RPA Monitoring Tool

PowerApp Documentation

18/10/2023

Author: Manuel Portero Leiva

Introduction

This document has the purpose to explain the different parts of the RPA Monitoring Tool, its code and functionalities, for understanding and replication purposes. The different parts of the architecture solution are show below.

Picture 1: RPA Monitoring Tool Main Layout

Architecture

The composition of the architecture starts in the PowerApp. The different parts of the app triggers different Http request the UiPath Ochestrator API via PowerAutomate and the requested information is stored in different Sharepoint lists.

The RPA app functionalities are :

- KPI Screen : A screen showing the different KPI of our UiPath RPA environment.

- Scheduled Jobs Screen : A screen showing the processes that are goin to run today in the next ours

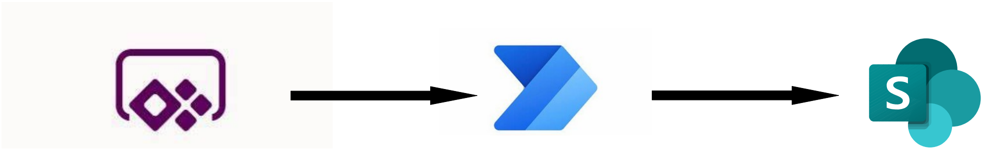
- Error LogsScreen : A screen showing all today’s the errors categorized by robots.

- TroubleShooting Screen : A screen displaying the different errors troubleshooting categorized by Robot Message and Error Type.

- Communication Screen : A screen used to comunicate via email with the RPA process owners.

- Action Screen : A screen set up for perform actions in the RPA environment (Restart machines, set new azure devops tasks etc,...)

A full diagram of the solution is shown below.



Picture 2: RPA Monitoring Tool Architecture

PowerApp

KPI Screen

The main screen is composed by the main landscape picture and the navigation buttons to the others screens.

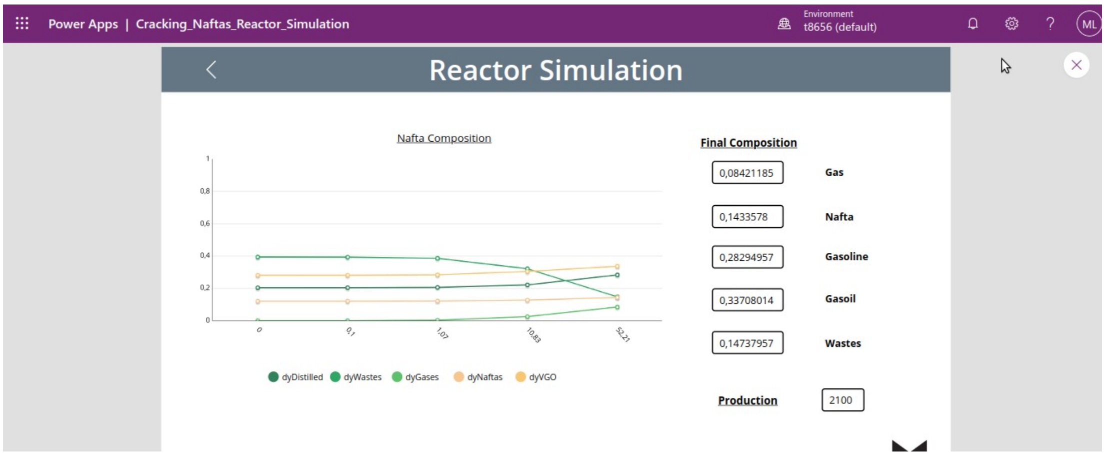
Picture 4: KPI Screen Layout RPA Monitoring Tool

The KPI of this screens are the number of today’s jobs, number of machines in the system, number of failed processes, number of robots of the system and the number of licenses. A part of that this screen has a heat map where we can truck easily which process of the system is failing.

The code for calculating each KPI and for the heat map is shown below :

