https://docs.google.com/document/d/1hz2FKSW2IgzfwXgS1Ic3LAJzKInGgU-w8kaXeYqkUuc/edit

*** file structure should be set different for running the platform.**

Overall steps

1. Type registration (definition present or not)
2. Send app repository(application -algos)
3. sensorInstance
4. Deploy config

***sensorType and sensorInstance can be registered any time irrespective of the order.**

Main features of the platform---

1. Providing API for development of algorithms.
2. Connecting and getting data from sensors.
3. Distributed deployment and support of the application.
4. Fault tolerance to the applications.
5. Scheduling on demand tasks in the application.
6. Providing dashboards to the user as well as admin of the application.
7. Handling sensor's alert.--Controlling sensors
8. Scalability in terms of reach (computation),
    a. To span large area
    b. Providing new server to increase load
9. TODO -- scaling and security/authentication

Platform manager - config files to their resp modules

1. HTML form of index, where when we upload a file(any config file), it is sent to a new URL(action=URL -HTML), NOw using flask (particular function is being called based on its URL) in platform manager.py.
2. Describing function- Sensor type registration-
    a. Here similar to http data transfer we use flask request object to get the file and save it .
    b. Now the function for validation are present in validator.py files which are imported to call the sensor_type_validation() function.
    c. If the file is validated successfully, the file is loaded into json and the json data is sent onto the topic-pm_to_sensor_type_reg to sensor_registration module.
    d. Based on success or failure, message is printed on the browser.
3. 1st upload the Sensor types not registered in the platform.
4. Upload the app.zip file
    a. First enter an application id. -Used to distinguish the application in the platform(saved in the database as an identifier for the application)
    b. Save the application id, request.files['file'].save(filename) will save file in root folder(where program is running) directly.

c. Create a folder and save the extracted files (stored in root folder) in folder-/datadrive/apps/
d. Validate the file using validator.py functions.
e. If validated successfully , read the zip file in binary format and save the details of the application in the mongodb collection(AppRepo.ID_sourceFolder)-collection.insert_one({"Application id ": appName,"folder":encoded_binary})

5. Upload the sensor_instance.json.----similar to sensor type registration(topic-pm_to_sensor_ins_reg).

ROLES -

Application Developer: (w.r.t application)

1. Write the algorithms by getting the sensor data using the api provided by the platform.
2. Write appConfig.json and submit a proper app.zip file to platform.

Application Configurator: (w.r.t application)

- Register the sensor Type and the sensor instances. Using sensorTypeRegistration.json and sensorInstance.json
- Provide a config file (deployConfig.json) to the platform containing when to deploy the algorithms and on which sensors.

End User: (w.r.t application)

- Interacts with the application UI which is provided by application developers.
- Will receive notification from the sensor manager.
- May trigger a particular action(control) on a sensor.

Platform Admin (w.r.t platform)

Setting up platform and monitoring it while running. Maintaining the platform is his role.

INDEX (platform-manager/templates/index.html)

Gives option to upload multiple files(

`uploadAppZip,uploadSensorType,uploadSensorInstance,uploadDeployConfig)`

Dashboards for each user -

1. Application developer will get option to upload application zip file - uploadAppZip
2. Configurator - will get the option to upload deploy, sensor type, sensor instance configurations files, option to view its application status (running fine?), application specific container id can be displayed, view sensor_types on platform,view sensor_instance it has registered
3. End user (w.r.t application) - will see application UI eg. bus fare
4. Platform admin - dashboards for platform components (components + return types status)

- 2 views - service container logs + application container logs

Terms -
enctype attribute specifies how the form-data should be encoded when submitting it to the server.
Respective pages will lead to different function call (defined in platformManager.py) using rest API(flask)

Kafka Topic Names:

Producer	Topic Name	Consumer	Purpose
platform_manager	pm_to_sensor_type_reg	sensor_registration	Send sensorTypeRegistration.json contents
platform_manager	pm_to_sensor_ins_reg	sensor_registration	Send sensorInstance.json contents
platform_manager	pm_to_sensor_binder	sensor_binder	Send deployConfig.json contents
sensor_binder	sensor_binder_to_scheduler	scheduler	Send deployConfig.json contents
scheduler	scheduler_to_deployer	deployer	Send deployConfig.json contents

