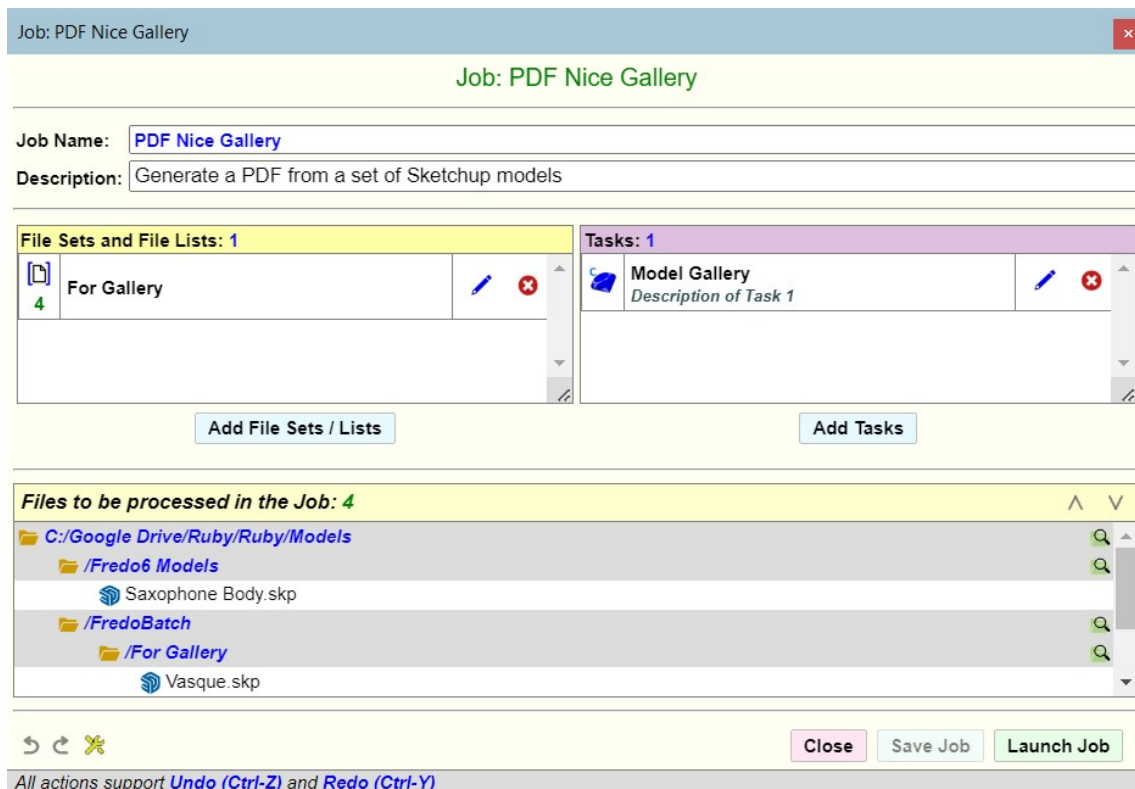


FREDOBATCH

EXECUTING TASKS ON A SET OF FILES

VERSION 1.2 – DECEMBER 2024

BY FREDO6



FredoBatch is an Extension for **SketchUp 2017 and above**.

The **home page of FredoBatch** is on the Sketchucation web site at:

<https://community.sketchucation.com/post/1611210>

Contents

1. Introduction and Main Principles	3
2. File Manager	6
2.1. <i>File Manager dialog</i>	6
2.2. <i>File Lists</i>	7
2.3. <i>File Sets</i>	8
3. Task Manager	10
3.1. <i>Custom Tasks</i>	11
3.2. <i>Param Tasks</i>	12
4. Job Manager	14
4.1. <i>Create and Edit a Job</i>	14
4.2. <i>Launching a Job</i>	16
4.3. <i>Job Execution</i>	16
4.4. <i>Reviewing the Job Execution Summary</i>	17
4.5. <i>Rollback Job</i>	20
5. Designing a Custom Task	21
5.1. <i>Task Method Names</i>	21
5.2. <i>Task Method Arguments</i>	22
5.3. <i>Error Handling</i>	22
5.4. <i>Task Context</i>	23
5.5. <i>Specifying Parameters for the Task</i>	24
6. Examples of Custom Tasks.....	26
6.1. <i>Example 1: Convert DWG/DXF files to Sketchup models</i>	26
6.2. <i>Example 2: Standard Scenes</i>	28
6.3. <i>Example 3: Model Gallery (simple version)</i>	31
6.4. <i>Example 4: Model Gallery (advanced version)</i>	36

FREDOBATCH

Executing Tasks on a Set of Files

1. Introduction and Main Principles

FredoBatch executes a set of tasks on a set of files:

- a **Task** is based on Ruby code, with a *top method* as an entry point. Task can be *built-in* (provided with FredoBatch) or *custom* (configured by the user).
- a **Set of Files** is specified as a combination of lists of files (*File List*) or open set of files with wild card and filtering on names and extensions (*File Set*).
- a **Job** is the combination of sets of files and tasks. This is what is executed by FredoBatch.

A job is created and then launched via the **Job Manager**.

For instance, the job below (*Purge and Convert*)

- applies to a set of Sketchup models (.skp files) contained in a File List (*Cloud Files*) and a File Set (*Shared Models*)...
- ...two tasks: *Purge unused* and *Convert to old SU versions*.

The screenshot shows the 'Job 3 (New Job)' window in FredoBatch. The job title is 'Purge and Convert'. The name field contains 'Purge and Convert' and the description is 'Purge unused and convert to older SU versions'. There are two file sets and two tasks defined.

File Sets and File Lists: 2		Tasks: 2	
4	Cloud Files	Purge unused on Models Perform a purge-unused on SketchUp models	Convert files to old SU versions Convert Sketchup files to old versions
91	Shared Models Bunch of models used in FredoConnect		

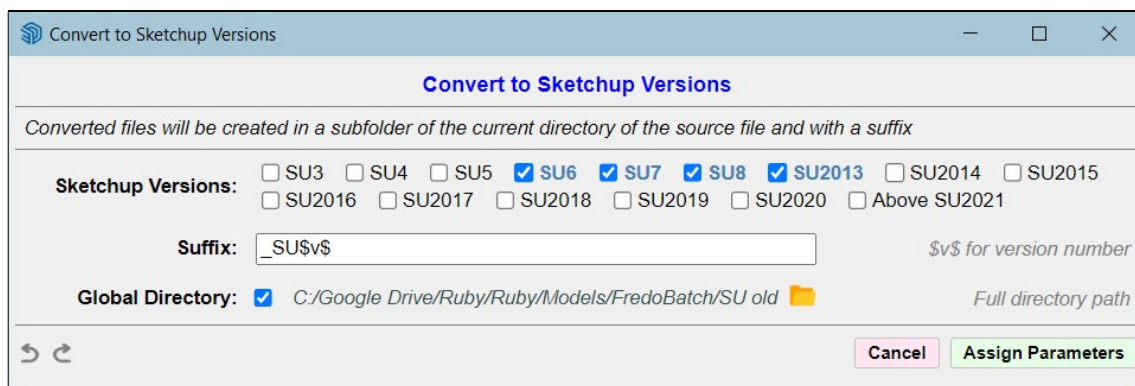
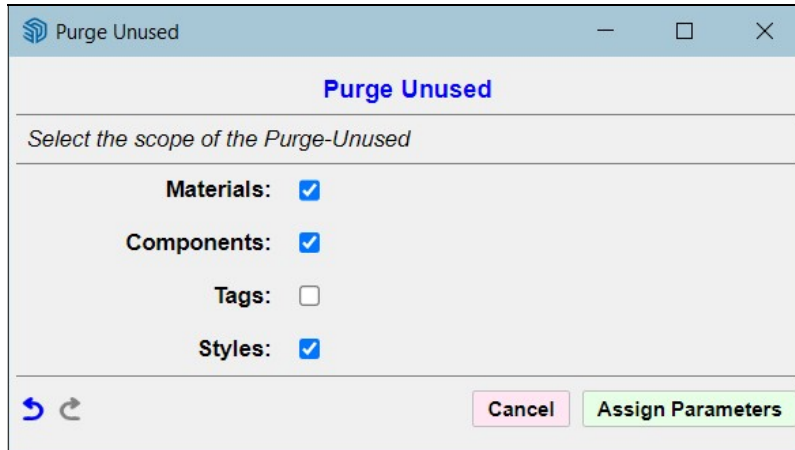
Files to be processed in the Job: 95

- C:/Google Drive/Ruby/Ruby/Models/Delaunay (2)
 - Cloud simple - SU2018.skp
 - Cloud simple.skp
- C:/Google Drive/Ruby/Ruby/Models/Curviloft (2)
 - Curviloft - Bug 24.skp

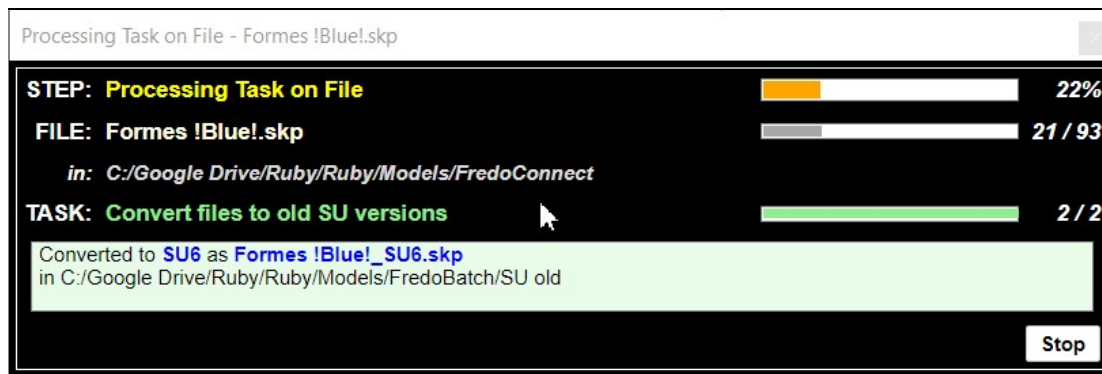
Buttons: Close, Save Job, Launch Job

All actions support Undo (Ctrl-Z) and Redo (Ctrl-Y)

