define for the setting, all settings are searchable via CommonSearchFields (status, processingEntity, activeFrom and source)

@Data public class CommonSearchableFields implements SearchableFields { private String status; private String processingEntity; private Instant activeFrom;

private List<String> idList; @Pattern(regexp = ”import|manual”, flags = Pattern.Flag.CASE_INSENSITIVE) private String source;

public CommonSearchableFields populateFromRequest(ServerRequest serverRequest) { CommonSearchableFields commonSearchableFields = newInstance(); serverRequest.queryParam(”status”).ifPresent(commonSearchableFields::setStatus);

serverRequest.queryParam(”processingEntity”).ifPresent(commonSearchableFields::setProcessingEntity);

serverRequest.queryParam(”source”).ifPresent(commonSearchableFields::setSource);

serverRequest.queryParam(”activeFrom”).ifPresent(activeFrom1

->

commonSearchableFields.setActiveFrom(Instant.parse(activeFrom1)));

return

commonSearchableFields; }

public CommonSearchableFields newInstance() { return new CommonSearchableFields(); }

@Override public List<Criterion> criteria() { final List<Criterion> criteria =

new ArrayList<>();

if (status != null) { criteria.add(Criterion.equalTo(SettingSearchFields.STATUS,

status)); } else { criteria.add(Criterion.notEqualTo(SettingSearchFields.STATUS,

”INITIAL”)); }

if (activeFrom != null) { criteria.add(Criterion.gte(SettingSearchFields.ACTIVE_FROM,

activeFrom)); }

if (source != null) { criteria.add(Criterion.equalTo(SettingSearchFields.SOURCE,

source)); }

if (processingEntity != null) { criteria.add(Criterion.equalTo(SettingSearchFields.PROCESSING_ENTITY,

processingEntity)); }

if(idList != null) { criteria.add(Criterion.in(SettingSearchFields.ID, idList)); }

return criteria; } } The below tells the framework how to extract the search

fields from the requests received

@Data public class CsmAgentSearchableFields extends CommonSearchableFields { private String csmAgentId;

@Override public CsmAgentSearchableFields populateFromRequest(ServerRequest

serverRequest) { CsmAgentSearchableFields searchableFields = (CsmAgentSearchableFields) super.populateFromRequest(serverRequest);
serverRequest.queryParam(”csmAgentId”).ifPresent(searchableFields::setCsmAgentId);

return searchableFields; }

@Override public CsmAgentSearchableFields newInstance() { return new CsmAgentSearchableFields(); }

@Override public List<Criterion> criteria() { final List<Criterion> criteria =

new ArrayList<>(super.criteria());

if (csmAgentId != null) { criteria.add(equalTo(CsmAgentSettingSearchFields.CSM_AGENT_ID,

csmAgentId)); }

return criteria; } } You also need to update the search fields map which specifies

the path to the searchable field from the perspective of a setting

@PostConstruct void updateSearchFieldsMap() {
settingSearchFieldsMapper.putMapping(CsmAgentSettingSearchFields.CSM_AGENT_ID.getName(),

”payload.csmAgentId”); } Repository Project Setup Additionally, the following

read side infrastructure needs to be defined:

Repository

ModelEntity

ModelEntityProvider

IndexInitialiser

Repository Repository, which extends ReactiveCRUDRepository and exposes

the query functionality of the setting stored in the database

public interface CsmAgentSettingsRepository

tory<CsmAgentSettings> {

extends

SettingReposi-

String CSMAGENT = ”csmagent-”;

Flux<CsmAgentSettings> findAll(Sort sort);

@Override default boolean supports(String id) { return id.toLowerCase().contains(CSMAGENT);

} } ModelEntity ModelEntity, defines how the setting will be represented in

the DB and also defines how the payload for the settings is created/updated

@Document(collection = ”settings-csm-agent”) @Data public class CsmAgentSettings extends MongoSettingReadModelEntity<CsmAgent> {

@Override protected Supplier<CsmAgent> payloadCreator() { return () -> CsmAgent.builder().build(); }

@Override protected BiFunction<Event, CsmAgent, CsmAgent> payloadUpdater() { return (event, csmAgent) -> csmAgent; }

