User Guide - LWC UI Orchestrator

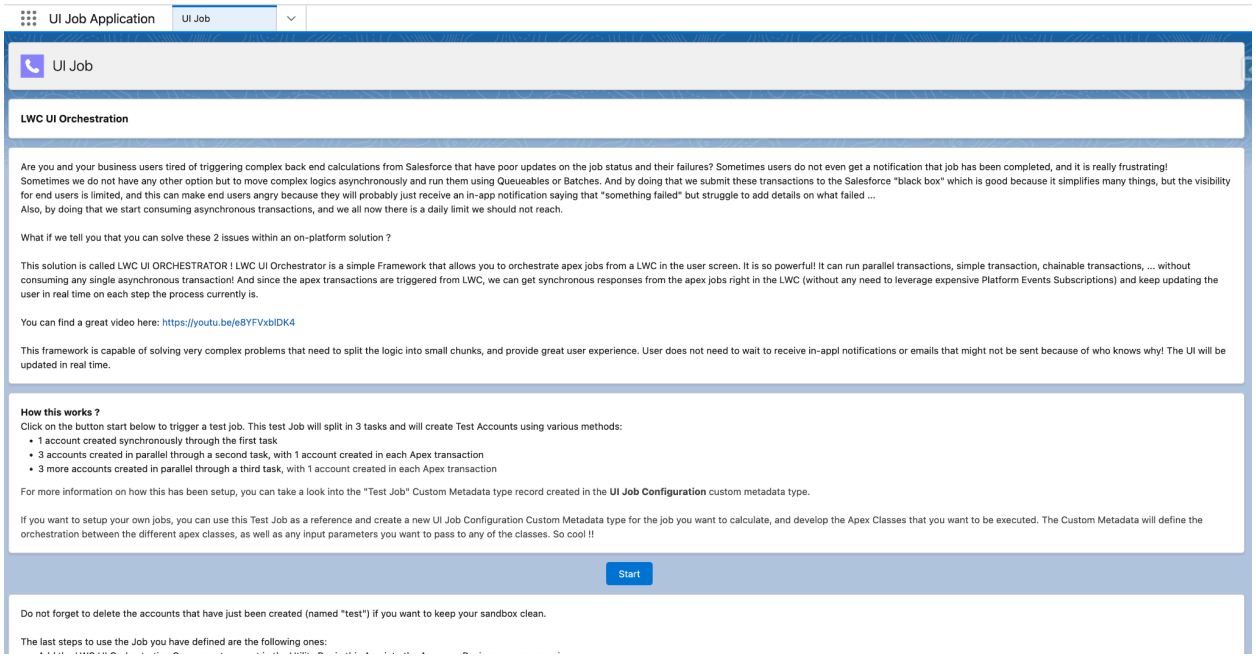
Intro

Are you and your business users tired of triggering complex back end calculations from Salesforce that have poor updates on the job status and their failures? Sometimes users do not even get a notification that a job has been completed, and it is really frustrating!
Sometimes we do not have any other option but to move complex logics asynchronously and run them using Queueables or Batches. And by doing that we submit these transactions to the Salesforce "black box" which is good because it simplifies many things, but the visibility for end users is limited, and this can make end users angry because they will probably just receive an in-app notification saying that "something failed" but struggle to add details on what failed ...
Also, by doing that we start consuming asynchronous transactions, and we all know there is a daily limit we should not reach.
What if we tell you that you can solve these 2 issues within an on-platform solution ?
This solution is called LWC UI ORCHESTRATOR ! LWC UI Orchestrator is a simple Framework that allows you to orchestrate apex jobs from a LWC in the user screen. It is so powerful! It can run parallel transactions, simple transactions, chainable transactions, ... without consuming any single asynchronous transaction! And since the apex transactions are triggered from LWC, we can get synchronous responses from the apex jobs right in the LWC (without any need to leverage expensive Platform Events Subscriptions) and keep updating the user in real time on each step the process currently is.

This framework is capable of solving very complex problems that need to split the logic into small chunks, and provide great user experience. Users do not need to wait to receive in-app notifications or emails that might not be sent because of who knows why! The UI will be updated in real time.

