Simflofy Processor Developer Guide

Simflofy v. 3.0 +

v1.0 S. Arnold

Jul 16, 2021

Processor Basics

Processors (also known as **Job Tasks)** are POJOs used to define a unit of work that will be performed against all documents in a job queue. The primary method of the processor is appropriately, **process()**. This is the method called by the migration manager to execute the work.

A processor will return the status of each document it processes. The creator of a processor can dictate how a document is treated when it leaves the processor. For example, if the processor deems the document failed, it can return a failed status. This means that the job will show a failure for that document due to the processor.

Simflofy comes packaged with several processors and post-processors out-of-the-box. The work a processor performs varies and includes things like cleaning up file names, removing duplicates, and pulling supplemental data from external systems.

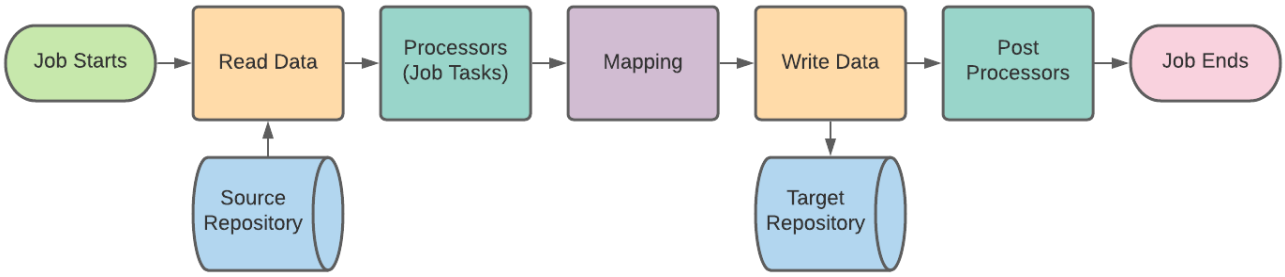
Simflofy Job Flow

In the diagram below you will see the order of operations for a basic Job in Simflofy. This job flow is the same whether you are using the job for migration, indexing, or data analytics.

It is important to note that the Processors (aka Job Tasks) will run against each Repository Document in the queue immediately after they are read from the repository, but before any mappings take place. This allows you to set fields on the Repository Document which can then be included in the mappings.

Post-processors will run after the document has been written to the target repository. This means you will have access to information about the newly created document. Most notably, you will know the newly created ID or UUID of the document. Only documents that have been successfully written to the target repository will be sent to the post processor. This is useful for performing work against the source repository without affecting un-migrated content (or content that failed to be read or processed). For example, if you want to lock the original file or delete the original file in the source repository you could leverage a post-processor for that task.

Job Process Flow



Processor Development

We designed processors to be as simple as possible for extension. They are a powerful tool that can significantly improve the quality of the data you are migrating, indexing, or analyzing.

Developing processors requires a working knowledge of Java and Spring. If you do not wish to delve that deep, try putting your logic in a JavaScript Processor Task.

When developing Processors for Simflofy an important thing to keep in mind is the impact your work will have on the overall performance of the migration job. Each document you process will be impacted by your processor.

Mitigate performance issues by:

1. Leveraging Init and Teardown methods properly
2. Caching where applicable
3. Understanding JVM and memory management

File Structure

The sample project (linked below) shows one example of setting up a custom extension project. The tree view looks something like this:

├── pom.xml

```
├── readme.md
└── src
    ├── main
    │   ├── java
    │   │   └── com
    │   │       └── simflofy
    │   │           └── sample
    │   │               └── SampleTask.java
    │   └── resources
    │       └── i18
    │           └── sample-task_en.properties
    └── test
        └── java
```

The primary work being done is in a single Java class, SampleTask.java. This class will define methods for setting up, running, and tearing down the processor. Optionally, you can create an i18n resource file to define labels for internationalization.

If you have access to Simflofy source repository (request access via support ticket) you will be able to add the maven dependencies to your pom. Otherwise you can grab the necessary jars from your version of Simflofy and add them as external dependencies.

Tasks should be compiled against the version of Simflofy they will run in. Typically if a task requires the API of a specific connector, the task will be packaged with that connector. Once compiled and added to a jar, the jar can be loaded through the WEB-INF/lib directory in tomcat or a shared loader space.

Package Name

Make sure your package name starts with **com.simflofy**. For example, you might set your custom package name as **com.simflofy.custom.companyName**. This will ensure that your components are loaded by Spring.

i18N

Simflofy leverages Spring for internationalization. There are some base methods available through the AbstractTask class to help pull in the proper labels for the locale.