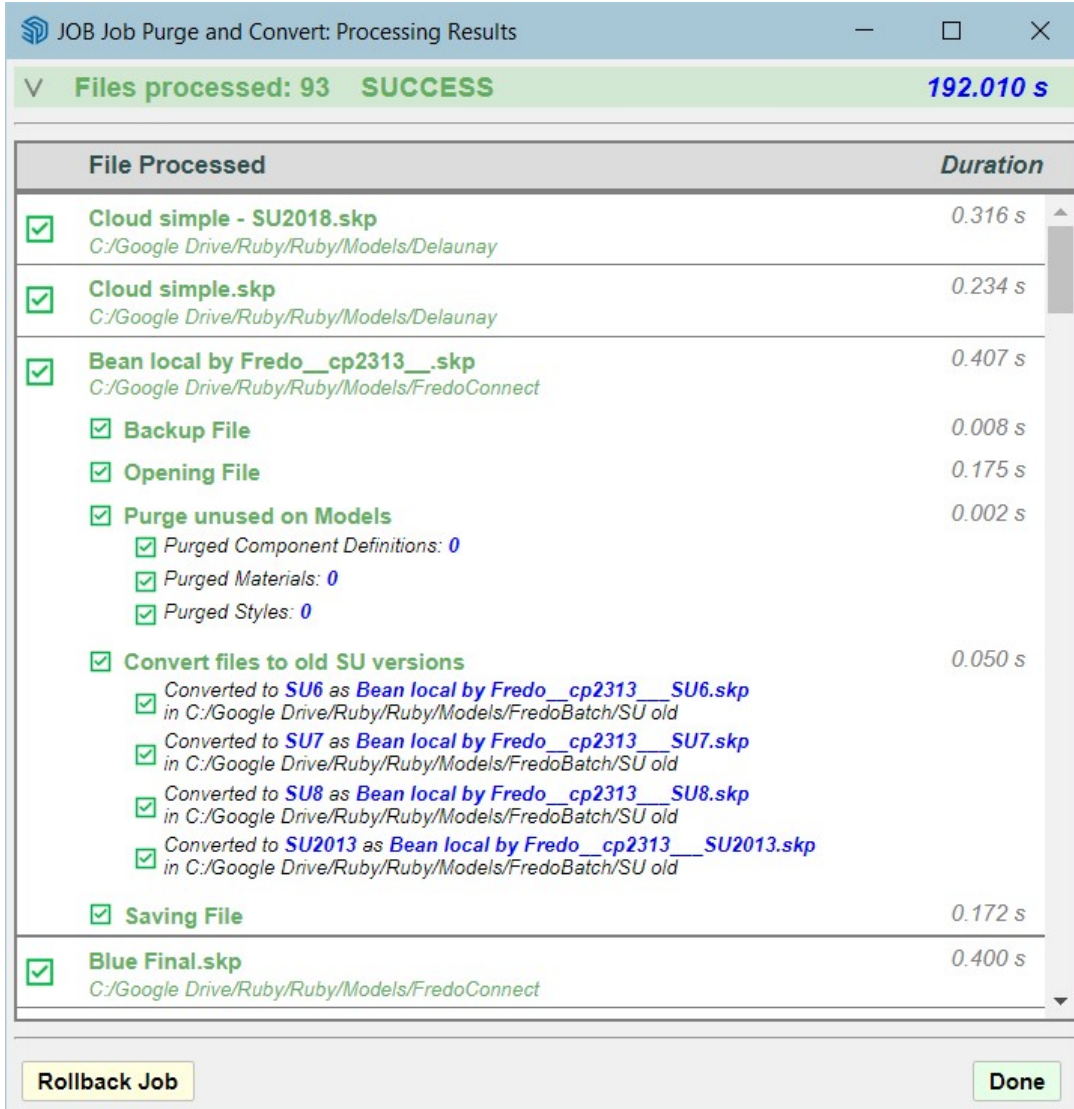
When you launch the job, you are prompted to specify **parameters** for each task:



Then a **Progress dialog** is displayed:

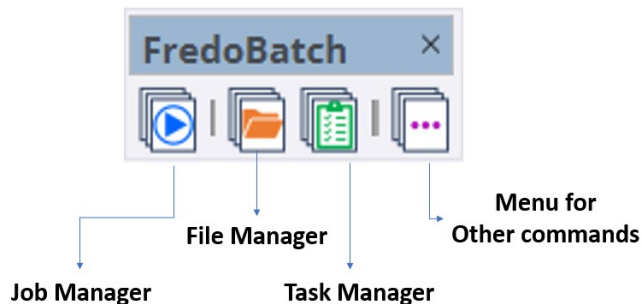


Finally, the process is summarized in the **Results dialog**, with the details of execution:



The Results dialog shows errors in the execution if some happened. It also allows to **rollback the job** if needed.

Tasks, Set of Files, Jobs are managed by FredoBatch via dedicated dialogs. This information is stored on the computer independently of models and versions of Sketchup.





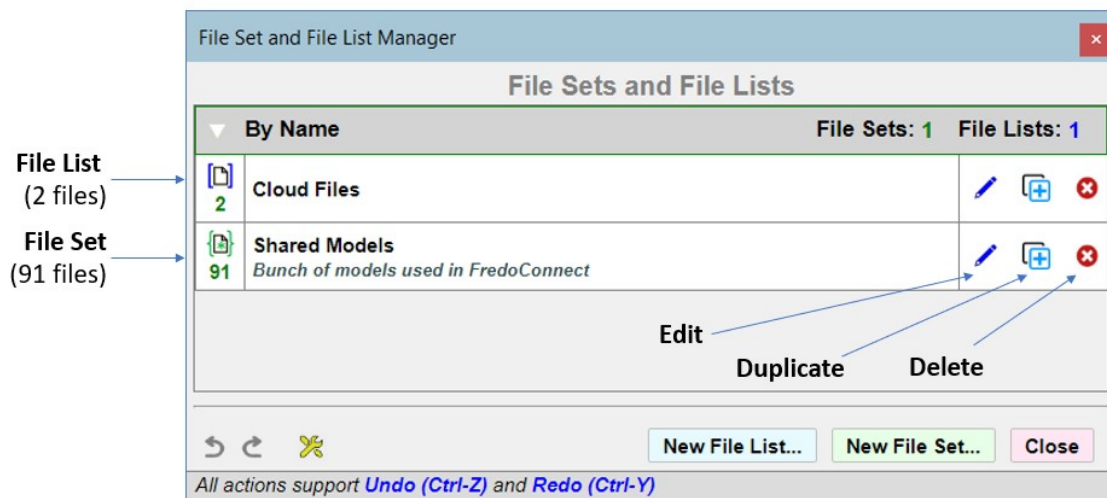
2. File Manager

Files can be specified in 2 ways:

- 1) **File List**: this is a **static** specification of files where you designate each file individually. Files can be in distinct directories.
- 2) **File Set**: this is a **dynamic** specification of files by combining a directory and a set of filters on the file names and extensions. This specification is evaluated at execution of the job, so it may contain files which were not present at the time you defined the File Set.

2.1. File Manager dialog

The File Manager dialog shows the File Lists and File Sets defined in your environment:



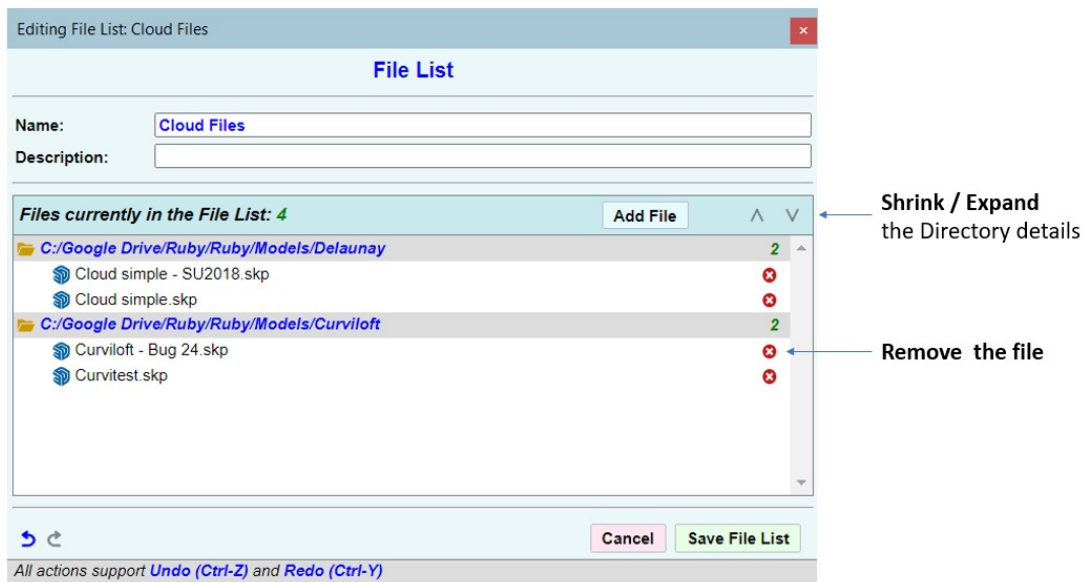
Each File List and File Set is given a name and short description, which will be displayed when you select them for a Job.

To **create** a File List or a File Set, click on the respective buttons at the bottom of the dialog.

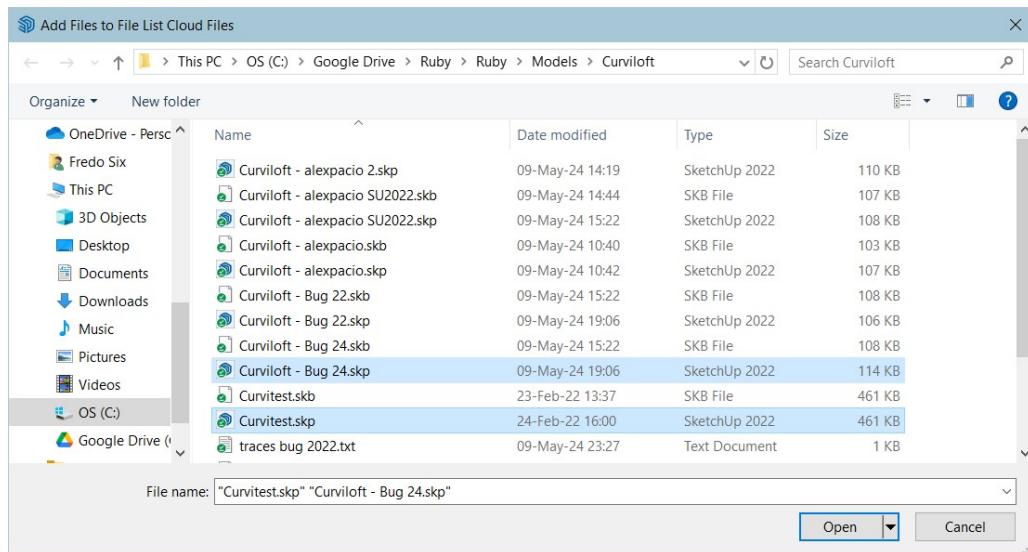
To **edit** a File List or a File Set, double-click click on the Edit icon on the corresponding row.