BOOTSTRAP -
Various roles of bootstrap -

1. Set up/Start platform
2. Initialize platform components
3. Set up all machines
4. Setting up kafka and docker environment

CONFIG FILES
appConfig:
Application developer uploads it for its own application.

```
"application_name": "Bus_management",
  "application_id": 2,
  "developer_id": 869,
  "environment":{
    "os": "Linux",
    "modules": [
      "pandas",
      "numpy"
    ]
  },
```

```
"algorithm_list":[
    {
      "algorithm_name":"fareCalculator",
      "script":{
        "name":"fareCalculator.py"
      },
      "input_sensors":[
        {
          "sensor_type":"gps-sensor"
        },
        {
          "sensor_type":"biometric-sensor"
        },
        {
          "sensor_type":"gps-sensor"
        }
      ]
    }
```

Platform_lib.py

```
import requests
url = "http://0.0.0.0:5001"
def getSensorData(id,index):
        res = requests.post(url + '/getSensorTopic',json={"id":id,"index":index})
        print(res.json())
        return res.json()["data"]

def setSensorData(id,index,val=None):
        res = requests.post(url + '/setSensorTopic',json={"id":id,"index":index,"val":val})
        return res.json()["data"]
```

Here input sensors are the different instances of sensors required by the algorithm. So we give the type of each instance required by this algo.
The application_id and folder:(binary format of app.zip) is inserted in AppRepo.ID_sourceFolder.

DeployConfig : structure - pending
Contains deployment info fo a given algorithm of an application(applications can be different)
In our platform can contain deployment details of different algos from different applications (sensor details and scheduling info)
DeployConfig is uploaded to the platform by application_configurator on platform_manager. It is verified (valid = validator.validate_deployConfig('./deployConfig.json') If valid it is sent on topic pm_to_sensor_binder else return error msg.

The deployconfig file is sent to sensor binder is used by sensor_binder module to bind the sensor instances and save the mapping in sensor map database collection. When the sensor is mapped an instance _id is generated, this is added to deploy_config data details and send to scheduler via topic sensor_binder_to_scheduler. Here scheduler inserts the deploy_config details in the **scheduler_logs.deploy_configs collection with instance _id=instance_id**. This details saved in the collection can be used to identify the pending jobs.

Also delete the details if the schedule info is of 1 time or stop request type ,also updated the deployconfig details for stop type.
Then scheduler sends the deploy_config data to deployer on topic- scheduler_to_deployer. Now deployer creates docker files and send the deploy_config details to the slc through topic-deployer_to_slc.

Sensor info - gives sensor instances info to an algo along with filters, based on filters we can bind actual instances to our algorithm instance.
Presently our platform contains static instances for an algo, we can upgrade it to add multiple sensor instances(add a field eg count where we can give count of sensor instances or make it as variable(like in collision detection algo))


```
{
  "noOfAlgo":4,
  "1":{
    "application_name":"Bus_management",
    "script_name" : ["fareCalculator.py"],
    "algorithm_name":"fareCalculator",
    "sensor_info":[
      {
        "sensor_type":"gps-sensor",
        "filter_sensors":[
          {
            "placeholder":"bus",
            "bus-id":"1"
          }
        ]
      },
      {
        "sensor_type":"biometric-sensor",
        "filter_sensors":[
          {
            "bus-id":"1"
          }
        ]
```

```
        },
        {
          "sensor_type":"gps-sensor",
          "filter_sensors":[
            {
              "placeholder":"admin",
              "admin-id":"1"
            }
          ]
        }
      ],
      "scheduling_info":{
        "request_type": "start",
        "start_time": "",
        "end_time": "",
        "days": "[]",
        "interval": "30",
        "repeat": "no",
        "job_id": "123 exists only if request type is stop"
      },
      "environment":{
        "lang":"python",
        "dependencies": [["requests", ""], ["kafka-python", ""], ["pymongo", ""], ["dnspython", ""]]
      }
    },
    "2":{
.
.
.
.
}
}
```

Here sensor_info is processed by sensor_binder module and scheduling info is processed by scheduler. For each sensor type required by the algorithm we provide filters to bind the senor instance.

sensorTypeRegistration.json-

```
{
  "sensor_type_name":"gps-sensor",
  "company":"samsung",
  "Sensor_data_structure":{ // return data
"placeholder":"string",
     "x":"float",
     "y":"float"
  },
```

```
  "control_functions":{
    "number_of_functions":0,
    "function_details":[]
  }
},
{
  "sensor_type_name":"temp-sensor",
  "company":"lg",
  "sensor_data_structure":{
    "temperature":"int"
  },
  "control_functions":{
    "number_of_functions":1,
    "function_details":[
      {
        "name":"switchOnAC",
        "number_of_parameters":1,
        "params":[
          {
            "time":"int"
          }
        ]
      }
    ]
  }
}
```

Sensor_registory.sensor_type - collection(mongoDB)
sensor_type_list will contain list of different sensor types(which can get registered in the platform) as dictionaries.