Demo Run

If you want to see the framework in action to understand a little bit more about how it works, after installing the package, please go to App Launcher -> UI Job Application and follow the steps and instructions shown in the UI Job App Page



High Level Approach

The idea is to be able to deliver a very complex calculation logic - aggregating records, calculating values, creating or updating records - which we will call a **Job** and divide this main job that you are planning to deliver into **Tasks**.
Tasks and Jobs are 2 custom metadata types (UI Job Configuration and UI Task Configuration) that we have created for you to be able to easily define your jobs and the tasks inside your jobs. Each task will be assigned to an Apex Class that will be in charge of executing the logic you want to be done as part of that task.
Tasks are executed sequentially when defined as part of a job.
Tasks can have multiple types, which will describe to the LWC UI Controller how the Apex methods should be called from the LWC:

- Synchronous: the task will be called only once
- Parallel: That task will executed a set of sub-tasks in parallel. The parallel subtasks can be associated with different or the same Apex classes. This one is very powerful if you need, for instance, to insert/delete data in mass from salesforce, and you can do it in parallel.
- Chain: Allows to execute sequentially the same apex class (for different set of records). Better than queueable because it provides real time visibility on the context of the current chained execution
- Queueable: Allows to execute Apex Queeuable from the LWC UI Controller

High Level Steps to leverage the app

- Add the LWC uiJobController in the utility bar of the apps that are used by your business users. This LWC will listen for events that will be sent from another LWC you will place in the last step of this installation
- Define the logic you want to build (UI Job) and split it into tasks (UI Tasks). Lastly, define for each task, if you want to run them in parallel, synchronously, chain or queueable. Create the custom metadata types of your Job structure
- Each task will need to be associated to an Apex Class that extends the class UIJobTaskAbstract. The name of the class will be also defined each Task custom metadata type record. This class will override a method run to run your logic. You can check the UIJobTaskClassSample for an easy example on how that works. Each Apex Class can receive input parameters defined in a JSON file, along with the recordId (that can come from the publisher that triggered the process)
- The last step is to trigger the process we have defined in the Job custom metadata type. For this, we can simply add the LWC uiJobCaller into any record page or App page, and while configuring it, we can add the name of the Job Custom Metadata type we want to trigger. Alternatively, we can expand this LWC, so when the event is published and captured by the uiJobController LWC, we can enrich the data passed and send a JSON with additional parameters, or the record ID. This helps to pass data across the functions

Detailed & technical Documentation

UI Job Framework

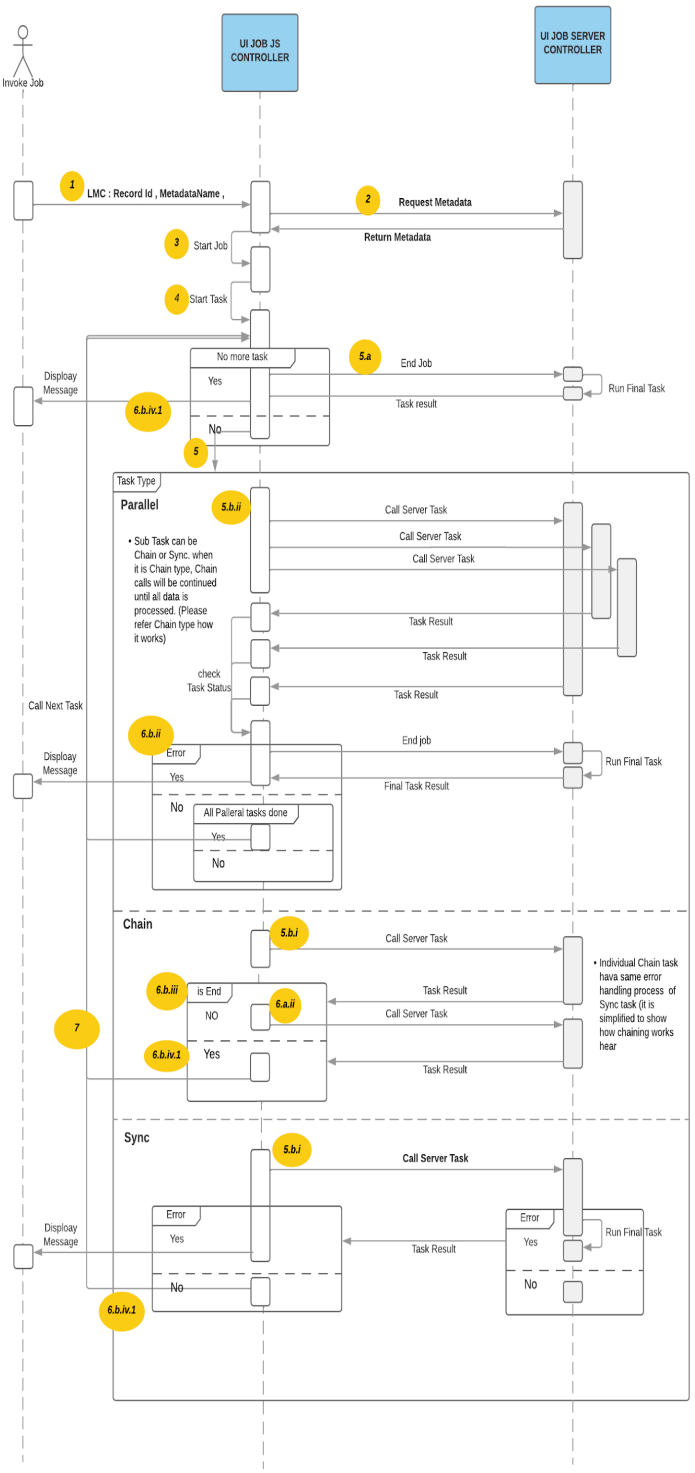
This framework has been implemented for large and complex processes that need to do complex calculations on large volumes of data. Compared to a standard framework using back-end asynchronous jobs, who have little control and visibility over which task will be executed next, this framework moves the orchestration of the tasks to the browser, so the orchestration is done directly in javascript in a LWC from the user browser. This:

