# Documentation technique – Projet agent Python

## Introduction

Cette documentation a pour but de vous présenter la création des différents composants de l’agent Python. Cet agent a été développé pour répondre aux critères suivants :

* Récolter des données d’un ou plusieurs ordinateurs,
* Stocker ces données dans un serveur,
* Et les afficher en temps réel, via un navigateur web.

## Diagramme des liaisons







## 

## 

## 



## Le client

Le client permet de faire récupérer les informations de la machine hôte, et de les envoyer vers le serveur API, en utilisant un format JSON pour le transfert et le formatage des données.

* Ici, on récupère l’IP de la machine hôte ainsi que le contenu du fichier de configuration.

import requests, json, psutil, platform,time,os,sys,socket

systeme = platform.system()

IP=socket.gethostbyname(socket.gethostname())

#recuperation de l'url dans le fichier init.conf

try:

if systeme == 'Windows' :

with open('D:\python\init.conf','r') as fichier:

contenu = fichier.read()

contenu = json.loads(contenu)

url=contenu["url"]

elif systeme == 'Linux' :

with open('/home/rt/test/init.conf'):

contenu = fichier.read()

contenu = json.loads(contenu)

url=contenu["url"]

except:

print("fichier init.conf n'existe pas ce referé a l'administrateur

* On vérifie l’existence du fichier service.conf en fonction de l’OS utilisé. Selon l’OS le chemin sera différent. S’il n’existe pas on le créé. Le fichier contient une liste de service à vérifier.

if systeme == 'Windows' :

if os.path.isfile('D:\python\service.conf'):

with open('D:\python\service.conf','r')as f:

service=f.read()

service=json.loads(service)

else :

try :

nomaip = platform.node()+"@"+IP

Jinit={"nom": nomaip,"os": systeme}

reqinit=json.dumps(Jinit)

urlinit= url+"/init"

r = requests.post(urlinit,reqinit)

contenuservice = r.json()

with open('D:\python\service.conf','w') as f:

f.write(json.dumps(contenuservice))

except:

print(sys.exc\_info()[0])

#sys.exit()

#ae

pass

elif systeme == 'Linux' :

if os.path.isfile('/home/rt/test/service.conf'):

with open('/home/rt/test/service.conf','r')as f:

service=f.read()

service=json.loads(service)

else :

try :

Jinit={"nom": platform.node()+"@"+IP,"os": systeme}

reqinit=json.dumps(Jinit)

r = requests.post(url+"/init",reqinit)

with open('/home/rt/test/service.conf','w') as f:

f.write(r.json())

except :

print('serveur injoignable')

sys.exit()

else :

print('system non gérée')

* On instancie une boucle for qui permet de retrouver les services dont on a récupérer les noms ci dessus.

#recherche d'un service

a = [p.name() for p in psutil.process\_iter(attrs=['name'])]

if 'cmd.exe' in a :

print ("gg")

else:

print(a)

* Le try ci-dessous va récupérer les informations concernant les processus, informations qu’elle stockera dans un JSON pour les transmettre.

try:

while(1):

#Récupération de l'hote

nomHost = platform.node()+IP

#Récupération de noyau

version = platform.release()

#Récupération du type cpu

cpu = platform.processor()

#Récupération de la fréquence cpu

cpuFre = psutil.cpu\_freq().current

cpuMax = psutil.cpu\_freq().max

if systeme == 'Windows' :

uptime = "pas de valeur"

elif systeme == 'Linux' :

# calculate the uptime

uptime\_file = open('/proc/uptime')

uptime = uptime\_file.readline().split()[0]

uptime\_file.close()

uptime = float(uptime)

(uptime,secs) = (int(uptime / 60), uptime % 60)

(uptime,mins) = divmod(uptime,60)

(days,hours) = divmod(uptime,24)

uptime = 'up %d jour%s, %d:%02d' % (days, days != 1 and 's' or '', hours, mins)

tabDisk = []

listdisk = psutil.disk\_partitions()

for disk in listdisk:

if (disk.fstype != 'squashfs'):

detaildisk = psutil.disk\_usage(disk.mountpoint)

tabDisk.append({'fileSystem':disk.device,'size':detaildisk.total,'used':detaildisk.used,'available':detaildisk.free,'pourcentage':detaildisk.percent,'mounted':disk.mountpoint})

with open('/home/rt/test/test.txt','a')as f:

service=f.read()

service=json.loads(service)