Sensor_type_name will define the name for that sensor,
Sensor_data_structure - will give datatypes that will be returned by a sensor as data.(type1:type2:type3)
Control_function- if a sensor can be controlled, control info will be given too(control functions detail). Number of function and what parameter and type each parameters that a function can take.
Whenever sensor_instance will get registered in our platform, first check will be to see it sensor_instance type is registered in a platform or not. If it is not registered, sensor_instance also can't be taken(registered).
Sensor_type_register.py - will receive sensor_type_registration from pm_to_sensor_type_reg(topic) sent by platform manager. All details will be stored in collection Sensor_registory.sensor_type - collection(mongoDB)

1. **_id:**609522b08ddba7ea1cb8b2fa
2. **control_functions:**Object

```
        1. number_of_functions:0
        2. function_details:Array
    3. company:"samsung"
    4. sensor_type_name:"gps-sensor"
    5. sensor_data_structure:Object
        1. y:"float"
        2. x:"float"
        3. placeholder:"string"
```

sensorInstance.json-
Represents individual sensor of a sensor type. Can be multiple instance of same type.

```
{
   "sensor_type":"gps-sensor",
   "ip":"127.23.65.90",
   "port":"8766",
   "no_of_fields":2,
"placeholder":"admin",
   "admin-id":"1"
 },
```

Will represents different filter for a sensor, we can use any combination when we are actually deploying algorithms from these filters to search for required sensor_instance for an algo instance. Also sensor_instance filters can be anything(any key-value pair).

pm_to_sensor_ins_reg - platform_manager will send sensor_instance_registration.json to sensor_instance_registration.py

Sensor_registory.sensor_instance - MongoDB

```
1. _id:60979ad8df41d0614b056a8e
2. ip:"127.23.65.90"
3. no_of_fields:2
4. sensor_type:"gps-sensor"
5. admin-id:"1"
6. placeholder:"admin"
7. port:"8766"
8. topic:"topic_in1620548311.84993"
9. Topic_control:"topic_control_gps-sensor_1620548311.84993"
```

Sensor_instance_registration.py - reads sensor_instance_registration.json and will create two topics for sending sensor data and controlling sensor, stores all this info in collection.

Pm_to_sensor_ins_reg(topic) will read from here. For each instance, we will create two topicstopic_w = "topic_in"+str(t) - sensor produces data here

topic_control = "topic_control"+str(t) - will receive data here

t is the current timestamp that'll get appended to the topic name.

Get file name= x["sensor_type"]+".py".

Dictionary so formed will be stored in db(Sensor_registory.sensor_instance)

Both topics will be created here only.

Enhancement: control topic can be created only when control_functions is more than 1.

Connect_sensor() - will be called with arguments name,topic_w,topic_control. It will start the py file.

os.system('python3 ' + name + ' ' + topic_w+' '+topic_control)

Sensor Manager- If we consider sensor manager platform lib may change. It works as an intermediate between the actual sensor instance and the algorithm instance. SO whenever an algorithm requests for the sensor topic, it calls platform lib api, which indirectly calls the sensor manager. So sensor manager searches the sensor_map collection with the given instance id and gets the topics. Now it creates 2 topics(as per sensor instance has control or not) and corresponding consumer from in_topic of sensor instance and producer for control_topic. So the new topics created will receive or send data from sensor manager. Now the sensor manager returns these 2 topics to the algo instance.
Here platform_lib now hits sensor manager api for the topics instead of senor_map api defined in platform manager.