- Increases visibility of task and job status
- Gives more flexibility on whether to execute tasks in parallel or sequentially, and its dependencies (compared to traditional Queueable or Asynchronous Apex)

- Consumes less processing that counts towards limits. Indeed, the tasks executed from this framework do not count as asynchronous transactions, where Salesforce has a limit of 250K asynchronous transactions per day.

Framework Overview

When the module (Flow/ Button etc) is send Event message by using CallUIJob LMC, UI Job LWC send a request to get Job and Task configuration to Server.
Start Job method will be called on JS controller when Job configuration is received.
Start Job Method calls the first Task with task configuration. JS controller will handle differently depending on the Type of Task.



1. *UI Job will be invoked through Lighting Message Channel and Invoker will pass*
   a. *Record Id*
   b. *Metadata Name*
2. *When the event through LMC is received InitPage method is invoked and initPage method requests the Configuration data of Job to UI Job Server controller*
3. *UI Job JS controller start the job by calling startJob JS method.*
4. *Start Job method will reset the status of JOB on JS controller and call the first Task by calling startTask method. I*
5. *startTask processes*
   a. *If there is no more Task to call startTask will call End Job*
   b. *If there is Task to call it will calling based on type of task*
      i. *Parallel : startTask get subtasks information from configuration and keep calling callServerTasks Method until all sub tasks are invoked*
      ii. *Non Parallel : startTask invoke callServerTask once*
6. *callServerTask processes a Task based on Type of Task*
   a. *Before Invoking the Server Controller*
      i. *Parallel : if sub Task is Chain , get the output of invoked task of the current sub task and set as input*
      ii. *Chain : get the output of invoked task of the current task and set as input*
   b. *After receiving the response*
      i. *Set the result of task into task (if Parallel set it into sub task.*
      ii. *Task Type : Parallel*
         1. *Sub Task is Success ,*
            a. *Chain and not end : Call the sub Task again with the output of invoked task as input out next call.*
            b. *Chain is end or Not Chain : Mark Task is End successfully.*
         2. *Sub Task is Failed : Set the result of Sub Task And Task is Failed*

        a. *Call endTask with Fail status,*
        b. *Call endServerTask with error message (** Other type of Task no need to call endServerTask because endServerTask will be called in server when task is failed). Call endJob*
    iii. *Task Type : Chain*
        1. *Task is Success.*
            a. *Is Not End : Call the task again with out of invoked task as input of subsequent task call.*
            b. *Is End : Set Task is successfully end*
    iv. *Task Type is not Parallel*
        1. *Task is Success*
            a. *Call a next task by calling startTask*
        2. *Task is Failed*
            a. *Call endTask with Failed Status endTask will call endJob*
7. *When Task is ended*
    a. *Call Start Task (go back to 4)*