#Mémoire utilisé

memoireused = psutil.virtual\_memory().used

#Mémoire free

memoirefree = psutil.virtual\_memory().free

#Mémoire buffers

memoirebuffers = psutil.virtual\_memory().buffers

#Mémoire cached

memoirecached = psutil.virtual\_memory().cached

service["id"]=14603

service["os"]= systeme

service["nomhost"]= nomHost

service["noyau"]= version

service["cputype"]= cpu

service["cpufrequence"]= cpuFre

service["uptime"]=uptime

service["metrique"]=tabDisk

service["moccupe"]=memoireused

service["mlibre"]=memoirefree

service["mbuffer"]=memoirebuffers

service["mcached"]=memoirecached

service["total"]=memoireused+memoirefree+memoirebuffers+memoirecached

service["cpufrequencemax"]=cpuMax

print(service)

* Ici, on transforme en JSON les données, et on les envoie toutes les 10 secondes.

donnees = json.dumps(service)

try:

r = requests.post(url, data=donnees)

except:

pass

time.sleep(10)

except IOError:

print(IOError)

## Le serveur

Le serveur consiste en la conception d’une API, qui va permettre aux clients d’envoyer les données de leur pc, de les stocker dans une base de données et de les traiter. Toutes les données traitées seront en format JSON et la base de données est sous MySQL.

## Description du code

* Ouverture de la connexion avec la base de données MySQL.

conn = mysql.connector.connect(host="localhost",user="tests4",password="4NqGjgZ", database="tests4")

cursor = conn.cursor()

* Création d’un Logger pour conserver les messages d’erreur côté client, les messages sont stockés dans un fichier client.log qui se trouve dans le répertoire logs.

logger = logging.getLogger()

logger.setLevel(logging.DEBUG)

# création d'un formateur qui va ajouter le temps, le niveau

# de chaque message quand on écrira un message dans le log

formatter = logging.Formatter('%(asctime)s :: %(levelname)s :: %(message)s')

# création d'un handler qui va rediriger une écriture du log vers

# un fichier en mode 'append', avec 1 backup et une taille max de 1Mo

file\_handler = RotatingFileHandler('./logs/client.log', 'a', 1000000, 1)

# on lui met le niveau sur DEBUG, on lui dit qu'il doit utiliser le formateur

# créé précédement et on ajoute ce handler au logger

file\_handler.setLevel(logging.DEBUG)

file\_handler.setFormatter(formatter)

logger.addHandler(file\_handler)

* Ouverture du fichier config.conf, qui va contenir l’adresse IP du serveur.

with open("config.conf", "r") as fichier:

    ip = fichier.read()

* On instancie la route qui permet de récupérer les données envoyées par le/les clients

@app.route('/api', methods=['POST'])

def home():

* Premier try/except pour repérer les erreurs présentes côté serveur, où l‘on récupère le premier échange en format JSON

try:

        y = json.loads(x)

* On teste si le client est déjà enregistré dans la base de données, si ce n’est pas le cas, on lui retourne un message d’erreur lui indiquant que le message n’existe pas.

        sql = "SELECT idHote from hote where idHote=%s"

        val = (y['id'],)

        cursor.execute(sql,val)

        rows = cursor.fetchall()

 if not rows:

            logger.error("L'hote n'existe pas")

            return json.dumps({'error':"ERREUR : L'hote n'existe pas"})

        else:

* On réalise ici les insertions de données dans la BDD, voir code complet dans l’annexe.

sql = """UPDATE hote SET nom=%s, OS=%s, uptime=%s, noyaux=%s WHERE idHote=%s"""

            val = (y['nomhost'],y['os'],y['uptime'],y['noyau'],y['id'])

            cursor.execute(sql, val)

            conn.commit()

            sql = """INSERT INTO cpu(idHote, frequence,frequenceMax, type) VALUES (%s, %s, %s, %s)"""

            val = (y['id'],y['cpufrequence'],y['cpufrequencemax'],y['cputype'])

* Création du journal de log des erreurs côté serveur, par exemple : connexion non réussie avec la base de données, insertion des données non abouties, …

    except (RuntimeError, TypeError, NameError, ValueError) as error:

        logger = logging.getLogger()

        logger.setLevel(logging.DEBUG)

        # création d'un formateur qui va ajouter le temps, le niveau

        # de chaque message quand on écrira un message dans le log

        formatter = logging.Formatter('%(asctime)s :: %(levelname)s :: %(message)s')

        # création d'un handler qui va rediriger une écriture du log vers

        # un fichier en mode 'append', avec 1 backup et une taille max de 1Mo

        file\_handler = RotatingFileHandler('./serveur.log', 'a', 1000000, 1)

        # on lui met le niveau sur DEBUG, on lui dit qu'il doit utiliser le formateur

        # créé précédement et on ajoute ce handler au logger

        file\_handler.setLevel(logging.DEBUG)

        file\_handler.setFormatter(formatter)

        logger.addHandler(file\_handler)

        # création d'un second handler qui va rediriger chaque écriture de log

        # sur la console

        stream\_handler = logging.StreamHandler()

        stream\_handler.setLevel(logging.DEBUG)

        logger.addHandler(stream\_handler)

        # Après 3 heures, on peut enfin logguer

        # Il est temps de spammer votre code avec des logs partout :

        logger.error(error)

* On instancie la route qui va permettre de retourner une liste de services au client en lui attribuant un numéro d’ID s’il n’en possède pas déjà un.

