CSTR Series Reactor Simulation

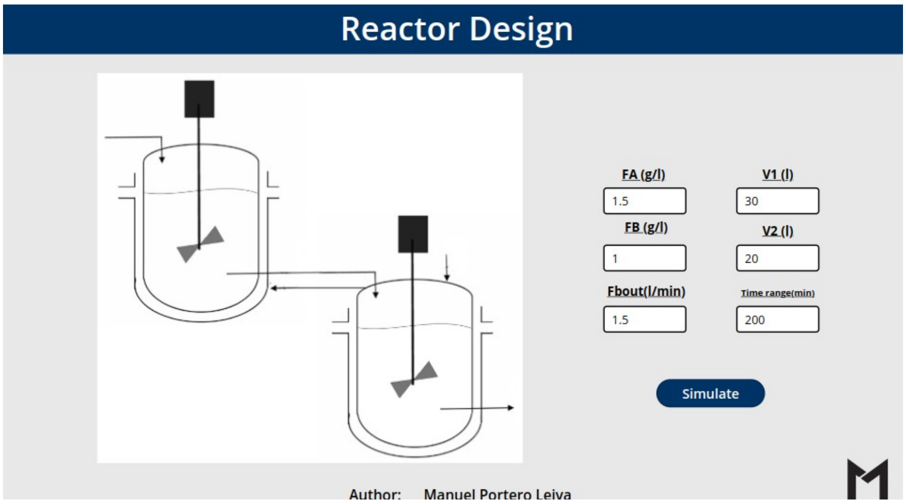
PowerApp Documentation

31/12/2022

Author: Manuel Portero Leiva

Introduction

This document has the purpose to explain the different parts of the Reactor Designing Function App, its code and functionalities, for understanding and replication purposes. The different parts of the architecture solution are show below.

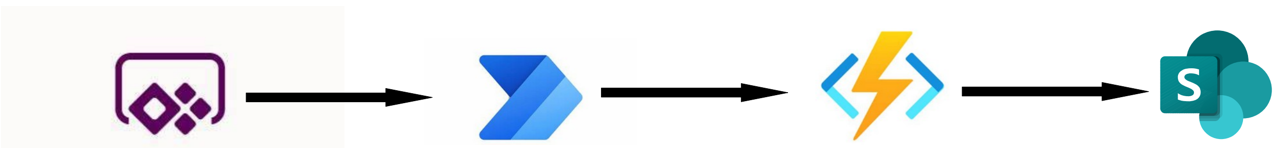


Picture 1: CSTR Series Reactor Simulation Layout

Architecture

The composition of the architecture starts in the PowerApp. Once the Reactor design is choosen and the calculate button is pressed, a Function app is triggered via powerAutomate and a Sharepoint list is filled with the reactor's design data.

A full diagram of the solution is shown below.

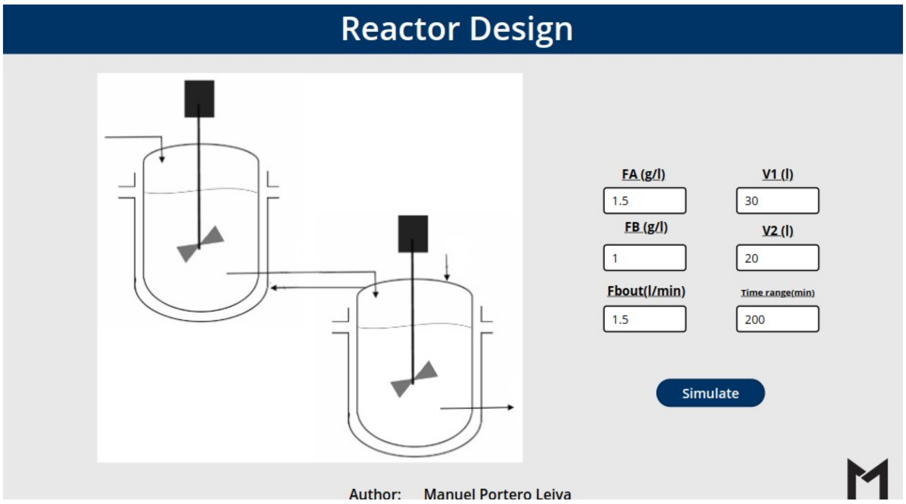


Picture 2: CSTR Series Reactor Simulation Architecture

PowerApp

Main Screen

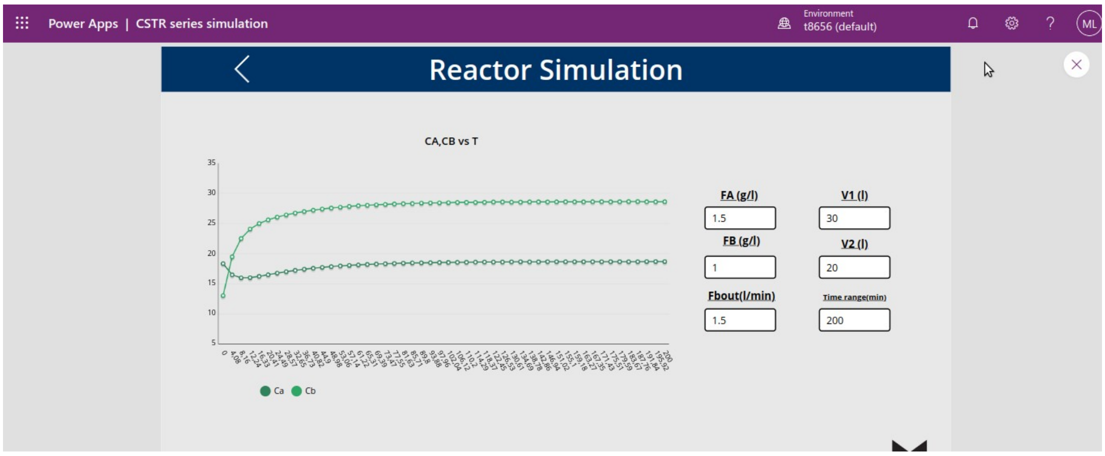
The main screen is composed by the main landscape picture and the navigation buttons to the others screens.