} ModelEntityProvider ModelEntityProvider, is responsible for creating the appropriate ModelEntity, based on the identifier that is input

@Component public class CsmAgentMongoSettingModelEntityProvider implements MongoSettingModelEntityProvider { // ”-” suffix added to avoid partial

match e.g. csmagent matching csmagentcurrency private static final String CSMAGENT = ”csmagent-”;

@Override public MongoSettingReadModelEntity provide() { return new CsmAgentSettings(); }

@Override public Class<? extends MongoSettingReadModelEntity> getEntityClazz() { return CsmAgentSettings.class; }

@Override public boolean supports(String id) { return id.toLowerCase().contains(CSMAGENT);

} } IndexInitialiser Index Initialiser, is responsible for creating indexes on the

collection

@Slf4j @AllArgsConstructor public class CsmAgentMongoSettingRecordIndexInitialiser {

private static final String STATUS = ”status”; private static final String

PROCESSING_ENTITY = ”processingEntity”; private static final String

PAYLOAD_CSM_AGENT_ID = ”payload.csmAgentId”; private static

final String COLLECTION_NAME = ”CsmAgentSettings”; private final

ReactiveMongoTemplate reactiveMongoTemplate; private final RepositoryRetryProvider repositoryRetryProvider;

@EventListener(ContextRefreshedEvent.class) public void initialise() {

log.info(”creating indexes”); final ReactiveIndexOperations indexOperations =

reactiveMongoTemplate .indexOps(CsmAgentSettings.class);

createIndex(indexOperations, STATUS, COLLECTION_NAME, repositoryRetryProvider); createIndex(indexOperations, PROCESSING_ENTITY,

COLLECTION_NAME, repositoryRetryProvider); createIndex(indexOperations,

PAYLOAD_CSM_AGENT_ID, COLLECTION_NAME, repositoryRetryProvider);

}

} JAVA copy icon Copied!

IPF Bulker ”The ability to aggregate payment

instructions into one or more bulk files, each containing multiple transactions,

potentially between multiple debtors and creditors.”

If an IPF implementation needs to store a collection of data elements in a structured file format, the Bulker will provide this functionality.

The individual features and characteristics of the Bulker are covered in the Concepts page.

Concepts The Bulker is responsible for bringing together individual transactions or components, and acts initially as a temporary storage area where the main IPF flow can store elements that will eventually end up in the structured file. When instructed, the Bulker will stream each item, in a preconfigured order, to a file at a predefined location.

Along with the items themselves, the Bulker can use pre-configured templates to include headers, footers and intermediary constructs in the final file.

The trigger for streaming the stored elements to file can be a ”manual” command sent from the IPF Implementation that set-up the Bulk and provided the elements, or, it can come automatically from a variety of sources. Automatic finalisation can be time based in Scheduled or periodic intervals, or it could be based on the characteristics of the Bulk such as number of elements or estimated total size of the output file. The method of finalisation is defined at the point the Bulk is first created.

In a situation when elements will continue to be generated and need to be stored, even after a Bulk has been finalised and a file created, the Bulk can be given the ”Recurrance” characteristic. In this situation the closing of one Bulk will automatically trigger the creation of a new Bulk, with identical configuration, for subsequent elements to be added to.

The structure of the output file needs to be provided in a template in the configuration of the Bulk. This configuration will instruct the Bulker where elements need to end up in the final documentation and any ordering that may be necessary.

As well as the ability to build Bulk files, the Bulker provides housekeeping features to ensure memory and storage is freed after files are produced and also a host of enquiry API’s in order to be able to find out the size, status and

structure of the Bulk while it grows. ----------

---------- Features This section outlines all the features

that are needed, and should be considered, when creating and configuring a

Bulk

Akka Bulker Aggregate

Bulk Aggregate

Bulk Element Adder* Bulk Initiation

Bulk Initiation

Bulker Finalisation

Component Parser

New Bulk Notification

Default

Bulk Outputstream Provider

Delivered Bulk Notification

Bulk Auto Closed Notification ---------- Akka

Bulker Aggregate Akka Bulker Aggregate is a BulkAggregate implementation

which uses akka event sourcing for ingesting bulk components and perform

validation.