@app.route('/api/init', methods=['POST'])

def init():

* Récupération du nom de la machine. Si elle n’existe pas dans la BDD, on ajoute son nom et son OS. On envoie ensuite une liste de services en fonction de l’OS de la machine. On ajoute les erreurs dans le fichier log “client.log”

x = request.data

try:

y = json.loads(x)

sql = """SELECT nom from hote where nom=%s"""

val = (y['nom'],)

cursor.execute(sql,val)

rows = cursor.fetchall()

try:

if not rows:

sql = """INSERT INTO hote(nom, OS, uptime, noyaux) VALUES (%s,%s,"","")"""

val = (y['nom'],y['os'])

cursor.execute(sql,val)

conn.commit()

id = cursor.lastrowid

sql = """SELECT nom from hote where nom=%s"""

val = (y['nom'],)

cursor.execute(sql,val)

rows = cursor.fetchall()

else:

sql = """SELECT idHote from hote where nom=%s"""

val = (y['nom'],)

cursor.execute(sql,val)

id = cursor.fetchall()

for i in id:

idStr = i

if y['os'] == "Linux":

return json.dumps({'id':idStr[0], 'services':[{'name':'bluetooth.service'},{'name':'cron.service'},{'name':'alsa-store.service'}]})

else:

return json.dumps({'id':idStr[0], 'services':[{'name':'cmd.exe'},{'name':'mpssvc'},{'name':'WSearch'}]})

except:

loggerClient.error("Mauvaises données")

except:

loggerClient.error("Aucune données envoyées")

* Lance l’API avec l’IP de la machine hôte

app.run(host = ip)

## L’interface web

Cette interface, créer en langage html et javascript, va présenter en temps réel, de manière ordonnée et visuelle les données qui ont été récupérées sur la base de données. On peut récupérer les données en fonction d’un hôte spécifique.

## Description du code

Méthodes appelées par le code html et les javascript

## 1 / **Code** html

* On ouvre une connexion avec la base de données MySQL et on instancie une variable qui va porter les requêtes à réaliser via la base de données

connection = pymysql.connect(host='192.168.3.26',

                             port=3306,

                             user='tests4',

                             password='4NqGjgkZ',

                             db='tests4',

                             charset='utf8mb4',

                             cursorclass=pymysql.cursors.DictCursor)

app = Flask(\_\_name\_\_)

def select(sql):

  cursor = connection.cursor()

  cursor.execute(sql)

  temp = cursor.fetchall()

  return temp

* Méthode récupérant tous les ID hôte, afin de créer la liste des hôtes présents dans la combobox sur la page du navigateur web

@app.route('/',methods=['POST', 'GET'])

def getAll():

  # Read a single record

  sql = "SELECT hote.idHote,hote.nom from hote"

  data = select(sql)

  return render\_template("index.html", taille = len(data), listeMachines=data)

* Méthode récupérant les métriques d’une machine sélectionnée, soient les informations techniques de l’hôte telles que : la capacité des disques dur, le pourcentage d’utilisation du processeur, le pourcentage d’utilisation de la mémoire vive.

@app.route("/machine/<int:idMachine>/getInfos")

def getInfos(idMachine):

  metrics = {}

  sql2 = "SELECT \* FROM disque where disque.idHote="+(str(idMachine)+" ORDER BY idDisque DESC LIMIT 10")

  metrics['memoryUsage']=select(sql2)

  sql3 = "SELECT \* FROM typepartition inner join disque on disque.idDisque=typepartition.idDisque where disque.idHote="+(str(idMachine)+" ORDER BY typepartition.idDisque DESC LIMIT 10")

  metrics['diskUsage']=select(sql3)

  sql4 = "SELECT \* FROM cpu where cpu.idHote="+(str(idMachine))

  metrics['cpuUsage']=select(sql4)

  return jsonify(metrics)

* Méthode permettant de récupérer les informations d’une machine, comme son système d’exploitation, son nom, la durée depuis laquelle est est allumée et son noyaux.