Picture 4: CSTR Series Reactor Simulation main Screen

Details Screen

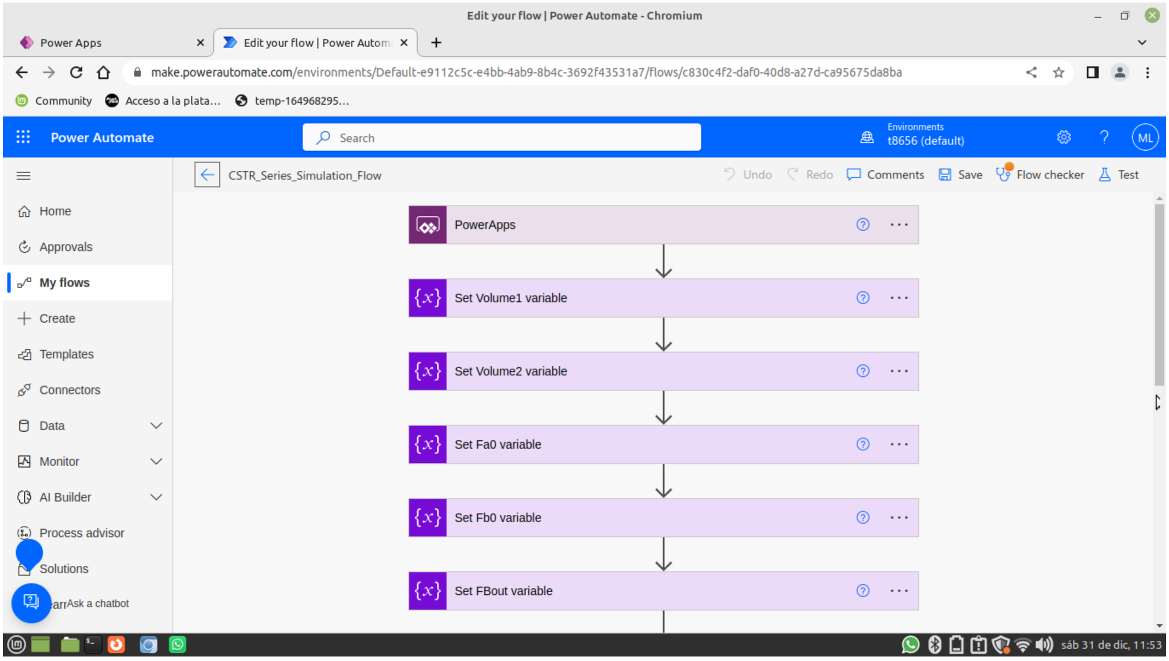
The Details Screen is composed by the different Reactor parameter's and a reactor design diagram.



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: PowerAutomate CSTR_Series_Nafta_Reactor_Simulation

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

CSTR Series 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:

    volumen_A = req_body.get('volumen_A') #L
    volumen_B = req_body.get('volumen_B') #L
    f_A = req_body.get('f_A') #L/min
    f_B = req_body.get('f_B') #L/min
    f_Bout = req_body.get('f_Bout') #L/min
    trange = req_body.get('trange') #min


    #Parsing variables

    volumen_A = float(volumen_A) #L
    volumen_B = float(volumen_B) #L
    f_A = float(f_A) #L/min
    f_B = float(f_B) #L/min
    f_Bout = float(f_Bout)
    trange = float(trange) #min

    #Rest variables declaration

    concentracion_A0 = 550/volumen_A #g/l
    concentracion_B0 = 260/volumen_B #g/l

    print(f'A: {concentracion_A0} g/L')
    print(f'B: {concentracion_B0} g/L')

    c_A = 10 #g/L
    c_B = 30 #g/L
    f_AB = 3 #L/min
    f_BA = 1.5 #L/min

    print(f'Balance TA: {f_A + f_BA - f_AB} L/min')
    print(f'Balance TB: {f_B + f_AB - f_Bout} L/min')

    def dSdt(t,S):
        S_A = S[0]
        S_B = S[1]

        dSadt = ( f_A * c_A - f_AB * S_A + f_BA * S_B )/volumen_A
        dSbdt = ( f_B * c_B + f_AB * S_A - f_BA * S_B - f_Bout * S_B )/volumen_B

        return np.array([dSadt, dSbdt])


    S0 = (concentracion_A0, concentracion_B0)
```

```
        t_span = (0,trange) #min
        t_eval = np.linspace(t_span[0],t_span[1])

        sol = solve_ivp(dSdt, t_span, S0, t_eval = t_eval)


        # Consolidating outputs:
        Ca = sol.y[0]
        Cb = sol.y[1]
        t = sol.t

        sol_json = []

        for item in range(50):

            sol_details = {
                        "Ca": Ca[item],
                        "Cb": Cb[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)
```