2.2. File Lists

The dialog to create or modify a **File List** is as follows:

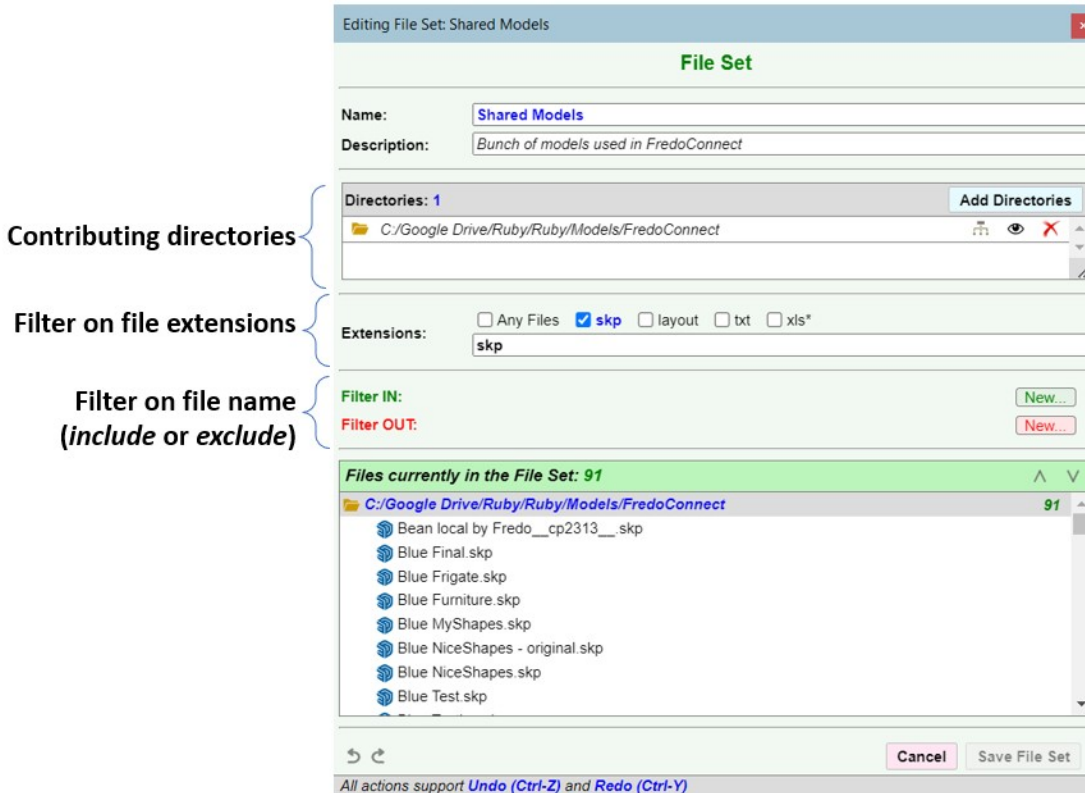


To add files, click on the Button **'Add File'**. You are prompted to select the path to the file(s), by browsing to a directory and selecting one or several files in this directory.



2.3. File Sets

The dialog to create or modify a **File Set** is as follows:



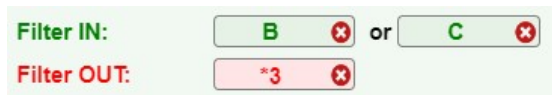
The specifications are based on:

- **One or several contributing directories**, which you select by browsing your computer.
- **A list of allowed file extensions** typed in the field (separated by space, comma or semi-column). For instance, the extension filter below allows both .skp and .dxf files, excluding all other extensions.

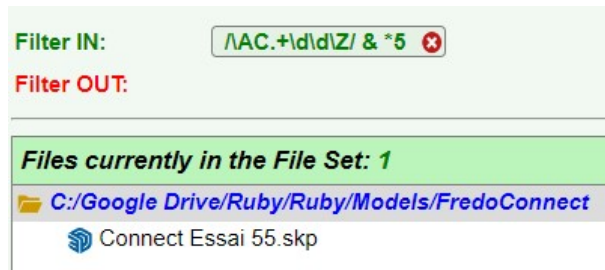


Note that you can use wildcards in the filter (ex: xls* will catch most Excel files).

- **Filters on file names**, either to include (IN) or to exclude (OUT). For instance, the name filters below keep all files whose name begins with *B* or *C* and does not contain 3.



Note that you can combine filter specifications with and-conjunctions (using ‘&’) as well as regular expressions (enclosed in /), if you are familiar with it. For instance, the filters below keep all files whose name begins with *C* and ends with 2 *digits* and also contain the *digit 5*.



The file shown are those that are present on your computer and respect the filtering at the time of creation or modification of the File Set. When you launch a job including the File Set, the files are dynamically recalculated.






3. Task Manager

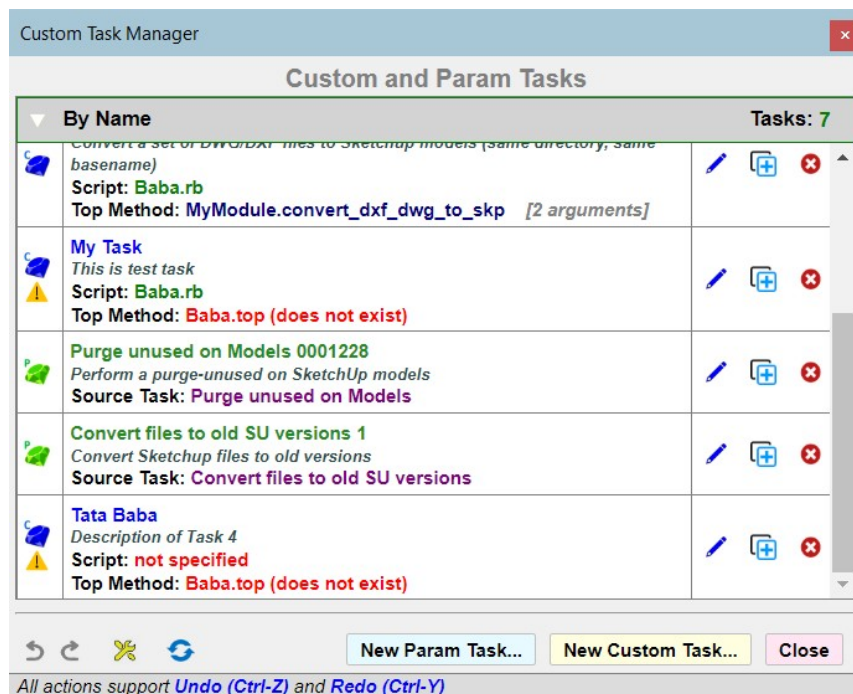
A task corresponds to the action to be executed based on a file as input. For instance, Purge Unused over Sketchup models.

The task may not necessarily act directly on the file itself, but instead use the file to perform other actions. For instance, a task ‘Convert to skp’ executed on DXF files, would take DXF files and convert them to Sketchup models. Likewise, the files could be instructions, say specifications of geometry in a CSV or TXT files, to perform construction of shapes in Sketchup.

In FredoBatch, there are 3 categories of tasks:

-  1) **Built-in Tasks:** they are provided by FredoBatch.
-  2) **Custom Tasks:** they are provided by the user, based on a top method in Ruby.
-  3) **Param Tasks:** these are instances of Built-in tasks or Custom Tasks where parameters are pre-selected, so that there is no prompt for parameters at execution of a job. For instance, from the Built-in Task *Purge Unused*, you can derive a Param task which will only purge materials and styles (but not components).

The **Custom Task Manager dialog** allows to create and manage Custom Tasks and Param Tasks:





3.1. Custom Tasks

Custom Tasks are specified by a **top method in a Ruby script**.

So, creating custom tasks requires that you are **familiar with Ruby**, or are instructed by someone who is familiar with Ruby. **Section 5** below gives more detail on how to script a custom task for simple and complex cases.

The creation and edition of Custom Tasks is done via the **Custom Task Editor dialog**:

The Custom Task is given a **Category¹**, a **Name** and a **Description**.

Script Properties specifies the **top method** entry (here *Baba.top*) and optionally a **Ruby script file** (here *C:/Google Drive/Ruby/Ruby/Models/FredoBatch/Baba.rb*) which implements this top method. Note that the Ruby script file is necessary only if the code for the task is not already present in your environment. For instance, some plugins may expose top methods which are compatible with FredoBatch, in which case you do not need a script file.

¹ Categories are for a future categorization of tasks, not effective in the current version of FredoBatch.

File extensions and **Sketchup Versions** can also be specified. Here, the task acts only on Sketchup models, created in SU2020 or above. This means that any other files will be skipped at execution of the task and Sketchup files with version < SU2020 will also be skipped. If you leave these fields empty, there is no restriction.

File Extensions:	<input type="text" value="skp"/>	<i>Allowed extensions separated by space (ex: skp xml)</i>
SU Versions:	<input type="text" value="SU2020 or above"/>	<i>Required SU Version or above</i>

The section **Task Processing** is more related to the Job execution.

- For **Sketchup files**, you indicate whether FredoBatch opens .skp files or not before processing the task.
- **Backup** should be checked whenever the files are going to be modified by the task. This enables an automatic backup of files before processing and therefore a possible rollback afterwards.

Task Processing	
Sketchup Files:	<input checked="" type="checkbox"/> Open Sketchup .skp file before running the task
Backup:	<input checked="" type="checkbox"/> Backup files for possible restore



3.2. Param Tasks

Param Tasks are derived from existing Built-in Tasks or Custom Tasks, with a static specification of parameters.

To create a Param Task, click on the button *New Param Task...* in the Task Manager dialog. You then need to select a task, Built-in or Custom:

Selection of Tasks (Custom, Params and Built-in)

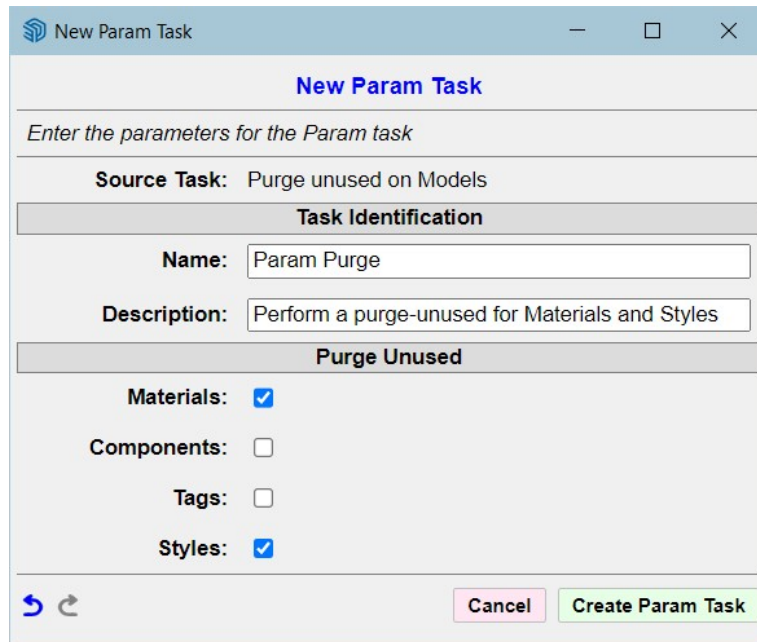
Please pick one or several tasks

Tasks: 5

- Convert files to old SU versions**
Convert Sketchup files to old versions
- Model Gallery**
Description of Task 1
- Purge unused on Models**
Perform a purge-unused on SketchUp models
- Purge unused on Models (all items)**
Purge unused on Models (all items)
- Standard Scenes**
Create Standard Scenes in Sketchup models

Cancel Customize Task

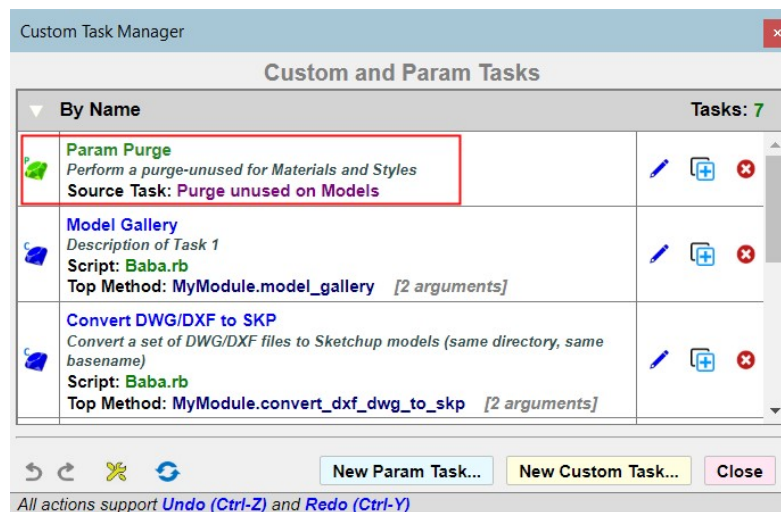
After selecting the base task (here the Built-in Task *Purge Unused on Models*), you get a dialog which includes the parameters of this base task populated by their default values:



As for any task, you must give it a name and description.