Maven Dependency To use the Akka Bulk Aggregate, the following dependency

must be provided, with a version matching ipf-bulker to ensure compatibility.

<dependency> <groupId>com.iconsolutions.ipf.bulk</groupId> <artifactId>ipfbulker-aggregate-akka</artifactId> <version>${ipf-bulker.version}</version>

</dependency>

---------- Bulk Aggregate Purpose

of Bulker Aggregate is to create single or recurring bulks, validating and

aggregating components that will be used by the Bulk Producer to create the

bulk file.

Single Bulk Aggregate Single Bulk aggregate is a component whose purpose is to

create a new bulk, aggregate bulk components and make sure those components

are valid. After all bulk components are collected by the aggregate, bulking can

be started.

Interface The BulkAggregate interface is defined as follows.

public interface BulkAggregate {

CompletionStage<BulkIdResponse> createBulk(CreateBulkCommand command);

CompletionStage<BulkComponentIdResponse> addComponent(AddComponentCommand

command);

CompletionStage<Response> updateComponent(UpdateComponentCommand

command);

CompletionStage<Response> removeComponent(RemoveComponentCommand

command);

CompletionStage<Response> closeBulk(CloseBulkCommand command);

CompletionStage<Response> openBulk(OpenBulkCommand command);

CompletionStage<Response> finaliseBulk(FinaliseBulkCommand command);

CompletionStage<BulkReportResponse> getBulkReport(GetBulkReportCommand

command);

CompletionStage<Response> terminateBulk(TerminateBulkCommand command); } createBulk is used to create a new bulk. Returns BulkIdResponse

with BulkId when bulk creation was successful

addComponent is used to add new component to the bulk aggregate, first component needs to be root component. Returns BulkComponentIdResponse with

SUCCESS result and BulkComponentId when adding component is successful

updateComponent updates the component that which is present in the aggregate

with the new content. Returns Response with SUCCESS result if the component

update is successful

removeComponent deletes the component which is present in the aggregate.

Returns Response with SUCCESS result if the component removal is successful

closeBulk Closes the aggregate, preventing client to remove and add new components to the aggregate, but allows updating components which are in the

aggregate. Returns Response with SUCCESS result if the bulk is closed

openBulk Re-opens the aggregate which is closed. Returns Response with SUCCESS result if the bulk is opened

finaliseBulk The point of no return, this is the signal to start processing data in

the aggregate and perform bulking. Returns Response with SUCCESS result if

the bulk is finalised

terminateBulk remove a whole bulk which is present in aggregate. Returns

Response with SUCCESS result if the bulk removal is successful

Recurring Bulk Aggregate Recurring Bulk Aggregate is a component for creating

recurring bulk. It is responsible for delegating components to currently open

Bulk Aggregate, and when that Bulk Aggregate gets finalised for any reason, it

will create new Bulk Aggregate with root component and forward components

to it.

Interface The RecurringBulkAggregate interface is defined as follows.

```
public interface RecurringBulkAggregate {
CompletionStage<CurrentOpenBulkResponse> createBulk(CreateBulkCommand
command);
```

CompletionStage<RecurringBulkComponentIdResponse>

nent(AddComponentCommand command);

addCompo-

} createBulk is used to create a new recurring bulk. Returns CurrentOpenBulkResponse with BulkId of the current single bulk and root component id to

which components will be sent via addComponent method

addComponent is used to add new component to the current open bulk. Returns

RecurringBulkComponentIdResponse with SUCCESS result and BulkId of the

single bulk aggregate to which component was added, and the BulkComponentId

of the added component.

Limited bulk specification support Since recurring bulk aggregate is responsible

for creating a new single bulk, it is also responsible for generating the root component for created single bulk, and the client can send only components which

will be added as child components of that root component. Client will need to

create an implementation of ComponentGenerator which will be responsible for

generation of single bulk root components created by recurring bulk aggregate.

public interface ComponentGenerator { String generateComponent(String componentName); } generateComponent is used to create a root component for the

single bulk created by the recurring bulk. It should return generated component

based on the passed componentName. componentName which is being passed

by the recurring bulk aggregate is the recurring bulk id.