PLATFORM MANAGER

1. Flask is running at host - host='0.0.0.0', port=5001
    a. app.run(host='0.0.0.0', port=5001,debug=True)
2. Describing function- Sensor type registration-
    a. Here similar to http data transfer we use flask request object to get the file and save it .
    b. Now the function for validation are present in validator.py files which are imported to call the sensor_type_validation() function.
    c. If the file is validated successfully, the file is loaded into json and the json data is sent onto the topic-pm_to_sensor_type_reg to sensor_registration module.
    d. Based on success or failure, message is printed on the browser.
3. 1st upload the Sensor types not registered in the platform.
4. Upload the app.zip file
    a. First enter an application id. -Used to distinguish the application in the platform(saved in the database as an identifier for the application)
    b. Save the application id, request.files['file'].save(filename) will save file in root folder(where program is running) directly.
    c. Create a folder and save the extracted files (stored in root folder) in folder-/datadrive/apps/

    d. Validate the file using validator.py functions.
    e. If validated successfully , read the zip file in binary format and save the details of the application in the mongodb collection(AppRepo.ID_sourceFolder).-
        i. collection.insert_one({"Application id ": appName,"folder":encoded_binary})
5. Upload the sensor_instance.json.----similar to sensor type registration(topic- pm_to_sensor_ins_reg).
6. Upload deploy_config_file for each algorithm
    a. If the file is validated successfully, we count no. of algo instances and for each instance we send the deployment details on the topic- pm_to_sensor_binder. Info is send in a topic algo wise.

All file uploaded!
Platform manager will send deployConfig to sensor_binder.py via topic pm_to_sensor_binder.

Sensor Binder:
Process deployConfig file received from platform manager. Based on sensor filters in sensor info(deployConfig.json) inserts mapping in sensor map (collection - binding_db.sensor_map)
Role :

1. We receive only details of a single algo from platform manager.
2. First we get the sensor info and created a list of same type of sensors.
3. Then created a search dictionary with key sensor type and its value. Further merged the filters of sensor in the same dictionary.
4. Now Counted the no of sensors in sensor_instance collection with the given search dictionary.
5. If we get the required number of sensors , we create a collection with index:object_id of the sensor_iinstance collection. Here index is the position of the sensor in the sensor list.
6. Now insert the collection in binding_db.sensor_map. For each collection inserted , an object id is created(default behaviour in mongoDB). This object id will uniquely represent the given algorithm (for which we are processing the deploying details).
7. Function bindSensor returns this - {"status_code": 200, "status": "OK", "instance_id": instance_id} - processRequest.
8. Now the object_id is considered as instance_id for the algorithm and it uniquely represents it.
9. Here we append the instance_id to the deploy_config data.
10. Now the sensor binder sends the instance_id and deploy_config data to the scheduler via topic- sensor_binder_to_scheduler.

Scheduler-
Receives the deploy_config details from the sensor_binder.[1]
Two threads will be running in parallel from main. 1 for running pending jobs and other for consumption from sensor binder.
Consume_from_sensor_binder - data received from sensor binder will be loaded and sent to handle_schedule_info function to be processed further.
Handle_schedule_info - if request type is start, function start_schedule function will be called with all scheduling details and deployConfig file, one we are back from function, data is set in collection[deploy_config].
If request type is stop, send_to_deploy function will be called

---

[1] Kafka
Docker
mongoDB
Mount file system
Flask
Virtual machine - azure
Basic architecture (distributed and others)
ssh

Start_schedule -this function will make a schedule based on its repetition(repeat) value, if repetition is true then we specify schedule on the basis of days, if day is empty, then we schedule it on start time every day. We are not considering interval value here.
If repetition is false, again will check if days are empty or not and will also schedule it based on interval.
If end_time is not empty (same in every case) - will update end_time in deployConfig in a new dictionary(not in database) and will schedule accordingly with new deploy config(stop_data).
Scheduler will call send_to_deployer at schedule time,
Send_to_deployer will send data(deploy config details) to deploy via topic schedular_to_deployer
And also remove collection from schedule is for one_time

signal.signal(signal.SIGINT, signal_handler) - this will handle ctrl-c or any abruption while running

Deployer-
Consume from the topic the deployConfig data from scheduler, Then based on the request type, call the create_files- creates docker and request file. Then send the deploy_config file to server life cycle. Otherwise send to send_stop_request function. There we send deploy_config file to server life cycle.
Creation of docker file and requirement file is done here.
Constant are written in caps.

SLC- Server Life Cycle-

- Monitoring and starting machines
- Deploy config requests are actually run by building image of docker (running application) on the machines.
- Gets app_monitoring info from a topic and restarts the application.