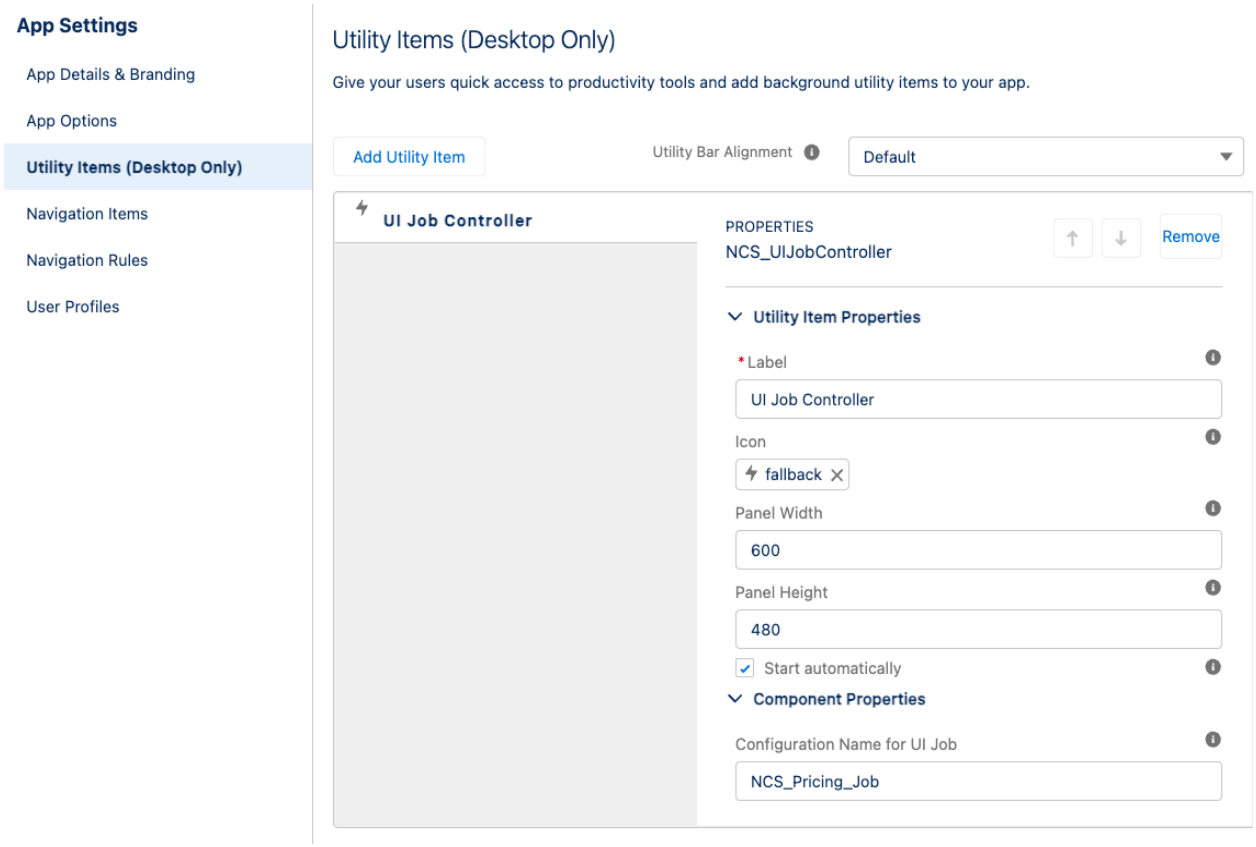
Components leveraged

Lightning Web Components

uiJobController

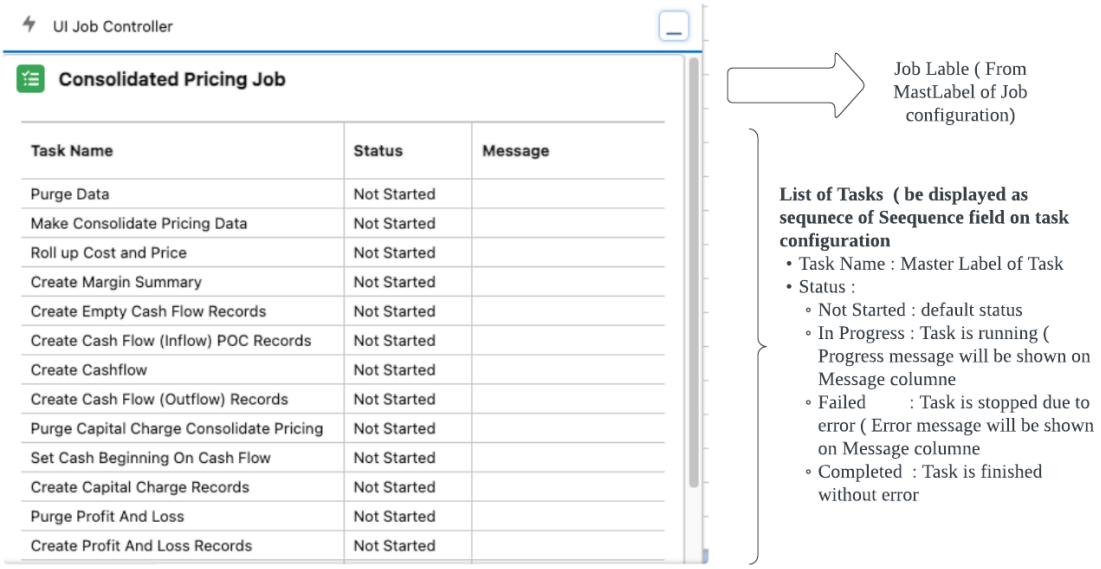
This LWC component only can be used in Console App and should be defined as Utility bar. It can't be used on the part of the record page or App.

Utility Item setting for UI Job Controller LWC



** Start automatically property should be set as true.
** Only Support Console App.

UI Job Controller during runtime



Permission Set

Two permissionSets are provided. Please find the detail below

Permission Set	**Description**
UI Job	Permission Set to use UI Job including - Access All Apex classes for UI Job - UI Job Status
UI Job Demo	Permission Set to access UI Job Demo Application and UI Job Tab

Lightning Components (LWC / Aura)

Two permissionSets are provided. Please find the detail below

Component	**Description**
callUIJobComponent	Calling UI Job Controller UI Job Controller should be registered before using this component

	Parameters are below - Configuration Name for UI Job : Developer name of UI Job defined in UI Job Configuration metadata - Label of Button : Label of button
UI Job Controller	Component which will be used to add UI Job controller into Record Page. Parameters are below - Enter Custom Meta Name : Developer name of UI Job defined in UI Job Configuration metadata - Whether Start Button is displayed: true (this configuration should be true) * it will not invoked by Caller component
uiJobCaller	Same function as callUIJobComponent (Align of button is center and not support change label)

Lightning Message Channel

CallUIJob

Message Channel to invoke UI Job. You can invoke this message channel to invoke the. certain UI Job. UI Job Controller LWC should be registered as Utility Item of current Application to invoke UI Job using this LMC.

Field	**Description**
recordIdStr	Id of record. (Required)

