# Generating a realistic IoT data stream using Python and storing into MongoDB Timeseries Collection. – Part 1

A Tutorial (of sort) where we take the reader through the creation of an IoT timeseries data stream using Python and store the data into an MongoDB Timeseries Collection.

(18 November 2024)

**Overview**

What follows will be a multi part expedition (tutorial) exploring the writing of a Python application to generate as realistic as possible sensor data (also commonly known as IoT, Internet of Things) and finally storing the data into a time series specific collection inside MongoDB on Atlas.

Allot can be said about time series data, and the multitude of use cases, but at the core it is any value that can be associated with a point in time. That’s it at its simplest definition. Once that’s understood it will be realized how big this paradigm is and how useful it can be.

There are a loads of publicly available data generators, that can all create fake data (commonly referred to as datagen’s), but data is only useable if it is realistic, relevant. Nothing against them, they have their purpose.

What am I saying… well a sensor cannot jump from 20deg C to 30Deg C between readings if the readings are say 1 seconds apart. An electric motor cannot run at 1500 RPM and then 2 seconds later at 2500 RPM. With some digging around how fake data is normally created it is quickly realized that a uniform distribution pattern is most likely utilized, giving an even distribution of data points between the min & max values giving.

This is all good if you just want to show a pipeline works end to end, and that it can handle the load/throughput/volume of data, but when the stream is supposed to try and show normal behavior and then potentially show a deviation from the norm, say for Machine Learning, well then points all over the spectrum becomes useless.

Now, regarding device instrumentation, and we will simply think here of industrial machines, gas fired ovens, electric pressure ovens, conveyer belts, oil pumps, fuel pumps, cooling pumps, etc.



Some possible sensors, a pump, it draws power, which can be measure as voltage and amperage.

It can have temperature sensors fitted; it can have a volume/flow rate sensor, it can have a revolution/speed counter.



Above we show a pressurized oven, which means we can fit pressure sensor, we can measure the power draw via voltage and amperage again. We can fit a temperature sensor.

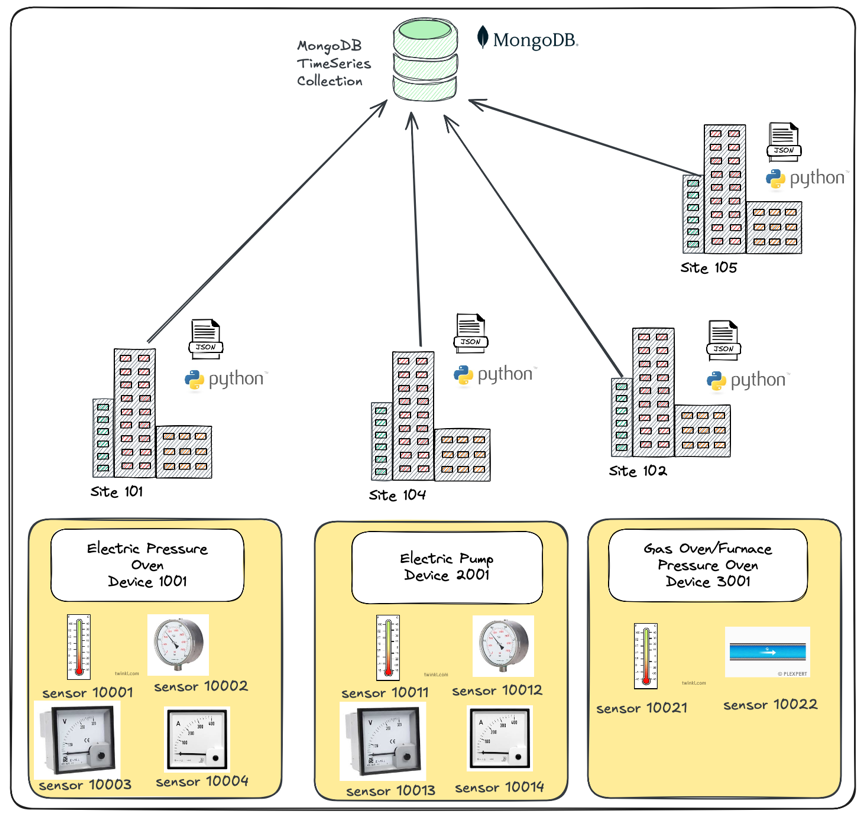


Not directly part of our sites/devices, but to explain the concept. Each wind farm can be a site. Each turbine can be a device, configured with multiple sensors. We can have a speed sensor to measure the blade speed, we can have a pressure sensor to measure the atmospheric pressure, we will have an electric generator, which means we can have a voltage and amperage sensor. For the generator we can have a temperature sensor.

Ok, so… what follows is my attempt to depict a manufacturing company, that has multiple factories/operating locations (we will refer to them as sites), each with multiple machines (referred to as devices).

Each device will have multiple sensors. As far as possible I tried to make the sensors relevant to the device. I’ve modelled the seed data to represent pre imagine combinations…

We can specify if the factories operate 24 hours a day, or only work during normal business hours (8-5).



I’ve also written the application in such a way that we can start by creating historical data, as fast as possible before changing into a current time mode where we create data and go into a sleep mode (for a specified amount of time)… as if the sensor is configured to only take a reading every X millisecond/seconds/minutes.

“Springled” into my seed data configuration are devices that are configured to miss behave, naughty they shall be, be that from a specified time in the past, or daily between a start end time.