Config files required

1. Config.json will contain machine (server) info so that when required we can ssh into vm. Machine ip, machine_ name, subscription_id, resource_group_name, username.
2. server_details.json

3 parallel threads -
check()- will check the status of all virtual machines(servers) through Kafka topic, Kafka topics are stored in DB by bootstrap(confirm?). A set(python set) is maintained to keep all topics of servers. An infinite loop is run, if a topic is found in db and not found in set, add topic to set and monitor it(consumer_handler).
consumer_handler() - for each server there will be different threads running, will receive constant updates from server, if no data is received, remove corresponding topic from set(server_status) and close that particular topic, further call restart_machine(server topic) to restart machine.

Restart machine() - get machines info from a config file, and call start_machine() to restart that machine, For each restart call the machine_status_checker.py. This will create a new topic for the machine to generate its status on it. Then store the topic details in machine_info collection of server_lifecycle.

```
content = { "_id" : machine_name,
          "ip" : ip,
          "topic" : topic_name
        }
```

deployer_request() - will run any algorithm requested based on instance_id (). So, if there is any request to run it will send deploy_config_details(received from deployer) to sendRequestService, will store instance_id, app_name, algo_name, request_type from config data received. If a request is of start type, call start_app_instance else call stop_app_instance.
Start_app_instance - will start the required container, it will set algo_path and docker_file_path, and will call load_balancer() to fetch machine to run the current instance. Node(name and id) chosen will be send to a function build_docker_image() to build image if not present. Finally run_container will run the container. After returning from function it will also store the instance-id ,app name and algo name to the collection **instance_id_to_app_algo** and instance id, node name and its id to **instance_id_to_server.**

restart_app_handler()- will run(rerun application if there is any request to run it). **app_monitoring_to_request** is the topic that'll consume requests provided by app_monitoring. Call restart_app() will be called to start application instance.

restart_app() - will get info based on instance_id from collection **instance_id_to_app_algo.** If no info, the container does not exist otherwise we start the application again.
To start application again, call start_app_instance with instance_id, app_name and algo_name.

Load Balancing - Load balancer is a class, can be used to find load on machines(servers) Start_app_instance will call load_balancer. select_machine() of load balancer will be called to find appropriate machine where we can run the current application.

select_machine() - will load server_details.json to find node_name and node_ip. For each server find load(using a formula described below) and machine which has the least load will be selected to run the application. Server details will be returned back to the calling function(start_app_instance)

Formula- load = 0.5*mem_usage + 0.5*cpu_usage
Paramiko
ssh-ssh.exec_command()
Load balancer object (initialization)has -
Commands for getting memory usage and CPU idle time-

```
self.memory_usage_cmd = 'free | head -2 | tail -1 | awk \'{print $2, $3}\'"
```

self.cpu_idle_cmd = 'vmstat 1 2 | tail -1 | awk \'{print $15}\'"
self.ssh = paramiko.SSHClient()
self.ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())

For each server ssh.exec_command(self.memory_usage_cmd) is called, it returns 3 parameters as (need to understand)
_, mem_stdout, mem_stderr = ssh.exec_command(self.memory_usage_cmd)
_, cpu_stdout, cpu_stderr = ssh.exec_command(self.cpu_idle_cmd)

Calculate load based on below formula,
mem_usage_parts = mem_usage[0].split()
mem_usage = int(mem_usage_parts[1][:-1]) / int(mem_usage_parts[0])
cpu_usage = 1 - int(cpu_idle[0][:-1]) / 100
load = 0.5*mem_usage + 0.5*cpu_usage

Service Life Cycle-

1. Get the details of each machine from **config.json--**
   "machines":{
   "52.188.83.232" : {
   "machine_name" : "IAS-Node-1-new",
   "subscription_id" : "b0582c3f-7c57-47c7-a6dd-a010685087ac",
   "resource_group_name" : "IAS-Node-1-new_group",
   "username" : "rootadmin"
   },
   "20.62.200.216" : {
   "machine_name" : "IAS-Node-2-new",
   "subscription_id" : "b0582c3f-7c57-47c7-a6dd-a010685087ac",
   "resource_group_name" : "IAS-Node-1-new_group",
   "username" : "rootadmin"
   },

user_name_ip_map - dictionary to store ip:username. (username- rootadmin)

2. Create connection with database-
   class ServiceLCDatabase:
   DATABASE_NAME = "service_lifecycle"
   COLLECTION_NAME = "service_status"
   SERVICE_NAME = "_id"
   CONTAINER_ID = "container_id"
   SERVICE_STATUS_INACTIVE = "INACTIVE"
   SERVICE_STATUS_ACTIVE = "ACTIVE"
   get_all_services_info()- Store the details of all the services from the above database in a list.
3. Now we start a thread for each machine to check_status.
   - User_name_ip_map will be traversed for each ip and a new thread will be created calling check_status.