ERROR Codes When client sends invalid commands Response with FAILURE

result will be returned, and it will also contain the Error.

List of error codes:

Error

Message

Description

For Command

Aggregate type

AC01

Parent id present for root component

Root component shouldn’t have parent

AddComponentCommand

Single

AC02

Parent doesn’t exist

35

Returned if the parent is not present in the bulk aggregate (doesn’t apply to root component)

AddComponentCommand

Single

AC03

Path not valid or not present in the BulkSpecification

Returned if component path is not defined in the BulkSpecification

AddComponentCommand

Single

AC04

Content not present

Returned if content is null or empty

AddComponentCommand

Single

UC01

Component doesn’t exist

Component with BulkComponentId not present in the Bulk Aggregate

UpdateComponentCommand

Single

UC02

Content not present

Returned if content is null or empty

UpdateComponentCommand

Single

RC01

Component doesn’t exist

Component with BulkComponentId not present in the Bulk Aggregate

RemoveComponentCommand

Single

RC02

Root component can’t be deleted

Returned if component can’t be removed because it is a root component

RemoveComponentCommand

Single

RC03

Component has child components

Returned when component can’t be removed because it has child components

RemoveComponentCommand

Single

RB01

”Recurring bulk id not valid”

Returned if recurring bulk id is not provided

CreateBulkCommand

Recurring

RB02

Recurring bulk specification not valid

Returned if BulkSpecification is not valid.

CreateBulkCommand

Recurring

RB03

Unknown bulk marked as closed

The command is not supported in the current state

On any other commands

Recurring

NSC03

Commmand is not supported

The command is not supported in the current state

On any other commands

Implementations As with other IPF libraries, default implementations for the most common use-cases are already provided. Links to the documentation for each implementation are listed below.

Akka ----------
---------- Bulk Element

Adder This provides a receive connector that consumes from a kafka topic a

bulk element information containing the content, path, bulkId and parentId,

When the bulk element is received, the receiver is passing the information to

an adapter interface in order to create the new bulk element.

Config

Type

Comment

Default

bulk-element-adder.transport

String

Property which defines which transport will be used.

kafka

bulk-element-adder.kafka.consumer.topics.new-bulk-request

String

The topic from which messages will be consumed.

BULK_ELEMENT_ADDER_REQUEST

---------- Bulk Initiation

This provides a receive connector that consumes from a kafka topic a bulk

initiation request information containing the bulkId and config. When the bulk

information is received, the receiver is passing the information to an adapter

interface in order to initialise a new bulk.

Config

Type

Comment

Default

bulk-initiation.transport

String

Property which defines which transport will be used.

kafka

bulk-initiation.kafka.consumer.topics.new-bulk-request

String

The topic from which messages will be consumed.

BULK_INITIATION_REQUEST ----------

---------- Bulker Finalisation Initiate finalisation of the bulk (identified by its ID). During finalisation all the necessary supporting structures within the Bulk are finished and validated. Any amendments which need to be made to a parent level element are made at this time. For example, in a three-level hierarchy, both the top level message component (”grandparent”) and the mid-level message component (”parent”) can be modified, but the bottom level message component (”child”) cannot be modified because it has no children of its own.

Finalisation does not edit the content of individual child elements, only the headers (ie counts and cumulative totals) and parents elements that ensure the children are accounted for and accessible.

Finalisation itself does not create an output File, but could call the ”Produce Bulk” method.

Config

Type

Comment

Default

bulk-finalisation.transport

String

Property which defines which transport will be used.

kafka

new-bulk.kafka.consumer.topics.bulk-finalisation

String

The topic from which messages will be consumed.

BULK_FINALISATION_REQUEST ----------

----------Automatic Finalisation of a Bulk Automatic finalisation can be time based

in Scheduled or periodic intervals, or it could be based on the characteristics

of the Bulk(Auto Close Triggers) such as number of elements or estimated

total size of the output file. The method of finalisation is defined at the point

the Bulk is first created. Currently, we support three types for automatic

finalisation.