metaDataName	The API name of Job, This should be used to invoke the Job. (Required)
inputJSON	Input JSON String for Job (Optional). It will be passed as InputJSON argument of first task

Custom Metadata type

Leverage custom Metadatas named UI Job Configuration , UI Task Configuration to define the configuration for UI Job framework. The Task class should be created by extending from UIJobTaskAbstract to invoked by UI Job LWC.

UI Job Configuration

Field	**Description**
MasterLabel	Define the label of Job which will be displayed on UI Job LWC
Custom Metadata Record Name	The API name of Job, This should be used to invoke the Job.
Final Job Task Class	The name of Batch Class which will be run after this Job is finished (regardless whether there is error)
Final Task Param String 1	Define the first Static Parameter (Type : String) will be passed when Final Task Class is invoked (Optional)
Final Task Param String 2	Define the first Static Parameter (Type : String) will be passed when Final Task Class is invoked (Optional)
Final Task Param String 3	Define the first Static Parameter (Type : String) will be passed when Final Task Class is invoked (Optional)

UI Task Configuration

Field	Description
MasterLabel	Define the label of Task which will be displayed on UI Job LWC
Custom Metadata Record Name	The API name of Task
ClassName	The API name of Class which will be run
Is Flow	Define whether Task will use Flow instead of Class. When it is true, Param String 1 : Flow Name Please refer to UIJobTaskFlow_Demo for how you can make each type of task. Flow should have the following text resources. Input ● paramOne ● paramThree ● paramTwo ● recordId Output ● outputMessage ● outputJSON Input & Output ● lastString For Chain Task, the properties of outputMessage will be passed as Input resources for subsequent task calls.
Param String 1	Define the first Static Parameter (Type : String) will be passed when Class is invoked (Optional)