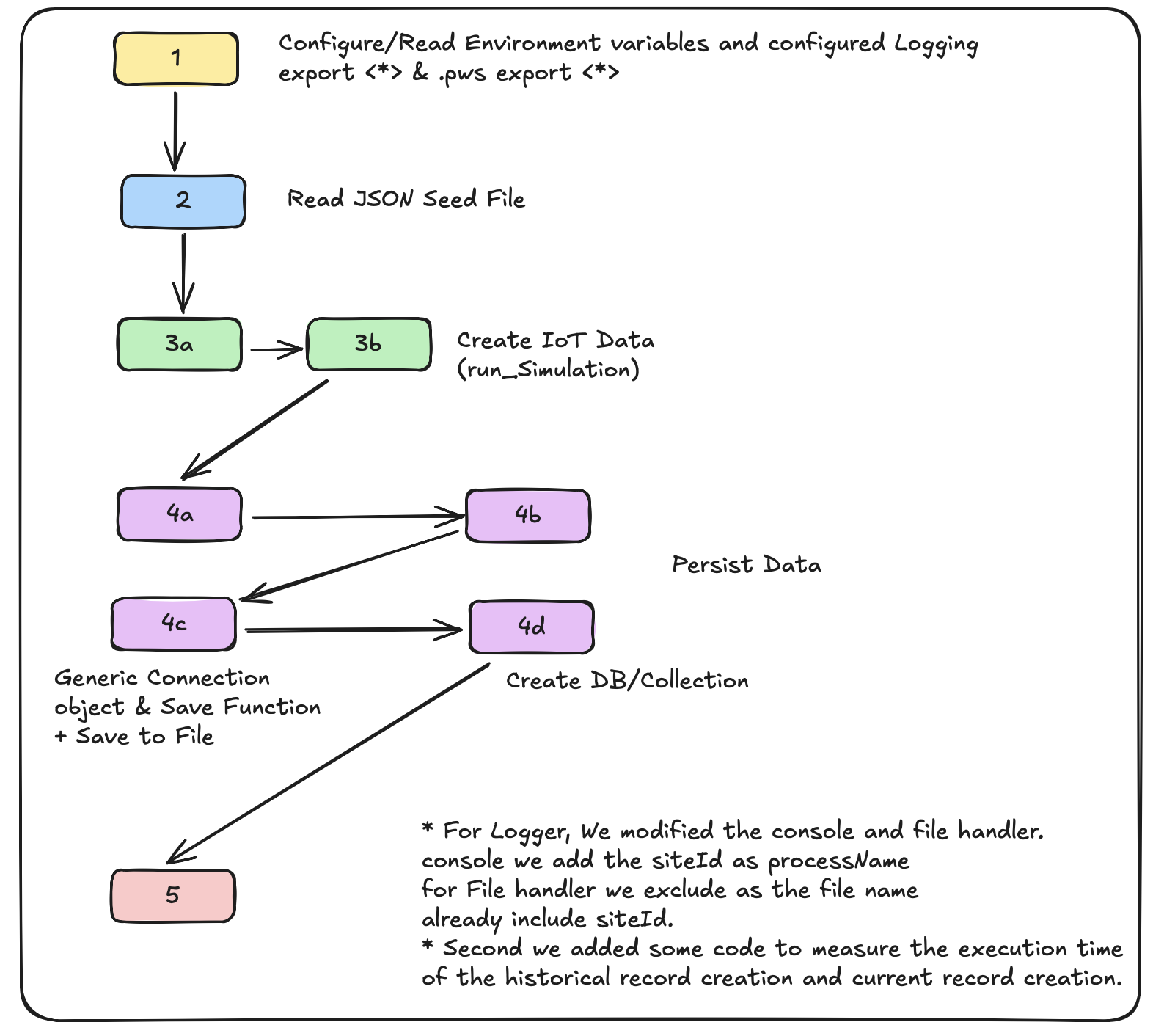
Hey… machines get tired… instead of working at 100% they might drop to say 70% at 3pm, and that then have down stream impact. Which impacts stability of the device or well that’s what we will explore, try and find in the follow up using Machine Leaning, and a later article.

The tutorial (of sort) is broken up as follows:

1. Reading our general environment variables and secondly configuring a basic logging to console and file framework.
2. Reading our JSON based seed file that defines our factories (sites), machinery (devices) and the sensors configured for the device.
3. Generating the sensor stream of measurements/readings.
4. Persisting our stream to first MongoDB Collection and then as JSON strings into a file (which can also be uploaded into MongoDB).
5. Polishing things… addressing some lessons learned above.

* Part 1 is short and simple.
* Part 2 we read in our JSON structured file, then pick out the factories specified in our SITESID environment variable read in in part 1.
* Part 3 is the entire data generation, this is the meat, this is long. To note, the reader will notice there is a section 3a and 3b in the code, 3a was the first simple, get it working. This section was then copied to 3b and improved on.
* Part 4 is where we push the stream into our MongoDB located time series specific collection.
* Again, we have a 4a, 4b, 4c… 4a we get the basics done, 4b is a copy of 4a with improvements, 4c is a copy of 4b and so on… Get the idea ;)
* Part 5 I make a change to logging, aka lessons learned from the above and secondly, I add some instrumentation to be able to report on start/stop/runtime/throughput.

Below is a diagram depicting how the end state was arrived at.



The entire Project can be found in my [Github repo](https://github.com/georgelza/Python_IoT_Generator).

This tutorial will not be as very beginner level, so I’m assuming you have the basic understanding of Python already, know what I mean by running a *virtualenv* environment, understand how to install packages etc.

Each of the *app1, app2* … directories have a local *README.md* file that explains what added/changed in the section.

That version of the app can be executed/run by running the run1.sh run2.sh and so one or by executing “*make app1”, “make app2”* in the root folder.

Where needed (as we progress) the *runX.sh* files are modified with the additional export of variables as required, similarly we have a version specific seed files located in *conf/.* As the tutorial progresses, we modify the structure of the file to accommodate the functionality as we grow our application.

For simplification, for now the application is not containerized. This does mean it’s advised to run a local *virtualenv* environment with the required packages installed in that environment.

For Mongo there is a containerized MongoDB Atlas, that can be stood up in section 4 via the make file located in the *devlab* directory. The README.md file will tell you what to do when.

Re comments. Where I introduced a new concept, or a new idea, I have added comments, or well more than just a simple comment… ;), this will however have been removed in the next version, to try and keep it clean. This also helps with drawing the reader’s attention to what’s new in the specific version.

**Part 1 / app1**

At this point, let’s start, all of this is contained in the app1 sub directory, part2 will be in app2 and so on.

In this section as per previously mentioned I will address general environment variables and the basic logging framework.

**Environment Variables:**

There are many ways to do this. My chosen option is a simple environment variable export using the:

*export <variable\_name>*

syntax.

For our app these are contained in the runX.sh script, but can also easily be part of a *Dockerfile* environment variables used during a docker run command or specified in a *docker-compose.yaml* as an environment variable. The benefit here is that the code to use it does not change at all.