A bulk may be configured to automatically enter the finalisation state when the

following criteria is met: periodic (i.e. 20 seconds after bulk is created) scheduled time (e.g. at midnight) auto close triggers (e.g. fullnessAutoCloseTrigger

(finalise the bulk after reaching the maximum number of components))

Example: scheduled-auto-close = { auto-close-by-age = 20s schedule-at = ”*/10

* * ? * *” }

auto-close-triggers = [”fullnessAutoCloseTrigger”]

Config

Type

Default

Comment

scheduled-auto-close.auto-close-by-age

Duration

0

Duration value that defines when the automatic finalization of the bulk will be

scheduled with the help of the Scheduler. The values can be anything supported

by the java.time. Duration class, if we want to exclude this option from the

function, we need to specify 0s for the value.

scheduled-auto-close.schedule-at

String

” ” Empty string

A CRON expression is entered for the value, which is parsed and based on

which the automatic closing of the bulk is scheduled. If we want to turn off this

function, we need to specify ”” for the value

auto-close-triggers

List<String>

[] Empty array

auto-close-triggers specify list of triggers that send the bulk to the finalized state

after the certain criteria have been met. The values can be string name for concrete implementation of AutoCloseTrigger interface. If we want to exclude this option from the function, we need to specify [] for the value.

finalise-on-auto-close

boolean

true

boolean value that describes whether the bulk should be finalized after automatically closed by any trigger(AutoClose) default value true.

Important: scheduled-auto-close.auto-close-by-age, scheduled-auto-close.scheduleat and auto-close-triggers are configured at the bulk level and are part of the

overall configuration for that bulk Example of configuration:

output ipf.bulker { configurations = [{ name = ”pain.001.001.09” file-nameprefix = ”bulk-” component-hierarchy { component-parser-name = ”xml”

marker = ”Document” children = [{ marker = ”CstmrCdtTrfInitn.PmtInf”

children = [{ before-elements = [”SplmtryData”] marker = ”CdtTrfTxInf” }] }

] } auto-close-triggers = [”fullnessAutoCloseTrigger”] scheduled-auto-close = {

auto-close-by-age = 30s schedule-at = ”*/10 * * ? * *” } finalise-on-auto-close

= true }] } Choice of which time to use If auto-close-by-age is specified

and schedule-at is turned off(empty string ””) as an option, the time from

auto-close-by-age will be used

If schedule-at is specified and auto-close-by-age is turned off(0s) as an option,

the time from schedule-at will be used

If both values are specified for scheduling, the value closest to the bulk creation

time will be used for auto close scheduler

If we want to turn off the automatic closing of the bulk, it is necessary to

configure auto-close-by-age = 0s and schedule-at = ””

@RequiredArgsConstructor @Value public class ScheduleAutoClose {

Duration autoCloseByAge; String scheduleAtCron;

@SneakyThrows public Instant scheduleAt(Instant createdAt) { if (autoCloseByAge == null && (scheduleAtCron == null || scheduleAtCron.isEmpty()))

{ throw new IllegalStateException(”At least one of autoCloseByAge or scheduleAtCron must be set”); }

if (autoCloseByAge == null) { CronExpression cronExpression = new

CronExpression(scheduleAtCron); return Objects.requireNonNull(cronExpression.getNextValidTimeAfter(Date.from(createdAt.atZone(ZoneId.systemDefault()).toInstant())))

.toInstant(); }

if (scheduleAtCron == null || scheduleAtCron.isEmpty()) { return createdAt.plus(autoCloseByAge); }

CronExpression cronExpression = new CronExpression(scheduleAtCron); Instant cronInstant = Objects.requireNonNull(cronExpression.getNextValidTimeAfter(

Date.from(convertToTimeZonedInstant(createdAt))) .toInstant());

Instant

byAge = createdAt.plus(autoCloseByAge); return cronInstant.isBefore(byAge)

? cronInstant : byAge; }

private Instant convertToTimeZonedInstant(Instant createdAt) { TimeZone.setDefault(TimeZone.getTimeZone(”UTC”)); return Date.from(createdAt).toInstant();

} } // end::class] The ClientComponent interface is defined as follows.

public interface AutoCloseTrigger { boolean isTriggered(Bulk bulk, BulkComponent component); AutoCloseTriggerType getName(); } isTriggered Defines

the condition after which the bulk is sent to the finalized state

getName Name of the trigger

Component Parser A ComponentParser is a

pluggable component whose purpose is to detect byte position where child

components will be injected into that component. Those positions are called

Insertion Points.