	Param String 1 - 3 can be used to use the one apex class for multiple tasks .
Param String 2	Define the second Static Parameter (Type : String) will be passed when Class is invoked (Optional) Param String 1 - 3 can be used to use the one apex class for multiple tasks .
Param String 3	Define the third Static Parameter (Type : String) will be passed when Class is invoked (Optional) Param String 1 - 3 can be used to use the one apex class for multiple tasks .
Parallel Job	Associated the child job definition (UI Job Custom Meta) which defined list of Tasks are run Parallely (Only when the Type of Task is Parallel)
Sequence	Define the sequence of Task in job, Tasks under the Job will be run by sequentially based on the value of this field
Type	Type of Task - Parallel : Tasks which is defined under Parallel Job are run parallelly - Sync : Task will invoked once to process all data - Chain : Task will called multiple time and individual call will process the part of data - Queueable : Task will be invoked by Queueable Apex - Bachable : Task will be invoked by Batch Apex

UI Job	The job which this task is belong to

Class

UIJobTaskAbstract Abstract Class

The Task class should be created by extending from UIJobTaskAbstract to invoked by UI Job LWC

Class Method

Method / Class	Arguments	Description
run	String recordIdStr : Record Id String inputJSON : input JSON (when the task is the first task, Input JSON will be from JOB invoker. When the task is not the first task, the Output JSON of the previous task will be passed as Input JSON.	Virtual method which should be overwritten when you create a Task Class. UI Job Framework will call this method to run the task.
ResultWapper	isSuccess (Boolean) : whether the Task is success / failed String message : error message / progress message Id jobId : job id of batch only for batch String outputJSON : Output JSON string boolean isEnd : whether all data has been processed : String lastString : Id/ indication of last	This Class is to return the result of Task

	processed data . it will be passed to the next task and the next task will use this to get next data set.	
successResult		This method will be used to create the Result Wrapper class instance. (for Sync type)
successResult	String lastId : It will be set as Last String and it will be passed to the next Task	This method will be used to create the Result Wrapper class instance with laststring.(for Chain Type)
successResult	String lastId : It will be set as Last String and it will be passed to the next Task String message : Message which will be displayed on the message column of task row (for user to indicate the progress of task)	This method will be used to create the Result Wrapper class instance with laststring and message. (For Chain Type and when you want to display a message for user to indicate progress)
raiseError	String message : Error message	This method will be used to raise an error from Task class.
convertJsonToMap	String inputJSON	This method convert JSON string into Map<String,Object>

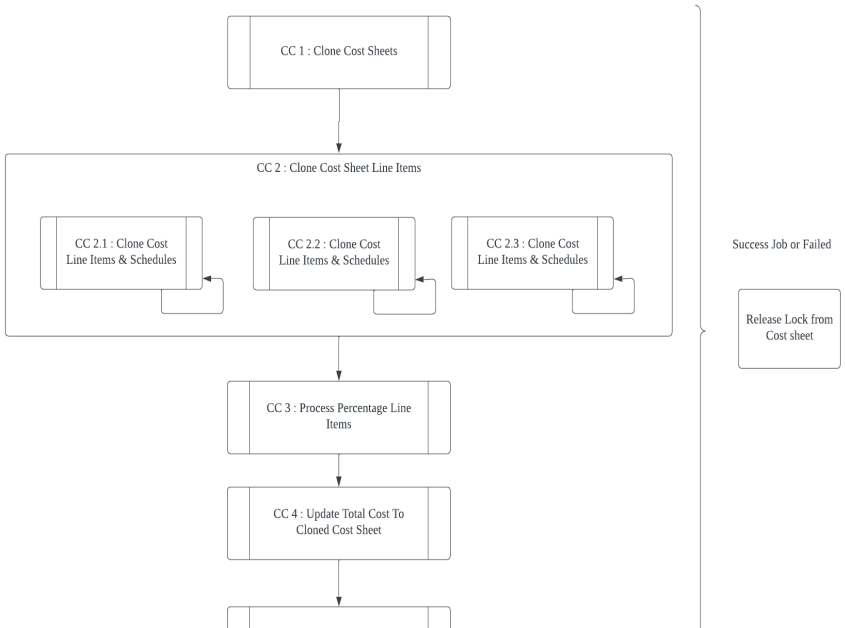
Class Attributes

Attribute	**Description**

String paramOne	The first Static Parameter (Type : String) which is defined on the Task Configuration
String paramTwo	The second Static Parameter (Type : String) which is defined on the Task Configuration
String paramThree	the third Static Parameter (Type : String) which is defined on the Task Configuration
String lastString	The last string output from the previous Task . this is only for chain
String taskType	Type of the task (please refer the detail of each type on UI Task Configuration Metadata section
String errorMessage	Error Message