This is then pulled into a variable using the below code. My environment configuration code lives under *app1/utils.py* in the *configparams* function.

```

*def configparams():*

*config\_params = {}*

*# General*

*config\_params["DEBUGLEVEL"] = int(os.environ["DEBUGLEVEL"])*

*config\_params["ECHOCONFIG"] = int(os.environ["ECHOCONFIG"])*

*config\_params["ECHORECORDS"] = int(os.environ["ECHORECORDS"])*

*config\_params["LOGGINGFILE"] = os.environ["LOGGINGFILE"]*

*return config\_params*

*# end configparams*

```

The function is called as per below. See app1/main.py, line 42

```

config\_params = utils.configparams()

```

**Logging Frame**

The logging functionality is instantiated using the logger function also contained in the utils.py file.

I’ve defined 2 loggers, namely a console logger and a file logger.

The logging level is defined by an environment variable called, *DEBUGLEVEL=X*

See *run1.sh* line 5:

The logging levels themselves being:

DebugLevels logging.setlevel

0 Debug -> Allot of information will be printed, including printing site configuration to logfile.

1 Info -> We just printing that processes is starting/ending

2 Warning -> Will decide

3 Error -> used in any try/except block

4 Critical -> used when we going to kill the program.

The logger is instantiated using the following code located in app1/main.py line 45

```

*logger = utils.logger(config\_params["LOGGINGFILE"] + "\_common.log", config\_params["CONSOLE\_DEBUGLEVEL"], config\_params["FILE\_DEBUGLEVEL"])*

```

As can be seen we pass our environment variable configured above into the *utils.logger* function located in *app1/utils.py.*

At this point we can call the function using the following code, 3 examples provided, see the main function in *app1/main.py* for more.

```

*logger.debug('Hello, This is a Debug (level=0) Message with no variables added.')*

*# Quick Example of how we can print a info message to the console, no added variables*

*logger.info('Hello, This is a Info (level=1) Message with no variables added.')*

*# same as above but we now add a variable to include/append to our text, the {logfile} variable does not need to be at the end of the line either.*

*logger.info('Hello, this is a Info (level=1) message with a single variable : {logfile}'.format(*

*logfile=config\_params["LOGGINGFILE"]*

*))*

```

We’ve covered what we will be building and shown the basic start.

For now, this will be it… See you in part 2.

Good luck, this is all fraught with rabbit holes, as always, so many and you can disappear so easily… but then that’s ½ the fun.



**About Me**

I’m a techie, a technologist, always curious, love data, have for as long as I can remember always worked with data in one form or the other, Database admin, Database product lead, data platforms architect, infrastructure architect hosting databases, backing it up, optimizing performance, accessing it. Data data data… it makes the world go round.

In recent years, pivoted into a more generic Technology Architect role, capable of full stack architecture.

[George Leonard](https://www.linkedin.com/in/george-leonard-945b502/)

[georgelza@gmail.com](mailto:georgelza@gmail.com)

<https://www.linkedin.com/in/george-leonard-945b502/>

<https://www.mongodb.com/community/creators/>

<https://medium.com/@georgelza>

# Generating a realistic IoT data stream using Python and storing into MongoDB Timeseries Collection. – Part 2

A Tutorial (of sort) where we take the reader through the creation of an IoT timeseries data stream using Python and store the data into an MongoDB Timeseries Collection.

(19 November 2024)

**Overview**

Ok, so we have our environment variables read into *config\_params* variable, and we have a logger object capable of outputting logging information to the console and to a file.

Next up in part 2, (the code will be found in app2/ directory) we will read in our seed file. The seed file (conf/Full2.json) defines our company sites, devices and the sensors per device.

First up, see app2/main.py, line 45.

```

*my\_seedfile = utils.read\_seed\_file(config\_params["SEEDFILE"], logger, config\_params)*

```

As can be seen we pass our *config\_params* into the *read\_seed\_file* call.

*utils.read\_seed\_file* is contained in the *app2/utils.py* file at line 135.

Once the seed file/data is retrieved using the above function call I pass it into a function that does a filter for us returning only the site we’re interested in. The sites we want to process/generate data for is defined via our *SITEIDS* environment variable.

This is done as per below (*main.py*, line 53)

```

*# Load/filter out the site data based on SITESIDS environment variable.*

*my\_sites = []*

*for siteId in config\_params["SITEIDS"]:*

*cur\_site = utils.find\_site(my\_seedfile, int(siteId), logger)*

*if cur\_site == -1:*

*os.\_exit(1)*

*# end if*

*utils.pp\_json(cur\_site)*

*my\_sites.append(cur\_site) # We will use this in app3*

*# end for*

```

*utils.find\_site* can be found in app2/utils.py line 190.

We basically just loop over the main *my\_seedfile* variable, comparing to see if the siteId of the record equals the *siteId* passed in as a variable. If we find our desired record, we simply return it. If we get to the end of the array, well then, we return -1 implying we did not find our record.

```

*"""*

*Find the specific site in array of sites based on siteId*

*"""*

*def find\_site(my\_seedfile, siteId, logger):*

*logger.info('utils.find\_site Called, SiteId: {siteId}'.format(*

*siteId=siteId*

*))*

*site = None*

*found = False*

*for site in my\_seedfile:*

*if site["siteId"] == siteId:*

*logger.info('utils.find\_site Retrieved, SiteId: {siteId} '.format(*

*siteId=siteId*

*))*

*return (site)*

*if (found == False):*

*logger.critical('utils.find\_site Completed, SiteId: {siteId} NOT FOUND '.format(*

*siteId=siteId*

*))*

*return (-1)*

*# end find\_site*

```

For now, this will be it… See you in part 3, hope this is all interesting.

Good luck, this is all fraught with rabbit holes, as always, so many and you can disappear so easily… but then that’s ½ the fun.



**About Me**

I’m a techie, a technologist, always curious, love data, have for as long as I can remember always worked with data in one form or the other, Database admin, Database product lead, data platforms architect, infrastructure architect hosting databases, backing it up, optimizing performance, accessing it. Data data data… it makes the world go round.

In recent years, pivoted into a more generic Technology Architect role, capable of full stack architecture.