Then, **you give values to the parameters** listed in the dialogs (here, 4 flags as checkboxes). These parameters will be registered and associated with the Param task.

The Param task is now visible in the Task Manager and can be used in a Job.



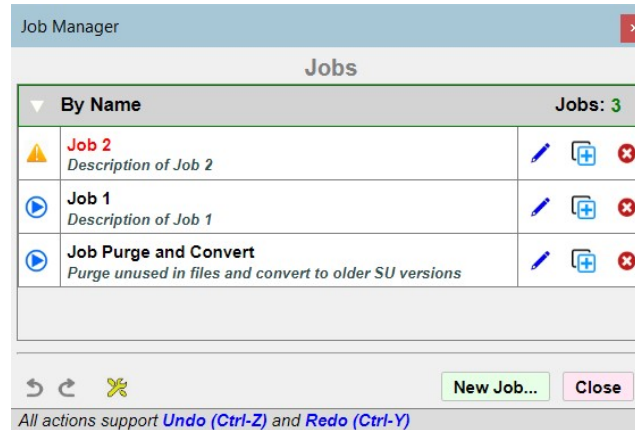
You can later change these values by editing the Param task.



4. Job Manager

A **Job** is specified as the **association of files and tasks**.

Jobs are created, launched and edited via the **Job Manager dialog**.

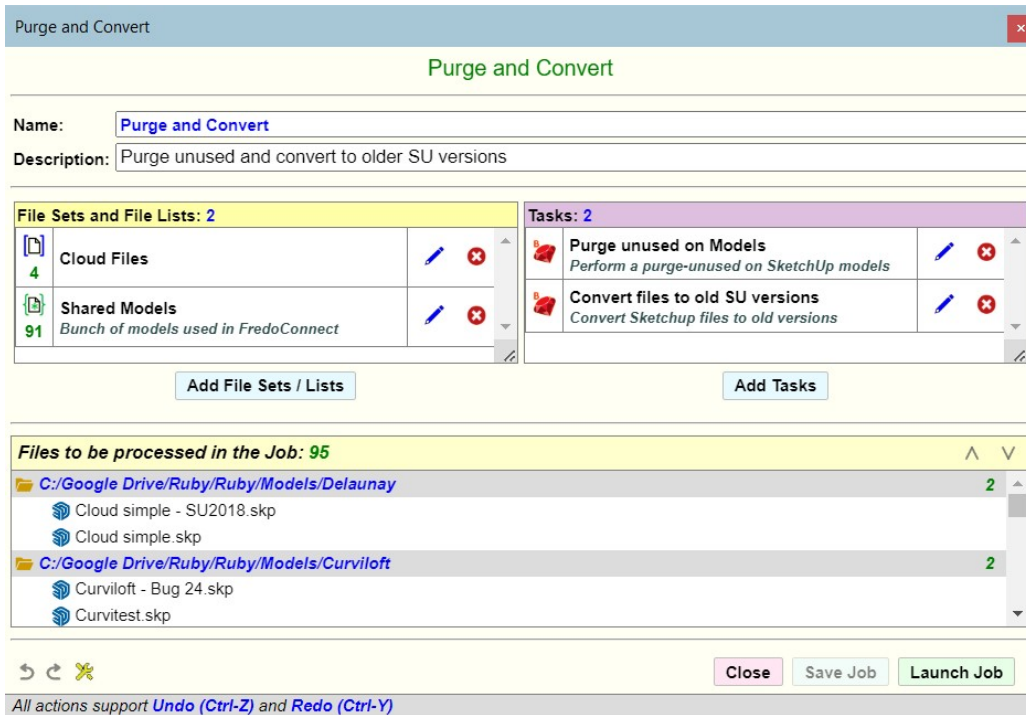


▶ If the **Launch** icon, located in the first column, is displayed, you can directly launch the job by clicking on the icon.

⚠ Otherwise, the **Warning** icon indicates that there is a problem with the job, either related to the tasks (missing or invalid) or files (no files).

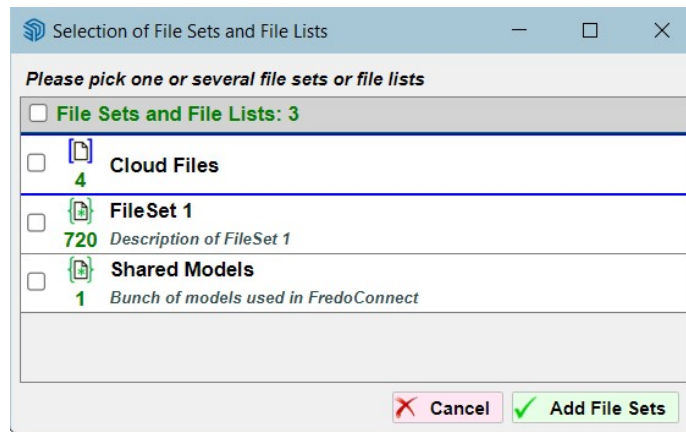
4.1. Create and Edit a Job

To create a Job, click on the button *New Job...* **To modify a Job**, double-click on the job row or click on its Edit icon. In both cases, the **Job Editor dialog** will be displayed:



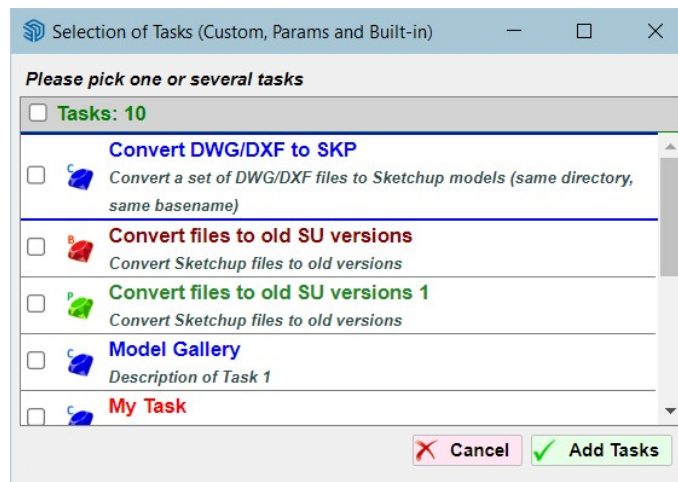
In addition to the name and description of the Job, you need to provide:

- **Files:** in the left list box, add File Lists and File Sets.



You can possibly review and even modify these definitions. The list at the bottom of the dialog will show the files that will participate in the Job execution.

- **Tasks:** in the right list box, add tasks: Built-in, Custom and Param tasks.



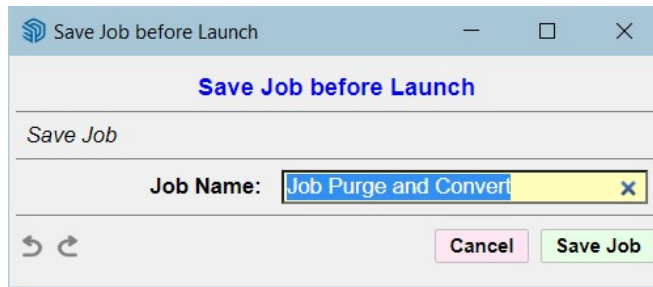
You can possibly review and even modify the tasks.

IMPORTANT: the order of tasks does matter for the job execution. To remove a task and place it at the end of the list, add it again.

4.2. Launching a Job

You can launch a Job from the Job Manager dialog or the Job Editor dialog.

If the job was just modified during the selection process, you are offered to save it either under the same name or a new name:

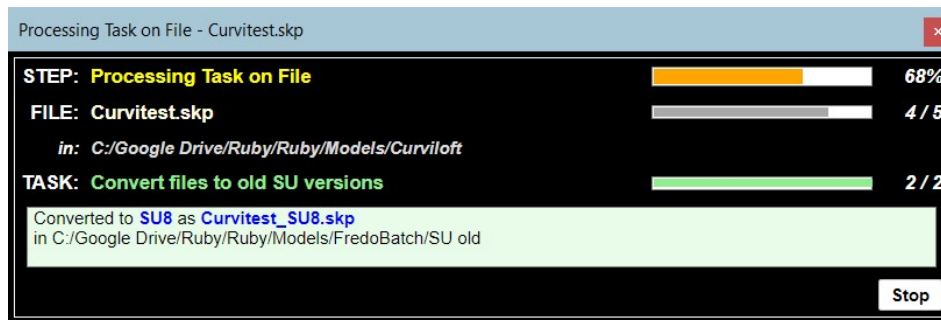


4.3. Job Execution

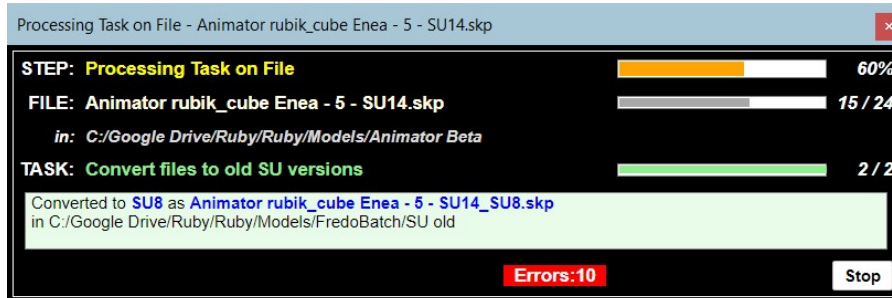
Before the actual execution of the job, you are prompted to **enter the parameters of the tasks** if they have some:



Then, the job is executed on the specified files. A **Progress dialog** is displayed:



In case errors are encountered, they are indicated in the Progress dialog.



You can pause or stop the execution by clicking on the **Stop** button. A confirmation message is displayed:

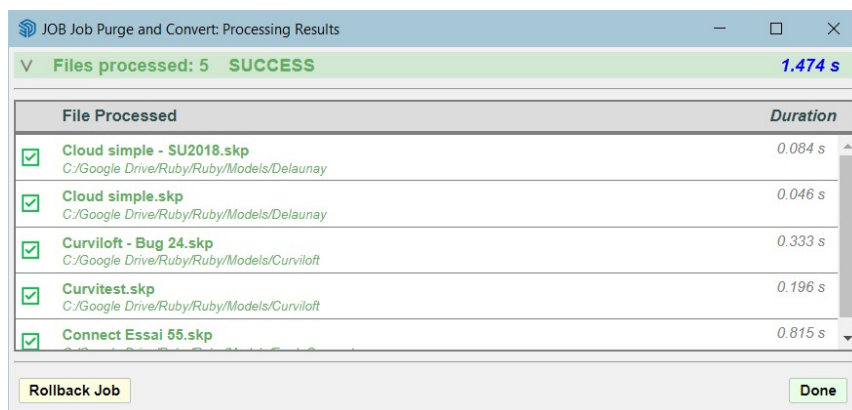


You have 3 options:

- **Resume Job:** just resume the job execution
- **Hard Stop:** FredoBatch will simply stop and exit from the job execution.
- **Rollback Job:** this option will restore the files to their original state. If the tasks support it, they will also restore the initial environment. This is the recommended option.