Example on how to setup a new Job through this Framework

The example below show how to configure Framework to clone cost sheet



Clone of Cost Sheet should be
1 , Clone Consolidated Cost Sheet along with Child cost sheet
2. Copy all Line items and Schedules
- We make it as parallel steps therefore line items and schedules for 3 costsheets at the same time . also sometimes Cost sheet can have a lot of line items to schedule to avoid to make it long running we use chain

3. The cost of Percentage line item should be calculated from total cost therefore we create separate task for that
4. Total Cost of Costsheet should be include Percentage based line item and other type of line item therefore we make it run after processing Percentage line items.
5. Release lock of cloned cost sheet therefore user can use

Sample Code for Task

Example - Code of Demo class

```
// Class should be global for UIJobController can access it
//  Your class should be extended by  lwcorch.UIJobTaskAbstract
global with sharing class UIJobDemo_UITask  extends lwcorch.UIJobTaskAbstract{

//Override rum method
  public override ResultWrapper run(String recordIdStr,String inputJSON)
  {
      // Use parmOne (Parameter_String_1__c) in Task Configuration) to make one class can work in different tasks
      switch on paramOne {
        // 1. Purge all Demo data
        when 'purgeData' {
           return purgeData();
        }
        // This task will create 140 demo account with 140 DMLs
```

```
        // DML in loop to show tasks are running separately and Orchestrated by UI
        when 'createDemoAccount' {
            return createAccount();
        }
        // This task will update all demo account with chain tasks (100  per task);
        // DML in loop to show tasks are running separately and Orchestrated by UI
        when 'updateAccountChain' {
            return updateAccountChain(inputJSON);
        }
      }
      return successResult();

   }
    private ResultWrapper purgeData()
   {

      List<Account> queryResult = [Select Id from Account where name like 'LWC
Orchestration Test Account%'];
      if(Account.sObjectType.getDescribe().isDeletable() )
      {
         Database.delete(queryResult,true,AccessLevel.SYSTEM_MODE);
      }
      return  successResult();
   }
   /**
    insert 140 accounts . insert DML in loop to run just before hit DML limit .
    This function does not follow code best practice. it is created to show how UI
    task works
    */
//Example for Sync Task
   private ResultWrapper createAccount()
   {

        List<Account> accList = new List<Account>();
        String nameStr = 'LWC Orchestration Test Account';
        for(integer i=0 ; i< 140; i++)
        {
            insert as system new Account(Name=nameStr);
        }

        return successResult();
  }
```

```
    /**
    Update 100 accounts . update DML in loop . This function does not follow the
    code best practice. it is created as purpose to show how Chain task works
   **/
//Example for Chain Task
  private ResultWrapper updateAccountChain(String inputJSON)
  {

      Map<String,Object> inputMap = new Map<String,Object>();
      integer count = 1;
      integer chainNum = 1;
      if(String.isNotEmpty(inputJSON))
      {
//Covert Input JSON string into Map using convertJsonToMap function (from
Abstract)
         inputMap= convertJsonToMap(inputJSON);
         count = Integer.valueOf(inputMap.get('count'));
         chainNum = Integer.valueOf(inputMap.get('chainNum'));
      }
      List<Account> accList = new List<Account>();
      Id lastId =null ;
      String nameStr = 'LWC Orchestration Test Account';
      for(Account acc : [Select id from account
                         where name like 'LWC Orchestration Test Account%'
                         and Id > :lastString
                         Order by Id limit 100 ])
      {
          lastId = acc.id;
          acc.name = nameStr + ' Chain : ' + String.valueOf(chainNum).leftPad(3, '0')
+ ' recordNo : ' + String.valueOf(count).leftPad(3, '0');
          update as system acc;
          count +=1;
      }
      chainNum +=1;
      //Set the variable to pass to the next chain call.
      inputMap.put('count',count);
      inputMap.put('chainNum',chainNum);
      //Add message to show the status of Task
      ResultWrapper result = successResult(lastId,'Data before Id(' + lastId +' ) has
been Updated.');
      result.outputJSON= JSON.serialize(inputMap);
```

```
        return result;
    }



}
```