[George Leonard](https://www.linkedin.com/in/george-leonard-945b502/)

[georgelza@gmail.com](mailto:georgelza@gmail.com)

<https://www.linkedin.com/in/george-leonard-945b502/>

<https://www.mongodb.com/community/creators/>

<https://medium.com/@georgelza>

# Generating a realistic IoT data stream using Python and storing into MongoDB Timeseries Collection. – Part 3

A Tutorial (of sort) where we take the reader through the creation of an IoT timeseries data stream using Python and store the data into an MongoDB Timeseries Collection.

(20 November 2024)

**Overview**

Ok, this is where things get real. Let generate our sensor data… This is part 3, we’ll start in app3\*. But what now, why do we have “\*”’ in the app name…

Well, app3a is the first version. It’s the bare minimum to do what we want. Some will call this more professionally the MVP (Minimum viable product).

After that app3b is created, it starts off with app3a as base and then we expand/improves on it. We can consider this as the production version.

I did it this way so that the reader can see the progression or is that how my mind (happily referred to as the “hamster”) works.

<This section, part 3, might end as 2 sections due to the amount of information that we need to cover>.

If you look at the below code, you will notice no comments. I removed the comments here to make the code more compact, see the git repo for the full commented versions.

Our new code starts in *app3a/main.py* line 90 onwards:

```

*current\_time = datetime.now()*

*processes = []*

*for site in my\_sites:*

*logger.info("Calling simulate.run\_simulation for SiteId {siteId}".format(*

*siteId=site["siteId"]*

*))*

*p = multiprocessing.Process(target=simulate.run\_simulation, args=(site, current\_time, config\_params))*

*processes.append(p)*

*p.start()*

*# end for*

*for p in processes:*

*p.join()*

*# end for*

*logger.info("COMPLETED run, logfile => {logfile}".format(*

*logfile=config\_params["LOGGINGFILE"]*

*))*

*# end main*

*if \_\_name\_\_ == "\_\_main\_\_":*

*main()*

```

So, what do we have here….

First, we get what the actual execution time is, as in right now, and then we create a variable called processes.

After the above, we simply loop over our *my\_sites* array. That’s the easy bit.

We then use *multiprocessing.Process* to call *run\_simulation* with some variables, the more not so easy bit. This is where the processes variable is used, but more on this after the below.

I’m first going to discuss the *run\_simulation* function and its associated helper functions which can be found in a*pp3*a and b*/simulate.py*.

“*run\_simulation*” and the associated helper functions are our entire body of creating our sensor data.

The functions contained in here are:

* is\_within\_time\_range
* generate\_payload
* progress\_value
* run\_simulation

Let’s have a short description about each…

**Function**: *is\_within\_time\_range*

This simply takes 3 variables, *start\_time, end\_time* and *current\_time* and returns a true of false based on if *current\_time* is within the start and end times.

```

*def is\_within\_time\_range(start\_time\_str, end\_time\_str, current\_time=None):*

*start\_time = datetime.strptime(start\_time\_str, "%H:%M")*

*end\_time = datetime.strptime(end\_time\_str, "%H:%M")*

*if current\_time is None:*

*current\_time = datetime.now()*

*return (start\_time <= current\_time <= end\_time)*

*# end is\_within\_time\_range*

```

**Function**: *generate\_payload*

In here we generate our timeseries based payload. To accomplish this, we pass in our sensor array, our device array, the *site\_id,* the *current\_time* and *time\_zone\_offset* as variables.

Re *current\_time*… it’s the time we want to associate the sensor measurement with, to be recorded as the timestamp value.

Re, *time\_zone\_offset*, we take the current time and apply the *time\_zone\_offset* to it, thus making all timestamp values for all measurements UTC based.

After that we simply build the payload by constructing our JSON object and assigning values to it from our device and sensor array’s.

Below is the code from *app3a/simulate.py* line 55.

```

*def generate\_payload(sensor, device, site\_id, current\_time, time\_zone\_offset):*

*offset\_hours, offset\_minutes = map(int, time\_zone\_offset.split(":"))*

*tz\_offset = timezone(timedelta(hours=offset\_hours, minutes=offset\_minutes))*

*local\_time\_with\_tz = current\_time.replace(tzinfo=tz\_offset)*

*timestamp = local\_time\_with\_tz.isoformat()*

*measurement = progress\_value(sensor, device["stabilityFactor"], device, datetime.now())*

*return {*

*"timestamp": timestamp, # Timestamp now includes the correct timezone*

*"metadata": {*

*"siteId": site\_id,*

*"deviceId": device["deviceId"],*

*"sensorId": sensor["sensorId"],*

*"unit": sensor["unit"]*

*},*

*"measurement": measurement*

*}*

*# end generate\_payload*

```

**Function**: *progress\_value*

Now this is where things get interesting…

First, we pass in our sensor definition via the sensor variable, we then pass in *stabilityFactor* which originates from the device section and then again, our *current\_time* and then a method value.

I’m going to start with *current\_time,* as per above in *generate\_value.* It’s the time that we will assign to timestamp field.

Sensor well that’s the detail we read if from our seed file that defines the configuration of the sensor. Well, what’s that. Let’s see. See *conf/Full3a.json*

Each sensor has defined with it, all values are required.

* sensorId
* min\_range
* max\_range
* unit
* sensorType
* sd
* mean

Additionally, before we discuss the above lets also list the values for device array:

* deviceId required
* deviceType required
* stabilityFactor required
* start\_time optional
* end\_time optional
* sensors required

*deviceId*, is the machine Id, unique across entire estate.

*deviceType,* is simply a description of the device, gas oven, electric oven, water pump, oil pump, fuel pump, etc.

s*tabilityFactor* and *start\_time* and *end\_time* goes together. *stabilityFactor* is a number/percentage to apply to the measurement value generated, if the *current\_time* is within the *start\_time / end\_time*.

And then the sensors are an array of documents, each document the definition of a sensor as listed above.

*sensorId*, is a unique id for the sensor across the estate.

*sensorType*, is the type of sensor, i.e. temperature sensor, pressure sensor, voltage sensor, ampage sensor, rpm sensor etc.

*min\_range*, a bottom we don’t want to cross.

*max\_range*, a high level we don’t want to cross.

*sd,* standard deviation

mean, well mean… the normal value we want to keep to.