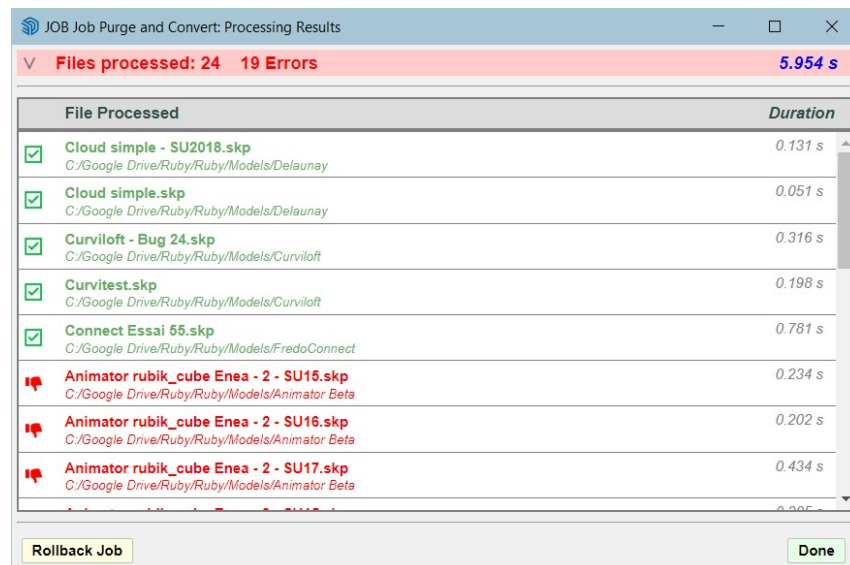
4.4. Reviewing the Job Execution Summary

At the end of the job execution, the **Execution Summary dialog** is displayed:

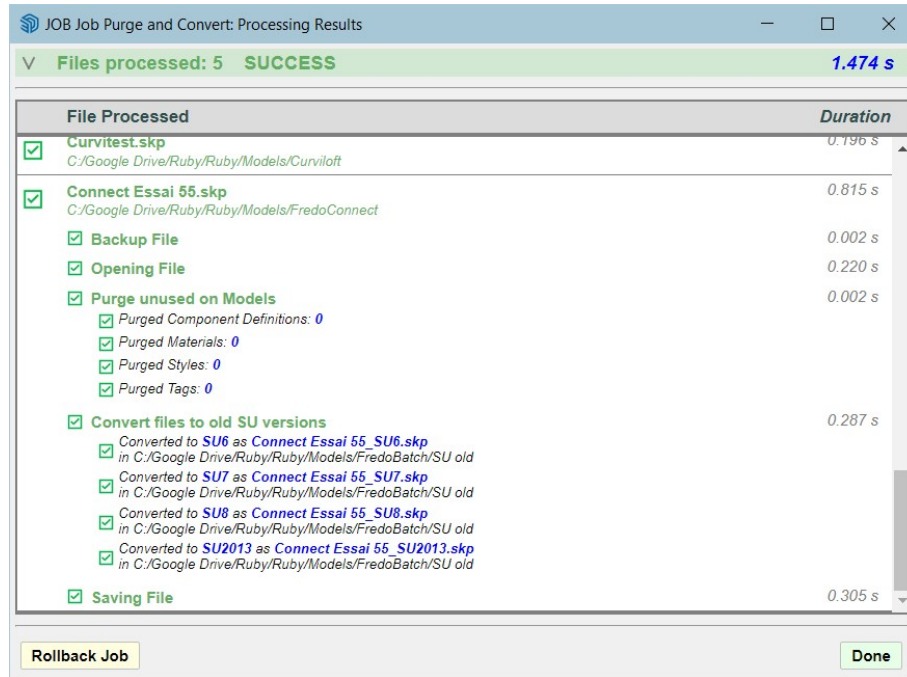


It indicates the number of files processed and the global status: Success or Error.

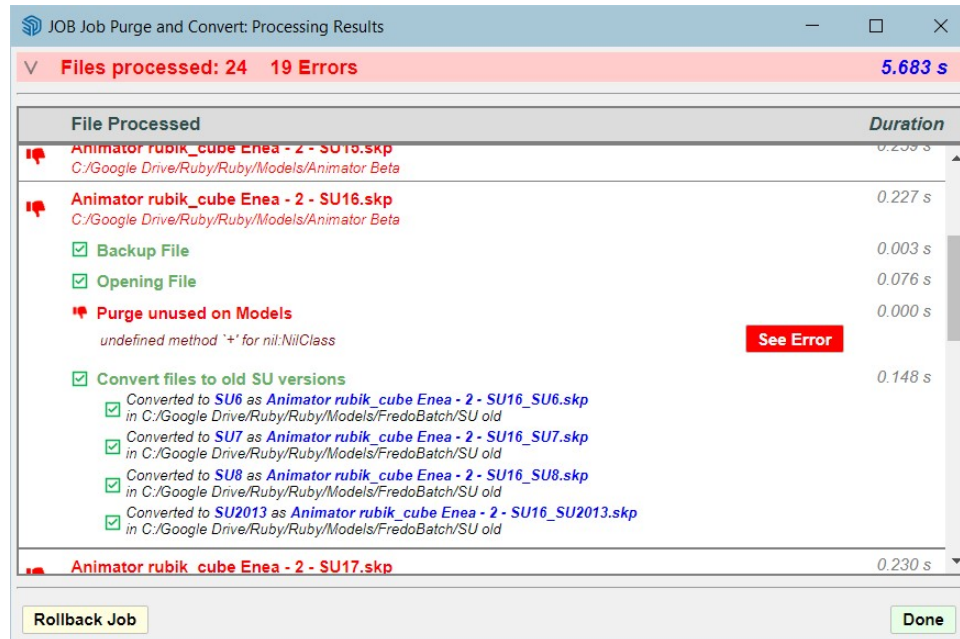
If there are some errors, corresponding file rows are displayed in red:



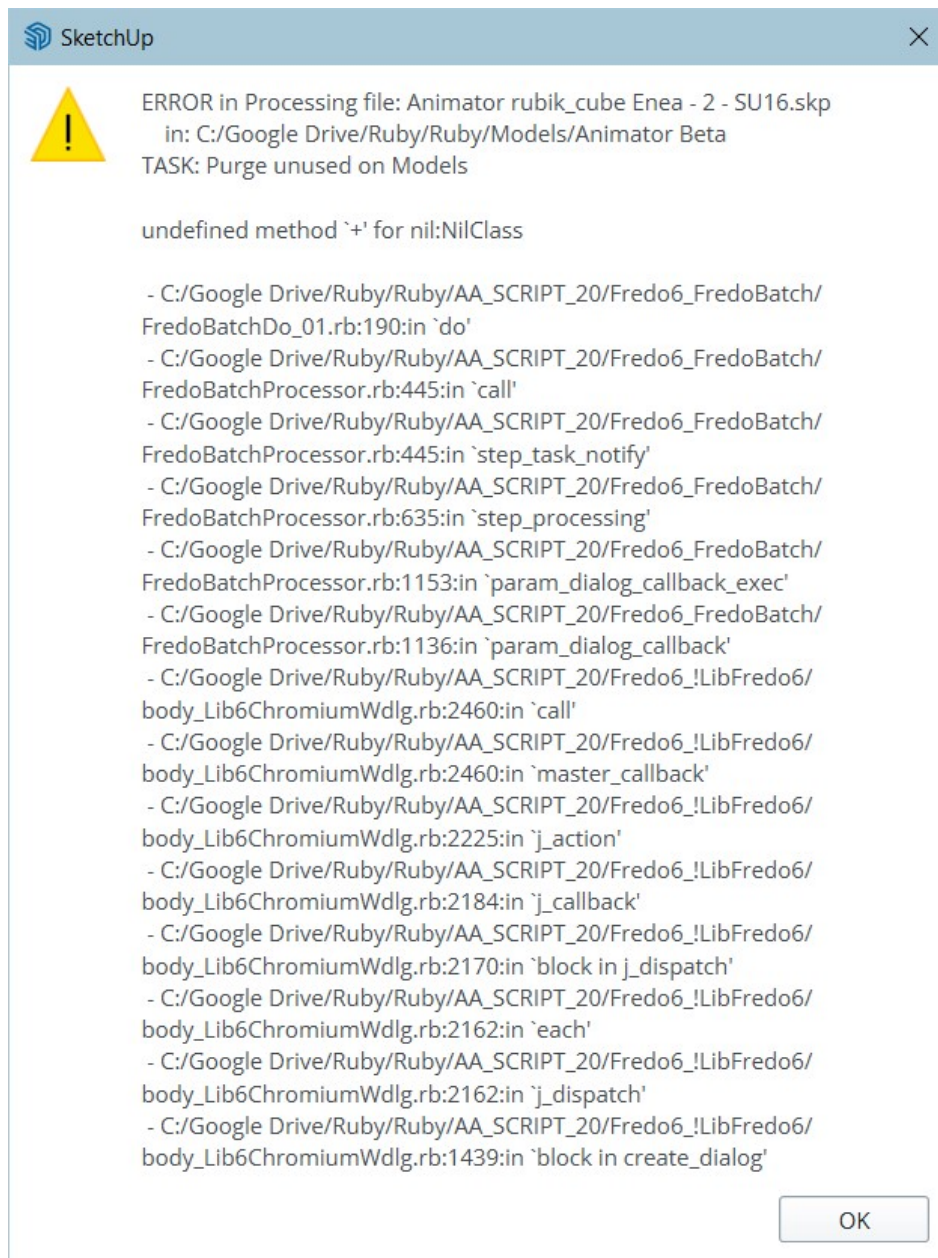
For each file row, you can get a **detail of the execution** by clicking on the row to expand it (double click to expand / shrink all rows).



For files which are in error, file rows are displayed in Red.



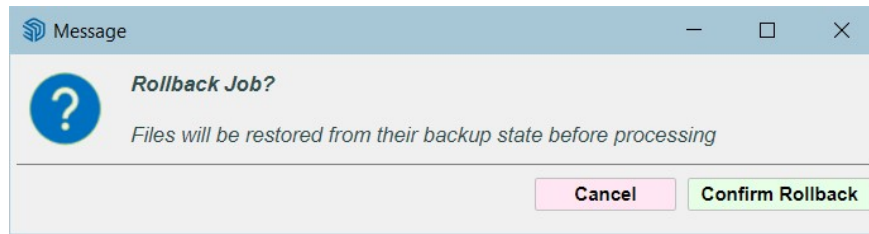
You can get details on the error, by clicking on the button **See Error**. Depending on the error (Ruby or others), you get the technical information about the source of the error.