Details Screen

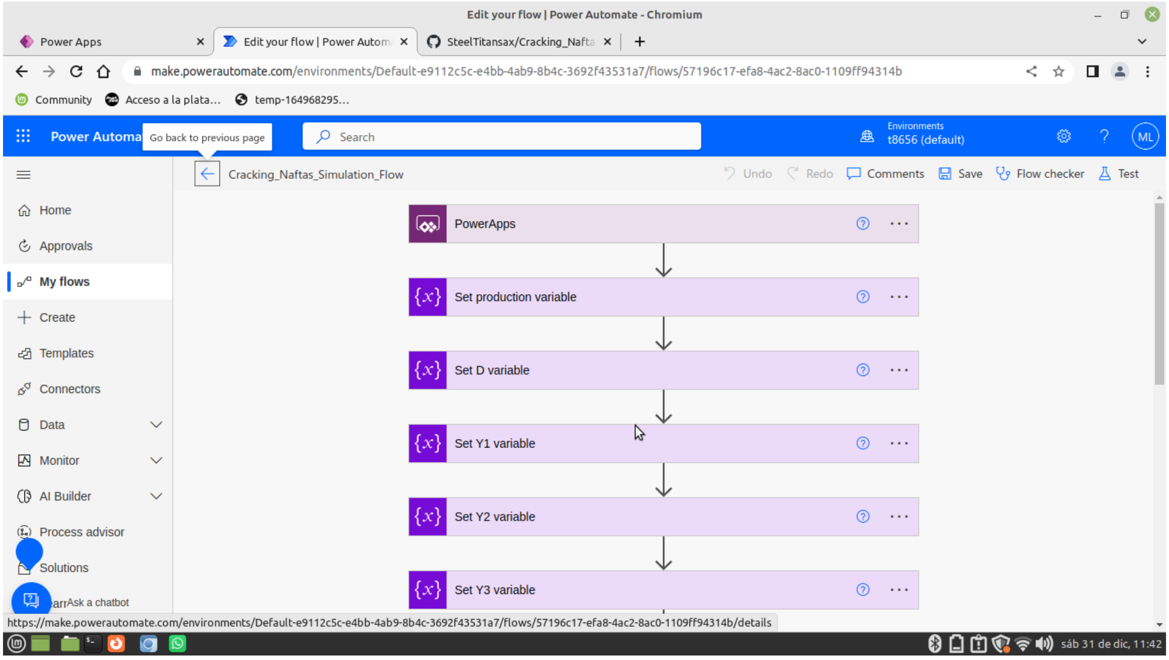
The Details Screen is composed by the different Reactor parameter’s and a reactor design diagram.



Picture 5: Details Screen

PowerAutomate Flow

The PowerAutomate flow receive the paremeters from the PowerApp clear, the Sharepoint list, call the Azure function with an Http Request action and finally fill the Sharepoint list with the new design parameters.



Picture 6: Setting Screen

Azure Function

The Reactor Design azure function will receive the parameters from the PowerAutomate flow and will calculate the Volume and other design parameters of the choosen reactor. The ecuations design code for each reactor type are shown below

Cracking Nafta's Reactor:

```
#Libraries
import logging
import azure.functions as func
import json
import numpy as np
from scipy.integrate import solve_ivp


def main(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Reactor designing Begins...')

    try:
        req_body = req.get_json()
    except ValueError:
        pass
    else:
     production = req_body.get('production') # petrol barrels per day
        production = float(production)
```

```
V0 = production * 0.006624470622 # m3/h
density = 0.715 #g/cm3

massic_flow_0 = V0  * density * 100**3 #g/h
D = req_body.get('D') #m

#      res , vgo , des , naf , gas
y1 = req_body.get('y1') # initial non-processed petrol composition
y2 = req_body.get('y2') # initial non-processed petrol composition
y3 = req_body.get('y3') # initial non-processed petrol composition
y4 = req_body.get('y4') # initial non-processed petrol composition
y5 = req_body.get('y5') # initial non-processed petrol composition

# Parsing input data

D = float(D)
y1 = float(y1)
y2 = float(y2)
y3 = float(y3)
y4 = float(y4)
y5 = float(y5)


y0 = [y1,y2,y3,y4,y5] # initial non-processed petrol composition


k1 = 0.147
k2 = 0.022
k3 = 0.020
k4 = 0.098
k5 = 0.057
k6 = 0.007
k7 = 0
k8 = 0.003
k9 = 0
k10 = 0

V_init = 0
V_final = 500

def reactionSystem(V,y):
    y1 = y[0]
    y2 = y[1]
    y3 = y[2]
    y4 = y[3]

    dyWastes = (-k1*y1-k2*y1-k3*y1-k4*y1)/V0
    dyVGO = (k1*y1-k5*y2-k6*y2-k7*y2)/V0
    dyDistilled = (k2*y1+k5*y2-k8*y3-k9*y3)/V0
    dyNaftas = (k3*y1+k6*y2-k8*y3-k10*y4)/V0
    dyGases = (k4*y1+k7*y2+k9*y3+k10*y4)/V0
```

```
        return np.array([dyWastes, dyVGO,dyDistilled,dyNaftas,dyGases])


    sol = solve_ivp(reactionSystem, (V_init,V_final),y0)

    # Consolidating outputs:
    dyWastes = sol.y[0]
    dyVGO = sol.y[1]
    dyDistilled = sol.y[2]
    dyNaftas = sol.y[3]
    dyGases = sol.y[4]
    t = sol.t

    sol_json = []

    for item in range(5):

        sol_details = {
                "dyWastes": dyWastes[item],
                "dyVGO": dyVGO[item],
                "dyDistilled": dyDistilled[item],
                "dyNaftas": dyNaftas[item],
                "dyGases": dyGases[item],
                "t": t[item]
            }

        sol_json.append(sol_details)

    logging.info(sol_json)

    return json.dumps(sol_json)

return func.HttpResponse("Reactor design succesfully...",status_code=200)
```