@app.route("/machine/<int:idMachine>" , methods=['GET', 'POST'])

def getMachine(idMachine):

  sql = "SELECT \* FROM hote where hote.idHote="+(str(idMachine))

  hostInfo=select(sql)

  return render\_template("select.html", host=hostInfo)# just to see what select is

if \_\_name\_\_ == "\_\_main\_\_":

    app.run(host='0.0.0.0')

## 2 / Code JavaScript

* Code présent dans le fichier index.js, il représente un événement qui se lancera lorsque le fichier html sera complètement chargé.

document.addEventListener('DOMContentLoaded', function() {

var elems = document.querySelectorAll('select');

var instances = M.FormSelect.init(elems);

});

function goToMachine(){

var machine = document.getElementById("selectedMachine");

var machineID = machine.options[machine.selectedIndex].value;

var base\_url = window.location.origin;

window.location.replace(base\_url+"/machine/"+machineID);

}

* Code présent dans le fichier machine.js, elle instancie toute la mise en forme des graphiques qui vont apparaître sur la page html.

var pageURL = window.location.href;

var idHote = pageURL.substr(pageURL.lastIndexOf('/') + 1);

var dataMachine;

var elem = document.querySelector('.collapsible.expandable');

var instance = M.Collapsible.init(elem, {

accordion: false

});

var autoReload = setInterval(refreshDatas,5000);

// Rafraichissement de la récupération des infos concernant une machine en fonciton de son ID

function refreshDatas(){

fetch('/machine/'+idHote+'/getInfos')

.then(response => response.json())

.then(data => {

this.dataMachine = data;

})

.catch(error => console.error(error));

drawChartLine1();

drawChart();

drawTable();

}

fetch('/machine/'+idHote+'/getInfos')

.then(response => response.json())

.then(data => {

this.dataMachine = data;

})

.catch(error => console.error(error))

google.charts.load('current', {'packages':['corechart']});

google.charts.setOnLoadCallback(drawChartLine1);

function drawChartLine1() {

var dataTable = new google.visualization.DataTable();

dataTable.addColumn('number', 'Total');

dataTable.addColumn('number', 'Free');

dataTable.addColumn('number', 'Used');

dataTable.addColumn('number', 'Buffers');

dataTable.addColumn('number', 'Cached');

for(var i = 0; i < dataMachine.memoryUsage.length; i++){

dataTable.addRow([dataMachine.memoryUsage[i].memoireTotal,

dataMachine.memoryUsage[i].memoireLibre,

dataMachine.memoryUsage[i].memoireOccupe,

dataMachine.memoryUsage[i].buffer,

dataMachine.memoryUsage[i].cache]);

}

var options\_stacked = {

isStacked: 'relative',

height: 300,

legend: {maxLines: 3},

vAxis: {

minValue: 0,

ticks: [0,.15, .3, .45, .6, .75,.9, 1]

}

};

var chart = new google.visualization.AreaChart(document.getElementById('chart\_div'));

chart.draw(dataTable, options\_stacked);

}

google.charts.setOnLoadCallback(drawChart);

function drawChart() {

var cpuUsed = 0;

var freeCPU = 0;

var dataTable = new google.visualization.DataTable();

dataTable.addColumn('string', 'CPU');

dataTable.addColumn('number', 'Used');

for(var index = 0; index < dataMachine.cpuUsage.length; index++) {

console.log(freeCPU,cpuUsed);

freeCPU = freeCPU + dataMachine.cpuUsage[index].frequenceMax;

cpuUsed = cpuUsed + dataMachine.cpuUsage[index].frequence;

}

freeCPU = freeCPU / dataMachine.cpuUsage.length;

cpuUsed = cpuUsed / dataMachine.cpuUsage.length;

dataTable.addRow(["Free",freeCPU]);

dataTable.addRow(["Used",cpuUsed])

var options = {

title: 'CPU USAGE',

pieHole: 0.4,

};

var chart = new google.visualization.PieChart(document.getElementById('donutchart'));

chart.draw(dataTable, options);

}

google.charts.load('current', {'packages':['table']});

google.charts.setOnLoadCallback(drawTable);

function drawTable() {

var dataTable = new google.visualization.DataTable();

dataTable.addColumn('number', 'Available');

dataTable.addColumn('string', 'File system');

dataTable.addColumn('string', 'Mounted');

dataTable.addColumn('string', 'Pourcentage');

dataTable.addColumn('number', 'Size');

dataTable.addColumn('number', 'Used');

for(var i = 0; i < dataMachine.diskUsage.length; i++){

dataTable.addRow([dataMachine.diskUsage[i].available,

dataMachine.diskUsage[i].fileSystem,

dataMachine.diskUsage[i].mounted,

dataMachine.diskUsage[i].pourcentage,

dataMachine.diskUsage[i].size,

dataMachine.diskUsage[i].used]);

}

var table = new google.visualization.Table(document.getElementById('table\_div'));

table.draw(dataTable, {width: '100%', height: '100%'});

}