- check_status()- here we receive the ip and other details of a machine. Also get the ip of present machine(IP_ADDRESS) where the service lifecycle is itself running.

```
if my_ip == ip:
    client = docker.from_env() #object to connect to server
  Else:  ----other ip machine
    docker_url = "ssh://" + user_name_ip_map[ip] + "@" + ip
    client = docker.DockerClient(base_url=docker_url, timeout=180)
```

Global dictionary - ip_to_client_map = dict()
So we save the client (object) of the given machine with key as ip in ip_to_client_map map.
Check_service_status () is called with the present machine details and the 2 maps- User_name_ip_map and ip_to_client_map.

- Check_service_status ()- Now the client object of the machine will represent the docker client on the machine. So we get the list of running docker containers on the machine. Now we make the list of all the container_id in the on the machine as container_short_id. Now we traverse the service_info json data to get each service name and get its corresponding container_id . Now we try to search this container_id in the container_short_id , if it is found the the given service is on this machine. Now we check the status of the service container-

```
if container_status == 'exited':
        print("Restarting container : ", container_id)
        ……..
else:
        print("Running:",container_id)
```

So if status is exited, first we make the status of the service as inactive in the db and sleep for sometime. Then we call the restart service.

restart_service()- Here we call the load balancer and get the new machine details to restart the service. The machine details will be ip and node_name. Now based on the ip we get the username from user_name_ip_map and docker_client from ip_to_client_map. Now we build and run the docker container. Then check if present machine also has the same one, if so we delete it.

After returning we update the status of service to active.

App Monitoring- Here we read the machine(server details) from server_details.json file. Now for each machine we get the container list of exited status(137,138,143).
containers = client.containers.list(all=True, filters={'exited': [130, 137, 143]})

We then send the container_id as instance_id to topics -**app_monitoring_to_restart** (topic consumer in server_life_cycle - here we check for the instance_id validity if it is an application or not).

Exited codes are :
Exit Code 137: Indicates failure as container received SIGKILL (Manual intervention or 'oom-killer' [OUT-OF-MEMORY])
Exit Code 139: Indicates failure as container received SIGSEGV
Exit Code 143: Indicates failure as container received SIGTERM

Dashboard logs and status
Fetch_status.py script will run to display requested logs and status of containers.
2 threads will run in parallel send_log and send_status
Send_log - consumes from dashboard_to_log(receives container id)
Load config.json to get machines information
Will continuously receives container_id, to process for a container id,

Steps,

Connect to SLC_DB.INSTANCE_ID_TO_SERVER to get find instance_id received.
If any collection is found, will fetch logs from container and pass it on to log_to_dashboard(topic).
Else, received msg(stored in instance_id) could be service container id, will connect to service_lifecycle.server_status database, and fetch collection corresponding to container id received and get logs from containers and send to topic log_to_dashboard.

Instance_id - see sensor_binder, we added key- value pair, (in case of application instance)

```
def sendDeployConfigToScheduler(self, instance_id):
    self.deploy_config["instance_id"] = instance_id
    producer.send(SENSOR_BINDER_TO_SCHEDULER_TOPIC, self.deploy_config)
```

Send_status:
Will fetch containers status of irrespective of whether it is a service container or application container, will connect to both database SLC_DB and service_lifecycle and get container id from it and will send dictionary (key value pair like containers_status[instance_id] = cont_status) to app_monitoring_to_dashboard

Sample Application
To build application developer will receive Template(AppConfig, sample algo), platformLib.py
appConfig will be filled by developer.
platformLib.py- will get imported in main algo.py for getting topics related to sensor.
APIs to be called are defined here, algo will call a respective function from paltformLib that'll hit platform API (platform_manager.py - rest API) to get respective topics (asked topic).