Now regarding the *progress\_value* function inside app3\*/simulate.py…

We start by saying, do we have a *start\_time, end\_time* specified at the device level, if yes then we see if the current time is within the start/end time… if yes then we continue to apply the *stabilityFactor* to the sd.

By default, we follow a normal distribution based on mean and sd values.

Now if you look at line 111 you will notice we don’t always use the sd as per the sensor record.

See *app3a/simulate.py* line 91

```

*def progress\_value(sensor, stability\_factor, device, current\_time, method="normal"):*

*sd = sensor["sd"]*

*mean = sensor["mean"]*

*if "start\_time" in device and "end\_time" in device:*

*device\_start\_time = datetime.strptime(device["start\_time"], "%H:%M")*

*device\_end\_time = datetime.strptime(device["end\_time"], "%H:%M")*

*current\_time\_local = current\_time.time() # We only care about the time portion*

*if device\_start\_time.time() <= current\_time\_local <= device\_end\_time.time():*

*# Scale standard deviation if within the specified time range*

*sd = sd \* (100 - stability\_factor) / 100*

*if method == "normal":*

*return round(random.gauss(mean, sd), 4)*

*elif method == "uniform":*

*lower\_bound = max(0, mean - sd)*

*upper\_bound = mean + sd*

*return round(random.uniform(lower\_bound, upper\_bound), 4)*

*# end progress\_value*

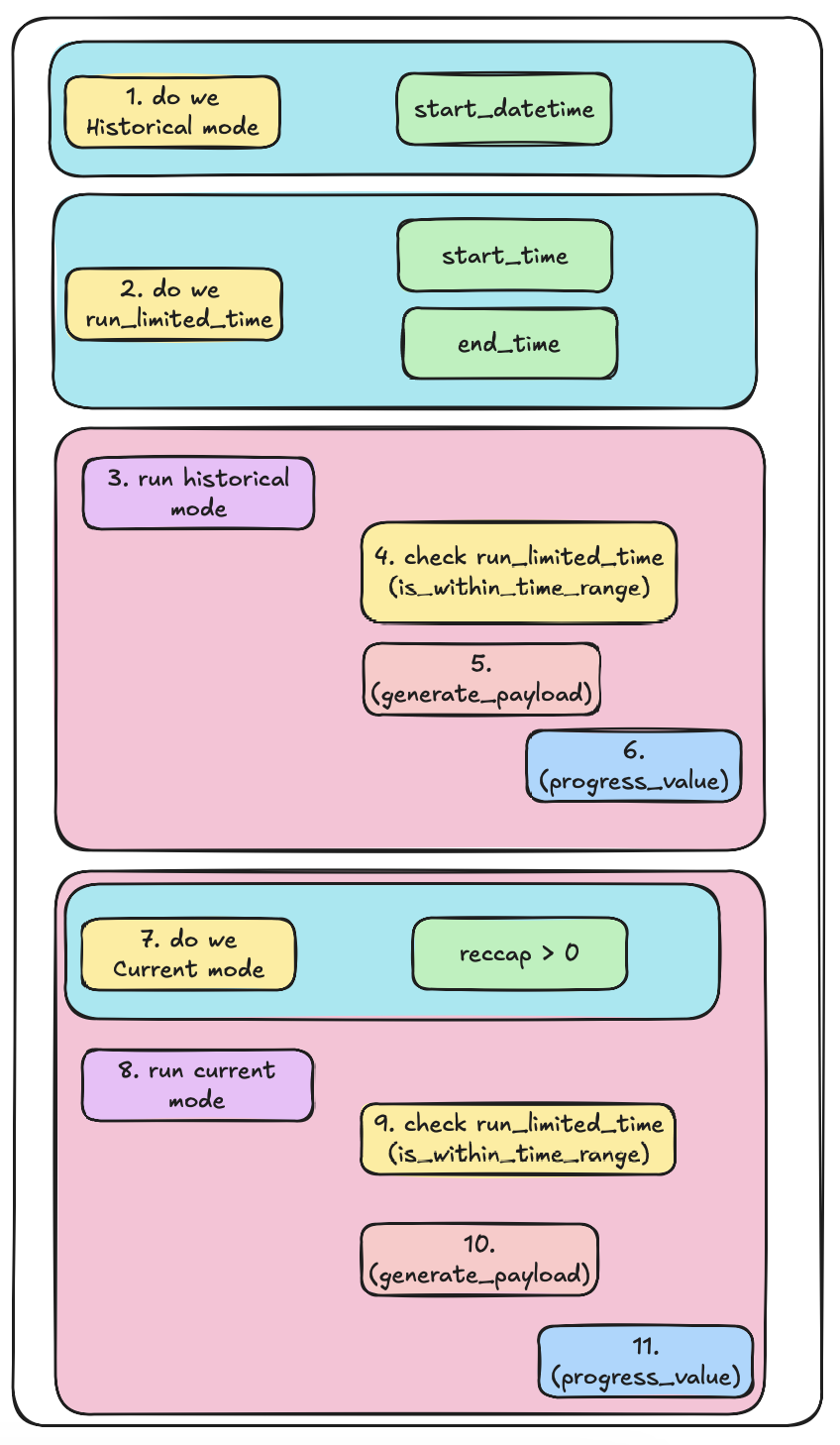
```

**Function**: *run\_simulation*

Firstly, we previously had one logger defined, created in main.py in main function… due to us now using multiprocessing, we could do one of 2 things, use that as a global logger or what I’ve chosen to do. I start here by first creating a local logger inside *run\_simulation,* allowing me to set settings per specific site. Because of this you will see for app3b we have a new Full3b.json file, which now include site specific logging settings. We then modify the logger file name to include the *siteId*.

What was also done at this stage is the splitting of the single set of settings being used to separate settings for console logger and file logger, allowing us to echo less information to the screen vs the file.

Next, we originally had one time zone offset, globally, what we now have is a site-specific offset, to go with the above, we moved the time zone offset to the site level definition. Simply put, if we have factories all over the globe, each of them has their own offset from UTC/GMT time. All values generated carry a timestamp that’s UTC based.



First, we check if we need to enable historical mode, this is done by checking if *start\_datetime* is specified at site level.