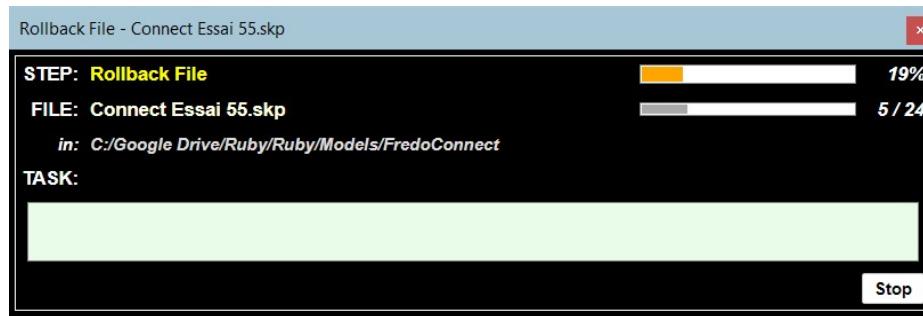
4.5. Rollback Job

After the execution of a job, you have the option to roll back the job, even if it is successful.

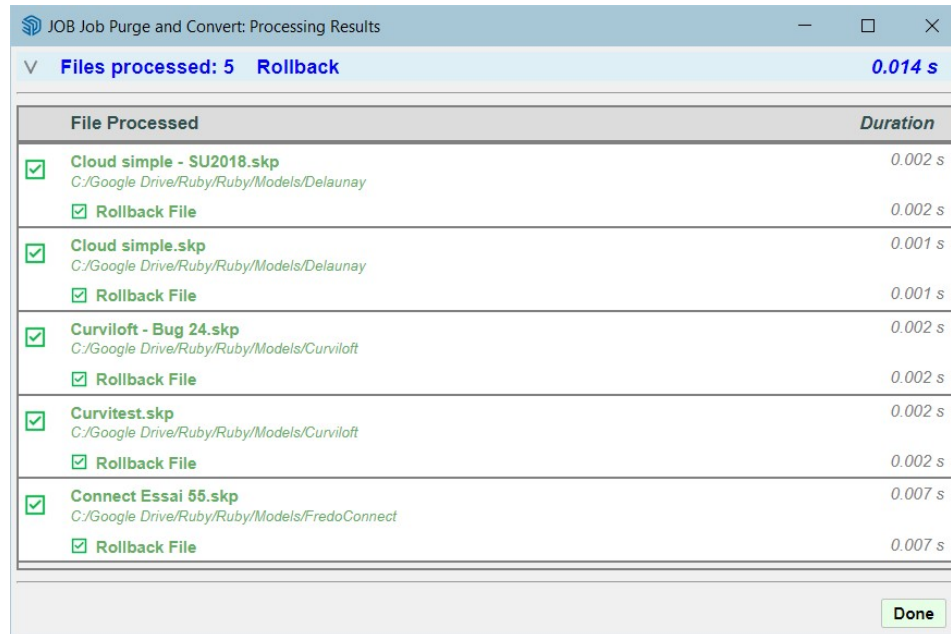
You will first get a confirmation message:



Then the rollback will be executed, with a Progress dialog:



At the end of the rollback, you get a Summary dialog detailing how the Rollback was processed (success or error):

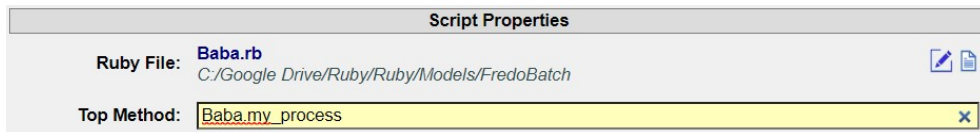


5. Designing a Custom Task

This section is more technical and requires that you are familiar with Ruby and the Sketchup API.

A Custom Task requires the specification of:

- 1) A **Top Method**, written in Ruby.
- 2) Optionally, a **Ruby script file**, which includes this top method, unless this top method is already present in your Sketchup environment (in general, coming from an extension). FredoBatch will ensure that the specified script file is loaded in the environment before processing the task.



5.1. Task Method Names

The Top Method name must be specified with its **complete path from the top level of Ruby**.

In the example above, the Top Method *Baba.my_process* is implemented in the module *Baba* and is named *my_process*. This means that the top method is defined as:

```
def Baba.my_process(...)
  Do something...
end
```

The Top Method is the method performing the execution of the task on a file. This is the minimum requirement.

For complex tasks, you can however specify **additional methods**. The name of these optional methods must respect a naming convention, by simply adding a **suffix** to the name of the Top method:

- **Params method:** this method will manage parameters for the task. Suffix is `__params` (ex: *Baba.my_process__params*).
- **Start method:** this method will be called before processing the files. This is useful to prepare the environment before job execution. Suffix is `__start` (ex: *Baba.my_process__start*).
- **Finish method:** this method is called after processing the files. This is useful to clean up the environment after job execution. Suffix is `__finish` (ex: *Baba.my_process__finish*).

5.2. Task Method Arguments

The **Top Method** may have 0, 1 or 2 arguments:

- `task_method()`: no argument
- `task_method(file)`: the single argument is the full path of the file to be processed
- `task_method(file, task_context)`: the second argument is a *TaskContext* object, which gives you access to all information related to the context of execution.

Whether you use arguments or not, you can always retrieve the Current File being processed and the Task Context by using the two methods anywhere in your code:

- `task_context = F6_FredoBatch.task_context`
- `current_file = F6_FredoBatch.current_file` or
`current_file = task_context.current_file`

Note also that when the Task is processing a Sketchup file (.skp) and that the task requires opening the Sketchup file, you can also retrieve the file path by calling: `Sketchup.active_model.path`.

The **Params Method** has 1 argument:

- `task_method__params(hsh_params)`: the argument is Hash array with the parameter keys and values (see section 5.5 below and the Appendix)

The **Start Method** may have 0 or 1 argument:

- `task_method__start()`: no argument
- `task_method__start(task_context)`: the argument is the *TaskContext* object.

The **Finish Method** may have 0 or 1 argument:

- `task_method__finish()`: no argument
- `task_method__finish(task_context)`: the argument is the *TaskContext* object.

5.3. Error Handling

FredoBatch calls all these methods within a `begin...rescue` block, so that potential errors are caught and will be displayed in the Result Summary dialog.

If an error occurs during the Params or Start methods, the job is aborted.

If an error occurs during the Top method, the file processing is stopped and FredoBatch goes to the next file.

If an error occurs during the Finish method, the error is reported in the Result Summary dialog.

The display of the error gives access to the error message and error backtrace.

5.4. Task Context

The **Task Context** is a Ruby object configured by FredoBatch to give access to relevant information in the context of the method call, but also to interact with FredoBatch processing.

The Task Context (`ctx`) exposes the following methods:

Information Methods

- `ctx.current_file`: path of the file under processing
- `ctx.name`: name of the current task
- `ctx.require_opening_skp?`: does the task require to open Sketchup files?
- `ctx.require_backup?`: does the task require backup?
- `ctx.extensions`: list of file extensions supported by the task if any
- `ctx.get_params`: get the parameters of the task (Hash array, as defined in the Params method of the task, with value populated by the user).
- `ctx.list_of_files`: list of all files participating to the job

Log Methods

- `ctx.log(message, type)`: log a message, which will appear in the Result Summary dialog or the Execution dialog or both, depending on the value of `type`: `:dialog` or `:result`. Useful to indicate the details of the task process.
- `ctx.logR(message)`: equivalent to `ctx.log(message, :result)`.
- `ctx.logW(message)`: equivalent to `ctx.log(message, :dialog)`.

Attribute Methods

These two methods can be used by the task to store and retrieve any information, which may be useful if the task is spread over different chunks of code. `value` can be any Ruby object.

- `ctx.set_attribute(attr, value)`: set the value of an attribute
- `ctx.get_attribute(attr)`: get the value of an attribute

Async Call

- `ctx.callme(delay, &continuation_proc)`: This method allows to give back control to the UI during a specified delay before continuing the task processing. It can be used if the task processing is long (see *Exemple 2 for illustration*).
- `ctx.wait_until(delay, nb_trials, &wait_proc)`: This method allows to give back control to the UI and continue the task processing when a condition is met (based on `wait_proc`). The check is performed periodically based on the specified `delay`. By precaution, the job is aborted after `nb_trials` attempts (see *Exemple 4 for illustration*).
- `ctx.wait_interactive(message, hconfig, &validation_proc)`: This method only works when Sketchup models are processed. It stops the processing and gives back control to the user to make adjustments and continue the processing (see *Exemple 4 for illustration*).

5.5. Specifying Parameters for the Task

It is common that a Task requires parameters that must be asked to the user.

One approach is that the task manages parameters on its own, by including the code to display a parameter dialog box and collect the answer. The display of the dialog would ideally be placed in the **Start method**. If some parameters are specific to the file being processed, a parameter dialog would also be displayed in the Top processing method.

FredoBatch provides a fast and easy way to prompt the user for parameters. This is the purpose of the Params method. This approach has the additional benefit to allowing the creation of Param tasks, by letting the user create tasks where the parameters are pre-populated.

The name of the Params method is defined after the name of the top method, with a suffix `__params`. It has one argument, which is a Hash array of parameter symbol and value, usually as the default set of parameters.

The purpose of the Params method is to build instructions for FredoBatch to display the parameters in a dialog, and within the Task editor dialog when creating a Param task.

Below is an example of a Param method for the Built-in task Purge Unused, where there are 4 parameters as checkbox for materials, components, tags and styles:

```
#PURGE UNUSED: Params method
def self.do__params(hparams=nil)
  @hparams_default = { :material => true, :component => true, :tag => true,
                      :style => true, :environment => true}

  #Default Parameters
  hparams = {} unless hparams
  hparams = @hparams_default.update(hparams)

  #Instructions for the Params dialog
  title = 'Purge Unused'
  msg = 'Select the scope of the Purge-Unused'
  hsh_dialog = { :width => 450, :message => msg, :unique_key => 'F6Do_purge_unused' }
  instructions = []
  instructions.push [:checkbox_single, :material, 'Materials', hparams[:material]]
  instructions.push [:checkbox_single, :component, 'Components', hparams[:component]]
  instructions.push [:checkbox_single, :tag, 'Tags', hparams[:tag]]
  instructions.push [:checkbox_single, :style, 'Styles', hparams[:style]]
  [title, instructions, hsh_dialog]
```

The Params method returns:

- `title`: a **title** for the dialog box
- `instructions`: a list of specifications for each parameter as an array of:
 - a **type** of parameters (here a single checkbox)
 - a **symbol** for the parameter, which will be the key in the Params Hash array
 - a **label**, as a string
 - an **initial value**
 - an **optional array of additional specifications** (not used in the example above)
- `hsh_dialog`: an optional Hash array containing **some configuration properties for the dialog** (here, there is a message, a dialog width and a unique key).

The corresponding dialog box is:



FredoBatch provides several types of parameters, checkbox, multi-checkboxes, radio button, text fields, numeric fields (integer, float, length) with an optional slider, combo, directory selection, file selection, See the Appendix for details.

6. Examples of Custom Tasks

Here are a few complete examples of Custom Tasks, from simple to more complex. This is illustrative and for the purpose of describing the principles of FredoBatch.

6.1. Example 1: Convert DWG/DXF files to Sketchup models

This method would **take a list of DWG and DXF files and create Sketchup models from them**. Each skp file is created in the same directory as the DWG/DXF file and with the same name. So,

- there are no parameters required for the task,
- There is nothing to pre-process and post-process.
- The task does NOT require opening SKP files
- The task does NOT require backup (since the DWG files are not touched)

Therefore, a **simple Top method is sufficient**.