appConfig.json filled by developer for lightControl.py algorithm-

```
{
  "application_name": "Bus_management",
  "application_id": 2,
  "developer_id": 869,
  "environment":{
    "os": "Linux",
    "modules": [
      "pandas",
      "numpy"
    ]
  },
  "algorithm_list":[
{
      "algorithm_name":"lightControl",
      "script":{
        "name":"lightControl.py"
      },
      "input_sensors":[
        {
          "sensor_type":"light-sensor"
        },
        {
          "sensor_type":"gps-sensor"
        }
      ]
    }
]
}
```

When we are creating the docker file of the algorithm instance , there we run the algorithm by giving argument as its instance id.-

```
"CMD [\"python\", \"-u\"," + "\"{}\",".format(file_name)+  "\"{}\"".format(instance_id) + "]\n".
```

sys.argv[1]- This is the instance id which is given by platform itself when running an instance of the algorithm.

When deploying the algorithm, the deployment details will be filled by the application configurater -

deployConfig.json---- for lightControl.py algorithm----------

```
{
  "noOfAlgo":1,
```

```
  "1":{
    "application_name":"Bus_management",
    "script_name" : ["lightControl.py"],
    "algorithm_name":"lightControl",
    "sensor_info":[
      {
        "sensor_type":"light-sensor",
        "filter_sensors":[
          {
            "bus-id":"1"
          }
        ]
      },
      {
        "sensor_type":"gps-sensor",
        "filter_sensors":[
          {
            "placeholder":"bus",
            "bus-id":"1"
          }
        ]
      }
    ],
    "scheduling_info":{
      "request_type": "start",
      "start_time": "",
      "end_time": "",
      "days": "",
      "interval": "30",
      "repeat": "no",
      "job_id": "123 exists only if request type is stop"
    },
    "environment":{
      "lang":"python",
      "dependencies": [["requests", ""], ["kafka-python", ""], ["pymongo", ""], ["dnspython", ""]]
    }
  }
}
```


lightControl.py -

To control lux level in a bus (specific bus), this algorithm is built, so ability to control sensor will be provided too apart from just getting data from a sensor.

Consumer topics (to get data from sensor)

bus_light_topicName = platform_libfile.getSensorData(sys.argv[1],0)

bus_gps_topicName = platform_libfile.getSensorData(sys.argv[2],0) // sys.argv
Producer topic - (to control sensor)
light_Control_topic_name = platform_libfile.setSensorData(sys.argv[1],0)

In lightControl.py

```
def lightControl():
        #bus_light_topicName = platform_libfile.getSensorData(sys.argv[1],0)
        bus_light_topicName = 'bus_light'

        #bus_gps_topicName = platform_libfile.getSensorData(sys.argv[2],0)
        bus_gps_topicName = 'bus_gps'
```

Consumer connection is created based on topic provided, and data on both topics are continuously consumed by the algo, whenever we found lux_level < 40, will produce data in a control topic like producer.send(light_Control_topic_name, '1') so sensor will update its lux level
To display bus_id and lux level on dashboard, it will produce data on topic bus_id (eg bus_1) and it is consumed by platform_manager.py

API for dashboard (defined in platform_manager.py)

```
@app.route('/dashboard/bus_refresh_<string:bus_id>', methods=["GET"])
dahsboardMsg = json.dumps({"Light": 'Switch on light as lux is {}'.format(light)})
#print("topic name : ",'bus_'+bus_id)
producer.send('bus_'+bus_id,dahsboardMsg)
```

Microservices are a hot topic in system design interviews. It is important to know why we use them instead of monolithic systems. The short answer is: Scalability. The detailed one would be:

Advantages:
1) The microservice architecture is easier to reason about/design for a complicated system.

2) They allow new members to train for shorter periods and have less context before touching a system.
3) Deployments are fluid and continuous for each service.
4) They allow decoupling service logic on the basis of business responsibility
5) They are more available as a single service having a bug does not bring down the entire system. This is called a single point of failure.
6) Individual services can be written in different languages.
7) The developer teams can talk to each other through API sheets instead of working on the same repository, which requires conflict resolution.
8) New services can be tested easily and individually. The testing structure is close to unit testing compared to a monolith.

Microservices are at a disadvantage to Monoliths in some cases. Monoliths are favorable when:
1) The technical/developer team is very small
2) The service is simple to think of as a whole.
3) The service requires very high efficiency, where network calls are avoided as much as possible.
4) All developers must have context of all services.