A component hierarchy is used to define how each component relates to each

other, and where within one component’s content its child component’s content

should be placed.

Concrete implementation are expected to know the format of the content of

a component, i.e. xml, json , csv, etc.. This is required, since the joining of

the components is dependent on the way the data is structured. That is why

a separate ComponentParser implementation is going to be needed for each

format.

Interface The ComponentParser interface is defined as follows.

public interface ComponentParser { String getName();

List<InsertionPoint> parse(String content, Node node); } getName returns a

name of the ComponentParser. parse takes a bulk component Content, a bulk

specification Node and returns a InsertionPoint list. Implementations Links to

the documentation for each implementation are listed below.

XML

New Bulk Notification This provides a receive connector that consumes new

bulk notification together with bulk component ids from which a new bulk is

to be composed. When the NewBulkNotification notification is received, the process of composing a new bulk is triggered.

Config

Type

Comment

Default

42

new-bulk.transport

String

Property which defines which transport will be used.

kafka

new-bulk.kafka.consumer.topics.new-bulk-request

String

The topic from which messages will be consumed.

NEW_BULK_REQUEST ----------

---------- Default

The DefaultJoiner provided by IPF is implemented using the StAX API, which

is included within the standard Java language.

The joiner has a dependency on the component store where it is expected to

be already populated with components that, when combined with a component

hierarchy to define the components relationships, can be merged together to

form a valid XML document that can be ingested by another system.

Usage Example This usage example follows the same scenario used to demonstrate the XML Splitter but in reverse. We will populate a component store,

create a component hierarchy to determine how the components should be joined

together and finally provide an output stream so that the joined content can be

streamed out to wherever it needs to go.

This example joins together only a handful of components for demonstrative

purposes, but this can scale to many more components. Below are each of the

component's content that is persisted within the component store.

example-library-component.xml <library> <name>Library of Alexandria</name> </library> example-book-component-1.xml <book> <author>Martin, Robert</author> <title>Clean Code</title> </book>

example-book-component-2.xml <book> <author>Bloch, Joshua</author>

<title>Effective Java</title> </book> example-chapter-component-1.xml

<chapter> <name>Clean Code</name> <startPage>1</startPage> </chapter> example-chapter-component-2.xml <chapter> <name>Meaningful

Names</name>

<startPage>17</startPage>

</chapter>

examplechapter-component-3.xml <chapter> <name>Introduction</name>
<startPage>1</startPage>

</chapter>

example-chapter-component-4.xml

<chapter> <name>Creating and Destroying Objects</name> <startPage>5</startPage> </chapter>
Let’s write an example program to first

load the components into the component store and then process them using

the XML Joiner.

ComponentStore<List<InsertionPoint» componentStore = new InMemoryComponentStore<>();

```
// Create component hierarchy var rootNode = Node.root(”library”, ”xml”);

var bookNode = rootNode.createChild(”book”, Collections.emptyList()); var

chapterNode = bookNode.createChild(”chapter”, Collections.emptyList());


// Populate the component store BulkId bulkId = BulkId.random(); var root =

Component.<List<InsertionPoint»builder() .bulkId(bulkId).id(ComponentId.of(bulkId.getValue()))

.index(0L).marker(”library”)

.content(readResourceFile(”example-rootcomponent.xml”)) .custom(List.of(new
InsertionPoint(bookNode, 49))).build();

var book1 = Component.<List<InsertionPoint»builder()
.bulkId(bulkId).id(ComponentId.random()).parentId(

.index(1L).marker(”library.book”)

.content(readResourceFile(”examplebook-component-1.xml”)) .custom(List.of(new
InsertionPoint(chapterNode,

73))).build();

var book2 = Component.<List<InsertionPoint»builder()

.bulkId(bulkId).id(ComponentId.random()).parentId(root.getId()) .index(2L).marker(”library.book”)

.content(readResourceFile(”example-book-component-2.xml”)) .custom(List.of(new

InsertionPoint(chapterNode, 76))).build(); var chapter1 = Component.<List<InsertionPoint»builder()

.bulkId(bulkId).id(ComponentId.random()).parentId(book1.getId())

.index(3L).marker(”library.book.chapter”) .content(readResourceFile(”examplechapter-component-
1.xml”)) .custom(Collections.emptyList()).build();

var

chapter2 = Component.<List<InsertionPoint»builder()
.bulkId(bulkId).id(ComponentId.random()).parentId(b

.index(4L).marker(”library.book.chapter”) .content(readResourceFile(”examplechapter-component-
2.xml”)) .custom(Collections.emptyList()).build();

var

chapter3 = Component.<List<InsertionPoint»builder()
.bulkId(bulkId).id(ComponentId.random()).parentId(b

.index(5L).marker(”library.book.chapter”) .content(readResourceFile(”examplechapter-component-
3.xml”)) .custom(Collections.emptyList()).build();

var
```