The screenshot shows the 'Edit Custom Task' dialog box with the following configuration:

- Task Identification:**
 - Category: General Purpose
 - Name: Convert DWG/DXF to SKP
 - Description: Convert a set of DWG/DXF files to Sketchup models (same directory, same basename)
- Script Properties:**
 - Ruby File: `Baba.rb` (C:/Google Drive/Ruby/Ruby/Models/FredoBatch)
 - Top Method: `MyModule.convert_dxf_dwg_to_skp`
- Filter on SU versions and Files by allowed extensions:**
 - File Extensions: `dwg dxf`
 - SU Versions: SU2017 or above
- Task Processing:**
 - Sketchup Files: Open Sketchup .skp file before running the task
 - Backup: Backup files for possible restore

Here is the code for the top method:

```
module MyModule

def MyModule.convert_dxf_dwg_to_skp(file_dxf_dwg, task_context)
  #Path to the SKP file
  file_skp = file_dxf_dwg.sub(/dwg\Z/i, 'skp').sub(/dxf\Z/i, 'skp')

  #Open a new file
  Sketchup.file_new

  #Import the DWG
  model = Sketchup.active_model
  model.import(file_dxf_dwg, false)

  #Save the file
  model.save(file_skp)

  #Log the information
  task_context.log("Converted DWG/DXF to SKP: <b><!blue>#{File.basename(file_skp)}<!></b>")

  #Close the file(on mac)
  model.close if RUBY_PLATFORM =~ /darwin/i
end

end #module MyModule
```

The Results Summary dialog will show as:



File Processed	Duration
<input checked="" type="checkbox"/> MyDWG - Copy (2).dwg C:/Google Drive/Ruby/Ruby/Models/FredoBatch	0.960 s
<input checked="" type="checkbox"/> Convert DWG/DXF to SKP <input checked="" type="checkbox"/> Converted DWG/DXF to SKP: MyDWG - Copy (2).skp	0.960 s
<input checked="" type="checkbox"/> MyDWG - Copy (3).dwg C:/Google Drive/Ruby/Ruby/Models/FredoBatch	0.699 s
<input checked="" type="checkbox"/> Convert DWG/DXF to SKP <input checked="" type="checkbox"/> Converted DWG/DXF to SKP: MyDWG - Copy (3).skp	0.699 s
<input checked="" type="checkbox"/> MyDWG - Copy (4).dwg C:/Google Drive/Ruby/Ruby/Models/FredoBatch	0.519 s

Done

6.2. Example 2: Standard Scenes

This task illustrates the use of the *params* methods and the processing of Sketchup models with modification and potential rollback.

The task takes a list of SKP model files and creates scenes with standard cameras.

We only need a *Params method* and a *Process method*. The Task will require to open the skp model files and a Backup so that we can roll back the job.

The screenshot shows the 'Edit Custom Task' dialog box. It has a title bar 'Edit Custom Task' and a subtitle 'Edit Custom Task'. Below the subtitle is the instruction 'Enter the parameters for the Custom task'. The dialog is divided into four main sections: 'Task Identification' with 'Category' (General Purpose), 'Name' (Standard Scenes), and 'Description' (Create Standard Scenes in Sketchup models); 'Script Properties' with 'Ruby File' (Baba.rb) and 'Top Method' (Baba.standard_scenes); 'Filter on SU versions and Files by allowed extensions' with 'File Extensions' (skp) and 'SU Versions' (SU2017 or above); and 'Task Processing' with 'Sketchup Files' (checked) and 'Backup' (checked). A 'Close' button is located at the bottom right.

Params method

Parameters are:

- The list of Standard Cameras
- An optional Prefix for the Scene naming
- An optional Suffix for the Scene naming
- Option to perform a Zoom Extents

The dialog will be displayed as:

The screenshot shows the 'Standard Scenes' dialog box. It has a title bar 'Standard Scenes' and a subtitle 'Standard Scenes'. Below the subtitle is the instruction 'Enter parameters for the creation of standard scenes'. The dialog is divided into three main sections: 'Standard Cameras' with checkboxes for Iso, Front, Top, Right, Left, Back, and Bottom; 'Scene Naming' with 'Scene Name Prefix' (pre_) and 'Scene Name Suffix' (_post); and 'Scene Properties' with 'Zoom Extents' (checked) and 'Position' (At Beginning). 'Cancel' and 'Assign Parameters' buttons are located at the bottom right.

The code for the *Params method* will be:

```
def Baba.standard_scenes_params(hparams)
  #List of cameras - Variable is at module level, so available in all methods
  @lst_standard_cameras = [:Iso, :Front, :Top, :Right, :Left, :Back, :Bottom]

  #Default Parameters
  hparams_default = { :cameras => @lst_standard_cameras, :zoom_extents => true, :position => :begin }
  hparams = {} unless hparams
  hparams = hparams_default.update(hparams)

  #Instructions for the Params dialog
  options_camera = @lst_standard_cameras.collect { |symb| [symb, symb.to_s]}
  hspecs_cameras = { :lst_options => options_camera }
  options_position = [[:begin, 'At Beginning'], [:end, 'At End']]
  hspecs_position = { :lst_options => options_position }
  title = 'Standard Scenes'
  msg = 'Enter parameters for the creation of standard scenes'
  hsh_dialog = { :width => 600, :message => msg }

  instructions = []
  instructions.push [:bandeau, 'Standard Cameras']
  instructions.push [:checkbox_multi, :cameras, 'Cameras', hparams[:cameras], hspecs_cameras]
  instructions.push [:bandeau, 'Scene Naming']
  instructions.push [:text, :prefix, 'Scene Name Prefix', hparams[:prefix]]
  instructions.push [:text, :suffix, 'Scene Name Suffix', hparams[:suffix]]
  instructions.push [:bandeau, 'Scene Properties']
  instructions.push [:checkbox_single, :zoom_extents, 'Zoom Extents', hparams[:zoom_extents]]
  instructions.push [:radio, :position, 'Position', hparams[:position], hspecs_position]

  [title, instructions, hsh_dialog]
end
```

Process method

The Process method just creates the scenes in each model. In the detail, we mark scenes which are created by the task so that we replace them if they have already been created by earlier job executions.

```
def Baba.standard_scenes(skp_file, task_context)
  #At this stage, the Sketchup model is open
  model = Sketchup.active_model
  supages = model.pages
  dico = 'MarkStandardScenes'

  #Get the params
  hparams = task_context.get_params
  lst_cameras = hparams[:cameras]
  zoom_extents = hparams[:zoom_extents]
  prefix = hparams[:prefix]
  suffix = hparams[:suffix]
  position = hparams[:position]
  lst_standard_cameras = (position == :end) ? @lst_standard_cameras : @lst_standard_cameras.reverse

  #Loop on cameras
  lst_standard_cameras.each do |camera|
    #Find a scene marked with the camera attribute - Delete it if it exists
    supage_prev = supages.to_a.find { |supage| supage.get_attribute(dico, camera.to_s) }
    supages.erase(supage_prev) if supage_prev

    #Skip the scene if not requested
    next unless lst_cameras.include?(camera)

    #Create the scene
    name = "#{prefix}#{camera}#{suffix}"
    ipos = (position == :end) ? supages.length : 0
    Sketchup.send_action("view#{camera}:")
    model.active_view.zoom_extents if zoom_extents
    supage = supages.add name, PAGE_USE_ALL, ipos
    supage.set_attribute(dico, camera.to_s, true)
    action = (supage_prev) ? 'Replaced' : 'Created'
    msg = "#{action} Scene <b><!--blue>#{name}<!--</b>"
    msg += ' with <b>Zoom Extents</b>' if zoom_extents
    task_context.log msg
  end

  #No need to save the model. FRedoBatch will do it if the model has been modified
end
```


The Result Summary dialog will show the details of the job execution:

File Processed	Duration
<input checked="" type="checkbox"/> Animator rubik_cube Enea - 2 - SU15.skp <small>C:/Google Drive/Ruby/Ruby/Models/Animator Beta</small>	0.820 s
<input checked="" type="checkbox"/> Animator rubik_cube Enea - 2 - SU16.skp <small>C:/Google Drive/Ruby/Ruby/Models/Animator Beta</small>	0.791 s
<input checked="" type="checkbox"/> Backup File	0.009 s
<input checked="" type="checkbox"/> Opening File	0.212 s
<input checked="" type="checkbox"/> Standard Scenes	0.328 s
<input checked="" type="checkbox"/> Replaced Scene <i>pre_Bottom_post</i> with Zoom Extents	
<input checked="" type="checkbox"/> Replaced Scene <i>pre_Back_post</i> with Zoom Extents	
<input checked="" type="checkbox"/> Replaced Scene <i>pre_Left_post</i> with Zoom Extents	
<input checked="" type="checkbox"/> Replaced Scene <i>pre_Right_post</i> with Zoom Extents	
<input checked="" type="checkbox"/> Replaced Scene <i>pre_Top_post</i> with Zoom Extents	
<input checked="" type="checkbox"/> Replaced Scene <i>pre_Front_post</i> with Zoom Extents	
<input checked="" type="checkbox"/> Replaced Scene <i>pre_Iso_post</i> with Zoom Extents	
<input checked="" type="checkbox"/> Saving File	0.242 s
<input checked="" type="checkbox"/> Animator rubik_cube Enea - 2 - SU17.skp <small>C:/Google Drive/Ruby/Ruby/Models/Animator Beta</small>	0.749 s
<input type="checkbox"/> Animator rubik_cube Enea - 3 - SU15.skp	0.804 s

Since the input skp files have been modified, the **Rollback Job** option is available.

6.3. Example 3: Model Gallery (simple version)

*This task illustrates the use of the **start**, **finish** and **params** methods. It also shows the bases of error handling and the asynchronous processing.*

The task takes a list of SKP model files and creates a single PDF, where each page displays a view of the models with a given camera.

The generation of the PDF is done via Layout, based on a **Layout template**.

The Template Layout file is based on a page, with a **rectangle** for the viewport, and a **title box** to display the path of each model. All visual elements are on Layer *On Every Page*.



Params method

Parameters are:

- The path to the Layout template file
- The camera for the view (iso, top, left,)
- The path to the PDF file

The code would thus be:

```
def MyModule.model_gallery_params(hparams)
  #Default Parameters
  hparams_default = { :camera => :iso, :use_date_suffix => true }
  hparams = {} unless hparams
  hparams = hparams_default.update(hparams)

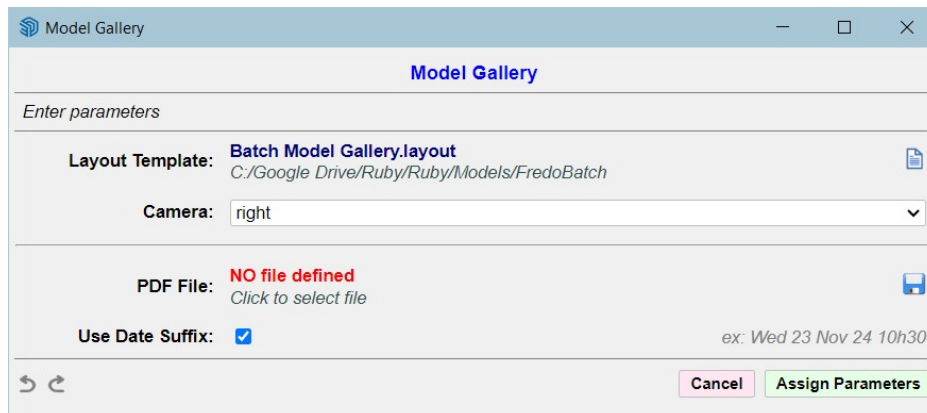
  #Instructions for the Params dialog
  lst_cameras = ['iso', 'top', 'right', 'left', 'bottom']
  klist_camera = lst_cameras.collect { |symb| [symb, symb.to_s, "tip #{symb}"]}
  hspecs_suffix = { :comment => 'ex: Wed 23 Nov 24 10h30' }

  title = 'Model Gallery'
  msg = 'Enter parameters'
  hsh_dialog = { :width => 600, :wid_comment => 200, :message => msg }

  instructions = []
  instructions.push [:file_open, :layout_template, 'Layout Template',
                    hparams[:layout_template]]
  instructions.push [:combo, :camera, 'Camera', hparams[:camera], klist_camera]
  instructions.push [:separator]
  instructions.push [:file_save, :pdf_path, 'PDF File', hparams[:pdf_path]]
  instructions.push [:checkbox_single, :use_date_suffix, 'Use Date Suffix',
                    hparams[:use_date_suffix], hspecs_suffix]

  [title, instructions, hsh_dialog]
end
```

The Parameter dialog will display as:



Start method

At the start of the task:

- we check that the PDF path has been provided
- we create a Layout document, based on the provided template.
- we identify the viewport rectangle where to display the model views (e.g. the largest rectangle within the template page).