Next up, we see if the factory run 24 hours or during a specified times as specified daily business hours. This is accomplished by first checking if *start\_time* and *end\_time* is specified at the site level. If yes then we have a reduced run time per day, running is then controlled by a variable “*run\_limited\_time*“ (step #2), see simulate.py line 154.

We now go into data generation, based on our rules.

First, we execute Historical data generation mode (step 3). What we do here is create old data, as if we’ve been running the app for months or years. We enable/disable this functionality by specifying *start\_datetime* at the site level, if specified then we want to run the historical data generation blocked defined on line 159 in app3a/simulate.py.

This is done by setting *oldest\_time* to *start\_datetime* configured for the site.

We then check if we’re in *run\_limited\_time,* if yes, let’s see if oldest time is within the run\_limited\_time values.

If our oldest time (time we’re currently simulating is outside the *run\_limited\_time* times) then simple increment the time with the site-specific sleep value and loop back to while at the top, otherwise we proceed. And proceed here means make a call to *generate\_payload* (step 5), followed by incrementing the *oldest\_time* and looping again. Inside this loop we call step 6.

All the above is done at how ever fast the platform can execute the code, allowing the user to create simulated data for many years in the past, up to the current time, the NOW.

Once the above has completed we go into Current data generation mode, this starts with step 7.

This is governed by the *reccap* variable set at site level. What this does is… define how many records/loops we want to execute, each with a sleep time between each loop, so this can be 1000 loops with .1 seconds, or 100 of 60 seconds, or set *reccap* crazy high and it simulates just continuous running.

Other than the above, all we do inside the loop is to check if the current execution time is inside or outside the *run*\_*limited*\_*time* block (step 9). Followed by *generate\_payload* (step 10) which calls *progress\_value* (step 11).

And that’s our data generation block for app3a.

```

*def run\_simulation(site, current\_time, config\_params):*

*logger = utils.logger(config\_params["LOGGINGFILE"] + "\_" + str(site["siteId"]) + ".log", site["console\_debuglevel"], site["file\_debuglevel"])*

*logger.info("simulate.run\_simulation - Site ID {siteId}: Starting simulation".format(*

*siteId=site["siteId"]*

*))*

*logger.debug('simulate.run\_simulation - Site ID {siteId}: Printing Complete site record')*

*utils.pp\_json(site, logger)*

*site\_time\_zone = site.get("time\_zone", site["time\_zone"])*

*if "start\_datetime" in site and site["start\_datetime"]:*

*oldest\_time = datetime.strptime(site["start\_datetime"], "%Y-%m-%dT%H:%M")*

*else:*

*oldest\_time = current\_time*

*run\_limited\_time = "start\_time" in site and "end\_time" in site*

*if run\_limited\_time:*

*start\_time = site["start\_time"]*

*end\_time = site["end\_time"]*

*# Historical phase*

*if "start\_datetime" in site and site["start\_datetime"]:*

*logger.info("simulate.run\_simulation - Site ID {siteId}: Running historical phase starting from {start\_datetime}".format(*

*siteId=site["siteId"],*

*start\_datetime=site["start\_datetime"]*

*))*

*while oldest\_time < current\_time:*

*if run\_limited\_time:*

*if not is\_within\_time\_range( continue*

*for device in site["devices"]:*

*for sensor in device["sensors"]:*

*payload = generate\_payload(sensor, device, site["siteId"], oldest\_time, site\_time\_zone)*

*logger.debug("simulate.run\_simulation - Hist Ph: Payload {payload}".format(*

*siteId=site["siteId"],*

*payload=payload*

*))*

*sensor["last\_value"] = payload["measurement"]*

*oldest\_time += timedelta(milliseconds=site["sleeptime"])*

*logger.info("simulate.run\_simulation - Site ID {siteId}: Completed historical phase starting from {start\_datetime}".format(*

*siteId=site["siteId"],*

*start\_datetime=site["start\_datetime"]*

*start\_time, end\_time, oldest\_time):*

*oldest\_time += timedelta(milliseconds=site["sleeptime"]) # Skip out-of-range times*

*))*

*# end Historical phase*

*# Current phase*

*if site["reccap"] > 0:*

*logger.info("simulate.run\_simulation - Site ID {siteId}: Running current phase".format(*

*siteId=site["siteId"]*

*))*

*for loop in range(site["reccap"]):*

*current\_loop\_time = oldest\_time + timedelta(milliseconds=site["sleeptime"] \* loop)*

*if run\_limited\_time:*

*if not is\_within\_time\_range(start\_time, end\_time, current\_loop\_time):*

*continue*

*for device in site["devices"]:*

*for sensor in device["sensors"]:*

*payload = generate\_payload(sensor, device, site["siteId"], current\_loop\_time, site\_time\_zone)*

*logger.debug("simulate.run\_simulation - Cur Ph: Payload {payload}".format(*

*siteId=site["siteId"],*

*payload=payload*

*))*

*sensor["last\_value"] = payload["measurement"]*

*time.sleep(site["sleeptime"] / 1000) # Convert milliseconds to seconds*

*logger.info("simulate.run\_simulation - Site ID {siteId}: Completed current phase".format(*

*siteId=site["siteId"]*

*))*

*logger.info("simulate.run\_simulation - Site ID {siteId}: Completed simulation".format(*

*siteId=site["siteId"]*

*))*

*# end run\_simulation*

*```*

And now onto app3b.

What have we changed: see *app3b/ README.md*

*app3b/simulate.py -> progess\_value*

As we now via app3a have a working MVP, we’re now going to step things up a bit.

Firstly, we add a deviation weight to the sensor’s configuration. As such there is *Full3b.json* now (previously we used *Full3a.json*).

See *app3b/simulate*, line 114,

The norm is 5. 5 means we try and keep the *sd* stable measurements created close around the mean value, implying normal stable operation.

If the weighting is 5-7 then slowly, raise the *sd,* causing the measurements generated to climb. If 7-10 then we push the turbo button (or is that disaster button) and *sd* is drastically increased.

If the weighting is below 5 then the simple opposite is done.

See *app3b/simulate*, line 128,

We also added a rule, if *stabilityFactor* is below 50 then *sd* will allow the calculated metric/measurement to go over the max and below the min value specified via *max\_range* and *min\_range*.

By having packaged the creation of the measurement into *run\_simulate* and the using *progress\_value* as we did, it allowed us at this point to have accomplished the above improvement without any changes in *run\_simulation* itself. All the changes were localized in *progress\_value* function.

One last thing. Now that we have the data generation side sorted…

As you will recall, we’re using multiprocessing (*app3a/main.py* line 98 and *app3b/main.py* line 106).

This allows us to call a totally isolated process for each site, allowing them to run in parallel, in complete isolation. This is partly why in 3a we moved from one global logger to a local logger in *run\_simulation*, why we moved the logging level parameters from global to a site-specific values, and how we now create the log file name (for the file logger) with the *siteId* appended.

Oh ye… and lastly, I did rename some variables as you would have noticed, as we had similar names across our seed-file, and I figured it would cause some confusion somewhere down the line.

For now, this will be it… See you in part 4.

Good luck, this is all fraught with rabbit holes, as always, so many and you can disappear so easily… but then that’s ½ the fun.



**About Me**

I’m a techie, a technologist, always curious, love data, have for as long as I can remember always worked with data in one form or the other, Database admin, Database product lead, data platforms architect, infrastructure architect hosting databases, backing it up, optimizing performance, accessing it. Data data data… it makes the world go round.

In recent years, pivoted into a more generic Technology Architect role, capable of full stack architecture.

[George Leonard](https://www.linkedin.com/in/george-leonard-945b502/)

[georgelza@gmail.com](mailto:georgelza@gmail.com)

<https://www.linkedin.com/in/george-leonard-945b502/>

<https://www.mongodb.com/community/creators/>

<https://medium.com/@georgelza>

# Generating a realistic IoT data stream using Python and storing into MongoDB Timeseries Collection. – Part 4

A Tutorial (of sort) where we take the reader through the creation of an IoT timeseries data stream using Python and store the data into an MongoDB Timeseries Collection.

(21 November 2024)

**Overview**

Data persistence… All of the above was done to get here… Without this section, we just burning coal (spending money on CPU cooling) ;)

So, up first we’re going to push our data into a MongoDB Collection, which is configured as a timeseries specific collection.

For the MVP (*app4a*) we will pre create our database and collection via the MongoDB Atlas Web console interface or Mongo Compass application.

Our biggest add here is *app4\*/collection.py*, which will contain all our code to save records to a target. Notice we also now using *Full4.json*. In *Full4.json* we add 2 new variables to the site definition:

* data\_persistence
* flush\_size

data\_persistance will either be:

0 – No persistence.

1 – MongoDB persistence

2 – Text file persistence as implemented in app4c/collection.py.

*flush\_size* defines how many records to pack together when we are utilizing MongoDB’s *insertMany* option.

In app4b, we consolidate the *insertOne* and *insertMany* used into *insertMongo*.

**Function:** *insertOne*:

```

*def insertOne(mongodb\_collection, siteId, payload, logger):*

*try:*

*result = mongodb\_collection.insert\_one(payload)*

*return result*

*except Exception as e:*

*logger.error('connection.insertOne - Site ID {siteId}, insertOne - FAILED, Err: {err}'.format(*

*siteId=siteId,*

*err=e*

*))*

*return -1*

*# end try*

*# end insertOne*

*```*

*Function: insertMany:*

*```*

*def insertMany(mongodb\_collection, siteId, payload, logger):*

*try:*

*result = mongodb\_collection.insert\_many(payload)*

*return result*

*except Exception as e:*

*logger.error('connection.insertMany - Site ID {siteId}, insertMany - FAILED, Err: {err}'.format(*

*siteId=siteId,*

*err=e*

*))*

*return -1*

*# end try*

*# end insertMany*

```

You will notice the above 2 are very very similar. So, with a little bit of rework we collapsed the 2 functions into the one (*insertMongo*), all by adding mode as input variable and using an if block.

**Function:** *insertMongo*:

```

*def insertMongo(mongodb\_collection, siteId, mode, payload, logger):*

*try:*

*if mode == 1:*

*mode = "insertOne"*

*result = mongodb\_collection.insert\_one(payload)*

*else:*

*mode = "insertMany"*

*result = mongodb\_collection.insert\_many(payload)*

*# end if*

*return result*

*except Exception as e:*

*logger.error('connection.insertMongo - Site ID {siteId}, mode {mode} - FAILED, Err: {err}'.format(*

*siteId=siteId,*

*mode=mode,*

*err=e*

*))*

*return -1*

*# end try*

*# end insertMongo*

```

In *app4c*, we want to add the ability to write the same payload to a text file, each measurement as a single line, but wait… we did not stop there… In *app4a* and *app4b* we called *app4a/collection.py* we called either *insertOne* or *insertMany*, we then changed that to *insertMongo* in *app4b*. In app4c we changed further.

As we’re now wanting to write to either a text file or a MongoDB collection, and we really don’t want to have code in *simulate.py/run\_simulation* that’s specific to each. What was added was a generic call in collection.py which is called from *run\_simulation*, the collection.py code then checks what the *data\_persistence* is set to and then locally calls the relevant function.

So, we start by adding *app4c/connection.py -> createConnectionToStore*

In here we either call *createMongoConnection* to open a connection to our MongoDB or we open a new file by calling *createFileConnection* and then pass back object to *run\_simulation*. One of these 2 objects are then pushed into the below calls as the data persistence store to be used.

**Function:** *createConnectionToStore*

```

*def createConnectionToStore(config\_params, site, mode, logger):*

*if site["data\_persistence"] == 1:*

*return createFileConnection(config\_params, site["siteId"], mode, logger)*

*else:*

*return createMongoConnection(config\_params, site["siteId"], mode, logger)*

*# end if*

*# end createConnectionToStore*

```

**Function:** *createFileConnection*

```

*def createFileConnection(config\_params, siteId, mode, logger):*

*file = None*

*try:*

*filename = config\_params["FILEROOT"] + "\_" + str(siteId) + ".json"*

*if filename != "":*

*file = open(filename, 'a') # Open the file in append mode*

*if file != None:*

*logger.debug('connection.createFileConnection - Site ID {siteId}, mode {mode} - Filename {filename} OPENED'.format(*