chapter4 = Component.<List<InsertionPoint»builder()
.bulkId(bulkId).id(ComponentId.random()).parentId(b

.index(6L).marker(”library.book.chapter”) .content(readResourceFile(”examplechapter-component-4.xml”))

.custom(Collections.emptyList()).build();

Mono.zip(Mono.fromCompletionStage(componentStore.save(root)),
Mono.fromCompletionStage(componentSt

Mono.fromCompletionStage(componentStore.save(book2)),
Mono.fromCompletionStage(componentStore.save(

Mono.fromCompletionStage(componentStore.save(chapter2)),
Mono.fromCompletionStage(componentStore.sa

Mono.fromCompletionStage(componentStore.save(chapter4))).block();

Joiner joiner = new XmlJoiner(componentStore); OutputStream stream = new

ByteArrayOutputStream();

var rootId = BulkComponentId.of(root.getId().getValue());
Mono.fromCompletionStage(joiner.join(rootId,

rootNode, stream)).block(Duration.ofSeconds(5));

String output = stream.toString(); System.out.println(output); Running this

code should print out the following to the console. The whitespace formatting

may look a bit different due to the way the components are appended to the

stream, but the content should effectively be the same.

output <library> <name>Library of Alexandria</name> <book> <author>Martin, Robert</author> <title>Clean Code</title> <chapter>

<name>Clean Code</name> <startPage>1</startPage> </chapter>

<chapter> <name>Meaningful Names</name> <startPage>17</startPage>

</chapter> </book> <book> <author>Bloch, Joshua</author> <ti-

tle>Effective

Java</title>

<chapter>

<name>Introduction</name>

<startPage>1</startPage> </chapter> <chapter> <name>Creating

and Destroying Objects</name> <startPage>5</startPage> </chapter>

</book> </library>

---------- IPF Debulker The IPF

Debulker provides the ability to process a bulk files which contains multiple

messages and transactions, potentially between multiple debtors and creditors.

The Debulker takes the responsibility for receiving the file bulk and breaking

it into components. It allows large (bulked) files to be processed by IPF and

does not perform activities which modify the incoming data (e.g. enrichment,

validation). Thus, it is a technical enablement feature to support large,

multi-transaction files.

Once the file is split, those components can then be processed by individual

component IPF flows, meaning we can split a PAIN001 or PACS008 into individual credit transfer instruction components and process each separately. The

Debulker itself does not provide those processing flows, but we do provide guides

and 'how to' sections as examples or starters.

Explore the concepts, features and guides:

Concepts

Features

Getting started

----------Concepts Introduction

At a high-level the Debulker will transformation an

incoming file into the component store. It does this by either polling for the file

or receiving a notification that a new file is available for Debulking processing.

The file can them be streamed and pushed through the appropriate splitter

which will publish a stream of events containing smaller chunks (components).

Key Concepts The following are key concepts which are explored in more detailed within the linked features section. They are explained here to show how

these concepts and features relate to each other.

Debulking Configuration Every type of bulk file to be processed requires a specific configuration, to tell the Debulker what sort of file format to expect (e.g.