Although we could find alternatives to manage this information (layout document and viewport bounds), the code below shows how to use the *Task Context* to store and retrieve this information (`task_context.set_attribute`).

The code would thus be:

```
def MyModule.model_gallery_start(task_context)
  #Retrieve the parameters and the Layout Template path
  hparams = task_context.get_params
  layout_template_path = hparams[:layout_template]

  #Check the PDF file
  pdf_path = hparams[:pdf_path]
  if !pdf_path || pdf_path.empty?
    return [:abort, 'PDF file path has not been provided']
  end

  #Create a Layout document based on this template
  document = Layout::Document.new(layout_template_path)
  task_context.set_attribute(:document, document)

  #On the first page of this template, find the biggest rectangle
  #This will be the bounds for creating viewports (i.e. Sketchup Models)
  template_page = document.pages[0]
  lst_rects = template_page.entities.find_all { |e| e.instance_of?(Layout::Rectangle) }

  #We abort the job if there is no rectangle in the template.
  if lst_rects.empty?
    return [:abort, 'No viewport defined in the Layout Template']
  end

  #Identify the biggest rectangle to get the bounds for the viewport
  vp_bounds = nil
  area_min = 0
  lst_rects.each do |rect|
    bounds = rect.bounds
    area = bounds.width * bounds.height
    if area > area_min
      vp_bounds = bounds
      area_min = area
    end
  end
  task_context.set_attribute(:vp_bounds, vp_bounds)

  #Set the date for the document
  ftext_current_date = template_page.entities.find do |e|
    e.instance_of?(Layout::FormattedText) && e.plain_text =~ /\$\$current_date\$\$/
  end
  if ftext_current_date
    sdate = Time.now.strftime "%a %d.%b.%Y %H:%M:%S"
    ftext_current_date.rtf = ftext_current_date.rtf.sub(/\$\$current_date\$\$/, sdate)
  end

  #Open a New file in Sketchup to have a clean environment
  Sketchup.file_new

  #Success
  true
end
```

During the Start sequence, if anything goes wrong, you can abort the job processing by returning `:abort` or `[:abort, message]`. The message will be displayed to the user.

Process method

The Process method is here **split into 2 methods**, to illustrate the use of the `task_context.call_me` asynchronous call, which gives back control to the UI.

Since we do not want to alter the original Sketchup files, we copy them to a temporary directory and open them in Sketchup. We then set the specified camera and create the corresponding page in the Layout document (viewport and title).

```
def MyModule.model_gallery(skp_file, task_context)
  #Copy the Sketchup file to a temporary place
  skp_tmp_path = File.join(Sketchup.temp_dir, File.basename(skp_file).sub(/\\.skp/i, "_temp.skp"))
  FileUtils.copy skp_file, skp_tmp_path

  #Open the temp sketchup file
  Sketchup.open_file skp_tmp_path, with_status: true

  #Set the view to the required camera and perform a zoom extent
  model = Sketchup.active_model
  camera = task_context.get_params[:camera]
  Sketchup.send_action("view#{camera}")
  model.active_view.zoom_extents
  task_context.log("Created the view <b><!--blue-->#{camera}<!-->/b>")

  #Save the temporary Sketchup file
  model.save

  #Async call
  task_context.callme { MyModule.model_gallery_phase2(skp_file, skp_tmp_path, task_context) }
end

def MyModule.model_gallery_phase2(skp_file, skp_tmp_path, task_context)
  #Retrieve the document and the viewport bounds
  document = task_context.get_attribute(:document)
  vp_bounds = task_context.get_attribute(:vp_bounds)
  pages = document.pages

  #Add a page for the Sketchup model
  basename = File.basename(skp_file, '.skp')
  page = pages.add basename

  #Create the viewport (Sketchup Model)
  layer_default = document.layers.find { |layer| layer.name == 'Default' }
  layport = Layout::SketchUpModel.new(skp_tmp_path, vp_bounds)
  layport.current_scene = 0
  document.add_entity(Layport, layer_default, page)

  #Set the title text to the skp file path (painful as no copy in the Layout API)
  rect_title = page.entities.find do |e|
    e.instance_of?(Layout::Rectangle) && e.style.fill_color.to_a == [0, 0, 0, 255]
  end
  if rect_title
    bb = rect_title.bounds
    x1, y1 = bb.upper_left.to_a
    x2, y2 = bb.lower_right.to_a
    center = Geom::Point2d.new(0.5 * (x1 + x2), 0.5 * (y1 + y2))
    ftext = Layout::FormattedText.new(skp_file, center, Layout::FormattedText::ANCHOR_TYPE_CENTER_CENTER)
    document.add_entity(ftext, layer_default, page)
    style = ftext.style
    style.font_family = "Verdana"
    style.text_color = 'white'
    style.text_bold = true
    style.font_size = 12.0
    ftext.style = style
  end

  #Open a new Sketchup file and Delete the temporary Sketchup file
  model.close if RUBY_PLATFORM =~ /darwin/i
  Sketchup.file_new
  File.delete skp_tmp_path
  task_context.log("Created the page for model <b><!--blue-->#{basename}<!-->/b>")
end
```


Finish method

After all models are processed, we remove the first page (template page) and export the Layout document to a PDF file, which is open to the user.

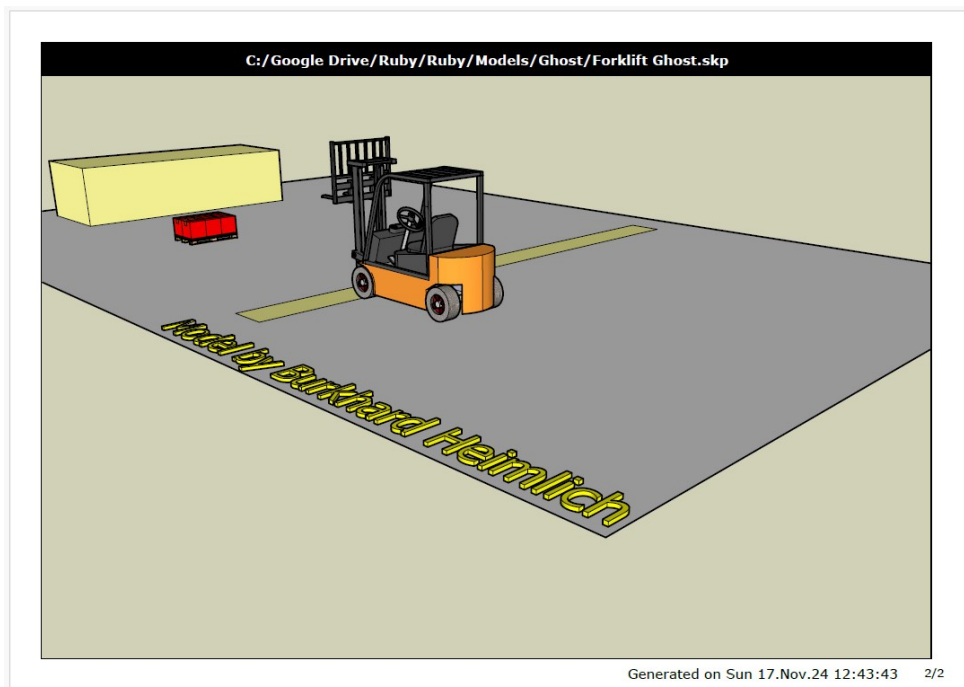
```
def MyModule.model_gallery_finish(task_context)
  #Retrieve the document and the PDF path
  document = task_context.get_attribute(:document)

  #Remove the first page
  document.pages.remove(0)

  #Generate the PDF
  hparams = task_context.get_params
  pdf_file_path = hparams[:pdf_path]
  if hparams[:use_date_suffix]
    sdate = Time.now.strftime "%d.%b.%y-%H.%M.%S"
    pdf_file_path = pdf_file_path.sub(/\.pdf\/z/i, "_#{sdate}.pdf")
  end
  document.export(pdf_file_path, {})

  #Open the PDF
  UI.openURL pdf_file_path
end
```

Below, one page of the generated PDF:



6.4. Example 4: Model Gallery (advanced version)

This task is a variant of the previous task with more advanced features. It illustrates the use of the **asynchronous interactive processing** (`wait_interactive`). Model gallery is available as a Built-in task in FredoBatch v1.2. See [this video](#) for illustration.

Interactive processing is a means to give control to the user when processing files.

In the case of the Model Gallery example, the task will open each Sketchup model and let the user choose the camera in the model. When done, the user will press a button **Validate** to continue the job (or skip the file, or abort the job):



Parameters can also be used in the Interactive Dialog, here Zoom Extents, to trigger (or not) a zoom adjustment of the user-defined camera in order to fit the model within the layout viewport.

To enter the interactive mode, the execution proc should invoke:

```
task_context.wait_interactive(msg, hconfig, &validation_proc).
```

The block `validation_proc` will be invoked when the user clicks on the Button **Validate**. The convention is that this proc should return:

- Either a `continuation_proc` to be called for **continuing the job**,
- Or any other value to **stay in the interactive mode**, for instance, if the user did not perform what is expected by the task.

Some parameters can be passed to the interactive mode with the hash array `hconfig`. For instance, you can pass the instructions to prompt the user for some parameters. In the example of Model Gallery,

```
instructions = []
instructions.push [:checkbox_single, :zoom_extents, 'Zoom Extents', zoom_extents]
hconfig[:instructions] = instructions
task_context.wait_interactive(msg, hconfig) do
  self.do_validate_view(skp_file, skp_tmp_path, task_context)
end
```

In the example above, the validation proc `do_validate_view` does not do anything complex, since the interactive mode is triggered to the user for possibly changing the camera (via pan, orbit, change the current scene, etc...). So, there is no negative condition preventing the continuation of the job. So, it is rather simple and just return the **continuation proc**.

```
#MODEL GALLERY: Validate the view
def self.do_validate_view(skp_file, skp_tmp_path, task_context)
  proc { self.do_phase2(skp_file, skp_tmp_path, task_context, :custom) }
end
```

Note that using **proc** allows to transmit context variables through arguments.