Make sure your Processor constructor sets a message source that matches the location of your properties file for internationalization.

```
public SampleTask() {
   ResourceBundleMessageSource bundleMessageSource = new
ResourceBundleMessageSource();
   bundleMessageSource.setBasename("i18/sample-task");
   bundleMessageSource.setDefaultEncoding("UTF-8");
   this.setMessageSource(bundleMessageSource);
   setMessageBase("sampleTask");
   setName("sampleTask");
}
```

getTaskMessage(String label)

This will look for messages prepended with “task”. ie “task.taskName”

getGenMessage(String label)

This will look for messages prepended with “general”. These labels have multiple references throughout the product.

getMessage(String label)

Get the label based on the current message base. Uses whatever message base is set in the constructor. ie

Use task messages to show labels in config.

```
epf.addCheckEP(TEST_CHECK_BOX, getTaskMessage("testMessage"),
getTaskMessage("testMessageDescription"));
```

Once the constructor has the setMessageBase method applied, the method above will return the value for the property keyed as seen below.

```
#Show additional i18n config
task.testMessage=Check?
task.testMessageDescription=This does not actually do anything. Go nuts.
```

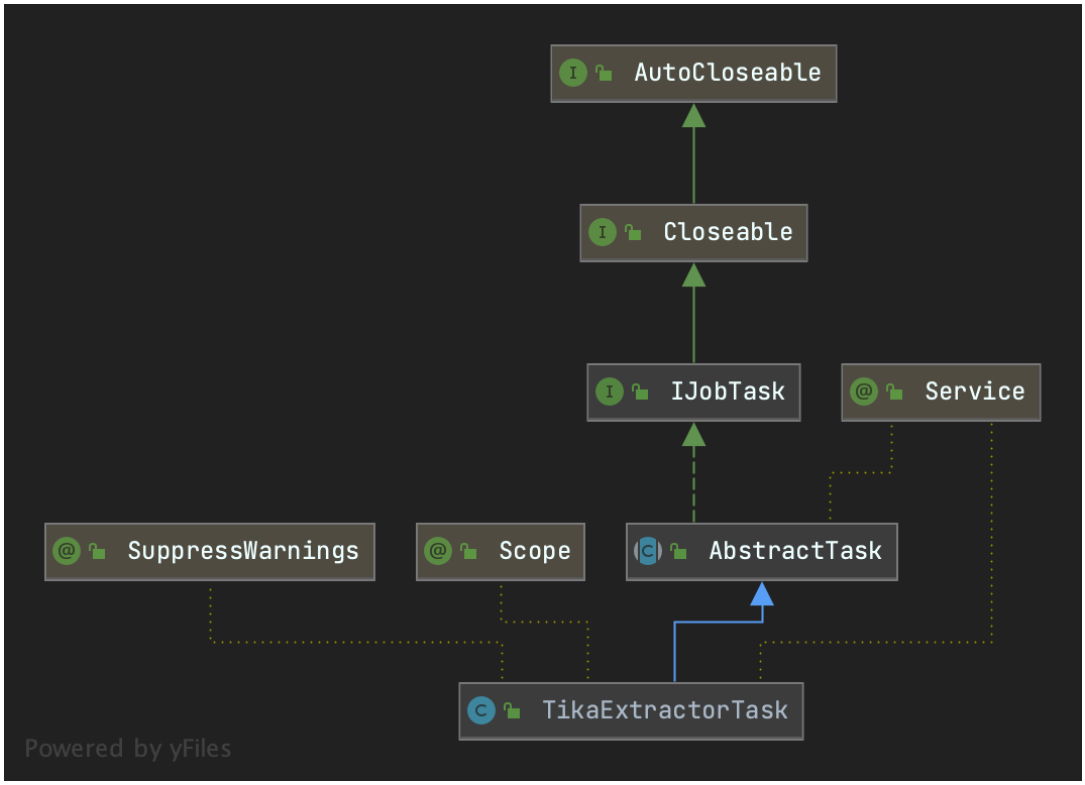
Dependencies

Processor dependencies will vary according to the work being done, but there are some standards for tasks. Notably, all tasks will need require the following module dependencies for compilation:

- simflofy-data (ie simflofy-data-3.0.1-SNAPSHOT.jar)
- simflofy-tasks-core
- simflome-core

These jar files can be found in the war for your version of Simflofy.

Class Setup



Annotations

Processor classes should include the following annotations in order to be properly recognized by Spring:

```
@Service
```

Designates the processor class as a bean which will then be recognized by core configuration.

```
@Scope("prototype")
```

Explicitly sets the scope of the class requiring Spring to create a new object instance every time bean is requested from the Spring container. Allows for better isolation and allows for init and tear-down separation.

Abstract Task

All tasks should extend the **AbstractTask** class. This class will provide some access to the IJobTask interface as well as some i18n functions and fields for UI form support.

Methods

getFormFields()

Create a list of form fields to capture user data for your task.

validateFormFields()

Server side form validation. Verify and notify form data entered by the user.

process()

Main method which will be called on each Repository Document processed by the job.

init()

Initialize your processor. This method will be called before process() to set up the processor for all documents in the queue.

close()

When all documents have been processed the migration manager will call the close method on the processor.

Initialize and Teardown

Proper utilization of the **init** and **close** methods is crucial for process performance. Use the init method to set your configuration, create re-usable clients, and set up caches or other ephemeral data structures that will be used while processing. The close method should be used to make sure nothing is left open and can also be used to perform any ancillary operations that need to be executed after that last document is processed.

Post Processors

<TODO>

Sample Code

Sample code for Simflofy can be found here:

https://github.com/simflofy/sample-code