XML, Json) and crucially a component hierarchy which provides the tree structure of that specific type of file. This tells the debulker how to break the file

apart, to split it, into its component parts.

File Notification There are two ways the Debulker can learn that there is a file ready for processing. The first is via a notification, an API is provided which is essentially a receive connector. The Debulker comes with a Kafka implementation of this receive connector. Thus an implementation could have another process or script run to send a Kafka event to a specific topic, thus communicating a new file is ready for processing.

File Polling The second way for a file to be fed into Debulker processing is to configure a File Poller, which will poll at a defined frequency for new files. The File Poller can also be used to sweep up missed files, whereby you could configure it to look for files not yet processed (useful in the case that file notifications could not reliably be sent).

Input Stream The Debulker provides a pluggable component whose purpose is to take a FileDefinition and return an InputStream. This decouples the debulker from the underlying details of the file storage, and allows a range of storage options (e.g. file system most commonly or S3 bucket).

File Processing Uniqueness We typically want the files to be processed once and only once, thus the debulker has the option to configure a duplicate check. This is based on the entire contents of the file and will stop processing a file which it has seen before.

Splitter A Splitter is a pluggable component, where most of the Debulker work is done. The Splitter takes a stream of data (current from a large file); and publishes a stream of events containing smaller chunks (components).

Component Store The File Component Store is a plugable component and represents the 'place' where payment components are stored. Typically this will be a Mongo backed store, but could equally be implemented or swapped for another implementation.

Client Processing Notification This is enabled using a pluggable component which sends notifications to a client indicating that components generated by the debulker are ready for processing.

Housekeeping Housekeeping functionality exists to remove components which have been processed by the client flows.

---------- Features Bulk Input Stream Provider

Local Filesystem

Client Processing

Client Processing Kafka

Debulker File Archiver

Local Filesystem

S3 Archiver

Debulking Configuration

Housekeeping component remover

Housekeeping file deleter

Housekeeping - scheduler

New File Notification

New file polled connector

Splitter

JSON Splitter

XML Splitter

Duplicate check

----------Bulk

Input

Stream Provider A BulkInputStreamProvider is a pluggable component whose

purpose is to take a FileDefinition and return an InputStream.

This decouples the debulker from the underlying details of the file storage and

consequently.

Since the debulker allows multiple input sources, the BulkInputStreamProvider

is expected to return a name that uniquely identifies it.

Interface The BulkInputStreamProvider interface is defined as follows.

public interface BulkInputStreamProvider { String getName();

InputStream stream(FileDescriptor fileDescriptor); } getName is used to

uniquely identify the implementation. stream takes a FileDescription returns

an InputStream. Implementations As with other IPF libraries, default implementations for the most common use-cases are already provided. Links to the

documentation for each implementation are listed below.

Local Filesystem ---------- Local Filesystem An implementation of BulkInputStreamProvider for

reading files from the local filesystem.

Maven Dependency To use the LocalFilesystemBulkInputStreamProvider, the

following dependency must be provided, with a verison matching ipf-debulkercore to ensure compatibility.

<dependency> <groupId>com.iconsolutions.ipf.debulk</groupId> <artifactId>ipfdebulker-bulk-input-stream-local</artifactId>

<version>${ipf-debulkercore.version}</version> </dependency>

---------- Client Processing

Client Processing is a pluggable module which purpose is sending notification

that components generated by debulker are ready to be processed, and handling notifications that components are processed so that debulker can perform

housekeeping operations.

Interface The ComponentProcessingInitiationPort interface is defined as follows,

and it is used to send notification that components are ready to be processed.

public interface ComponentProcessingInitiationPort {

CompletionStage<Void> initiateProcessing(InitiateComponentProcessingCommand

command); } initiateProcessing is used to send InitiateComponentProcessingCommand. The ComponentProcessingCompletionPort interface is defined

as follows, and it will handle notification that processing of components is

completed. Application which uses this module will need to implement it.

public interface ComponentProcessingCompletionPort { CompletionStage<Void> processingComplete(ComponentProcessingCompleteCommand

command); } processingComplete is used to handle InitiateComponentProcessingCommand. Implementations We use connector library to send and

consume messages to/from specific transport. Currently, there is only - kafka

implementation.