*siteId=siteId,*

*mode=mode,*

*filename=filename*

*))*

*return file*

*else:*

*return -1*

*#end if*

*# end if*

*except IOError as err:*

*logger.critical('connection.createFileConnection - Site ID {siteId}, mode {mode} - FAILED Err: {err} '.format(*

*siteId=siteId,*

*mode=mode,*

*err=err*

*))*

*return -1*

*# end try*

*# end createFileConnection*

```

**Function:** *closeMongoConnection*

```

*def createMongoConnection(config\_params, siteId, logger):*

*try:*

*if config\_params["MONGO\_USERNAME"] != "":*

*config\_params["MONGO\_URI"] = f'{config\_params["MONGO\_ROOT"]}://{config\_params["MONGO\_USERNAME"]}:{config\_params["MONGO\_PASSWORD"]}@{config\_params["MONGO\_HOST"]}:{int(config\_params["MONGO\_PORT"])}/?{config\_params["MONGO\_DIRECT"]}'*

*else:*

*config\_params["MONGO\_URI"] = f'{config\_params["MONGO\_ROOT"]}://{config\_params["MONGO\_HOST"]}:{int(config\_params["MONGO\_PORT"])}/?{config\_params["MONGO\_DIRECT"]}'*

*logger.debug('connection.createMongoConnection - Site ID {siteId} - URI: {uri} '.format(*

*siteId=siteId,*

*uri=config\_params["MONGO\_URI"]*

*))*

*try:*

*myclient = pymongo.MongoClient(config\_params["MONGO\_URI"])*

*myclient.server\_info() # force connection on a request as the*

*# connect=True parameter of MongoClient seems*

*# to be useless here*

*except pymongo.errors.ServerSelectionTimeoutError as err:*

*logger.critical('connection.createMongoConnection - Site ID {siteId} - FAILED Err: {err} '.format(*

*siteId=siteId,*

*err=err*

*))*

*return -1*

*# end try*

*mydb = myclient[config\_params["MONGO\_DATASTORE"]]*

*my\_collection = mydb[config\_params["MONGO\_COLLECTION"]]*

*logger.info('connection.createMongoConnection - Site ID {siteId} - CONNECTED'.format(*

*siteId=siteId*

*))*

*return my\_collection*

*except Exception as e:*

*logger.critical('connection.createMongoConnection - Site ID {siteId} - FAILED, Err: {err} '.format(*

*siteId=siteId,*

*err=e*

*))*

*return -1*

*# end try*

*# end createMongoConnection*

```

Next, we introduce app4c/connection.py -> savePayloadToStore

**Function:** *savePayloadToStore*

```

*def savePayloadToStore(connection, site, mode, payload, logger):*

*if site["data\_persistence"] == 1:*

*return writeToFile(connection, site["siteId"], mode, payload, logger)*

*else:*

*return insertMongo(connection, site["siteId"], mode, payload, logger)*

*# end if*

*# end savePayloadToStore*

*```*

*This will call writeToFile or insertMongo, using the above created data persistence object.*

*Function: writeToFile:*

*```*

*def writeToFile(file, siteId, mode, payload, logger):*

*try:*

*if file:*

*if mode == 0:*

*mode = "writeOne"*

*# Convert the payload dictionary to a JSON string*

*payload\_json = json.dumps(payload)*

*file.write(payload\_json + '\n') # Add a newline at the end*

*else:*

*mode = "writeMany"*

*for record in payload:*

*# Convert each payload to a JSON string and write it to the file*

*payload\_json = json.dumps(record)*

*file.write(payload\_json + '\n') # Write each payload on a new line*

*# end if*

*return 1*

*except IOError as e:*

*logger.error('connection.writeToFile - Site ID {siteId}, mode {mode} - FAILED, Err: {err}'.format(*

*siteId=siteId,*

*mode=mode,*

*err=e*

*))*

*return -1*

*# end try*

*# end writeToFile*

```

Function: *insertMongo*

We covered this above already in *app4b/*

We lastly also have a *closeConnectionToStore* which in turn call *closeFileConnection*

or *closeMongoConnection*.

Function: closeConnectionToStore

```

```

Function: *closeFileConnection*

```

*def closeFileConnection(file, siteId, mode, logger):*

*if file:*

*try:*

*file.close()*

*except IOError as e:*

*logger.error('connection.closeFileConnection - Site ID {siteId}, mode {mode} - FAILED, Err: {err}'.format(*

*siteId=siteId,*

*mode=mode,*

*err=e*

*))*

*# end try*

*# endif*

*# end close\_file*

```

Function: closeMongoConnection

```

*def closeMongoConnection(client, siteId, mode, logger):*

*try:*

*client.close()*

*except Exception as e:*

*logger.error('connection.closeMongoConnection - Site ID {siteId}, mode {mode} - FAILED, Err: {err}'.format(*

*siteId=siteId,*

*mode=mode,*

*err=e*

*))*

*return -1*

*# end try*

*# end closeMongoConnection*

```

In *app4d*, we add code to check if the database and the collection already exist, if not we create it.

Basically, if the database exists, we simply create a time series specific collection as follows.

**Function**: *create\_mongo\_ts\_collection*

```

*def create\_mongo\_ts\_collection(db, config\_params, siteId, logger):*

*try:*

*time\_series\_options = {*

*"timeField": config\_params["TIMESTAMP\_FIELD"], # Field that stores the timestamp*

*"metaField": config\_params["METADATA\_FIELD"], # Optional: Field to store metadata*

*"granularity": config\_params["RETENSION\_LEVEL"] # Can be 'seconds', 'minutes', or 'hours'*

*}*

*# Add code to make Collection TS specific.*

*db.create\_collection(config\_params["MONGO\_COLLECTION"], timeseries=time\_series\_options)*

*logger.debug('connection.create\_ts\_collection - Site ID {siteId}, Collection {collection}, CREATED'.format(*

*siteId=siteId,*

*collection=config\_params["MONGO\_COLLECTION"]*

*))*

*return 0*

*except Exception as e:*

*logger.critical('connection.create\_mongo\_ts\_collection - Site ID {siteId}, Collection {collection}, Create FAILED, Err: {err}'.format(*

*siteId=siteId,*

*collection=config\_params["MONGO\_COLLECTION"],*

*err=e*

*))*

*return -1*

*# end try*

*# end create\_ts\_collection*

```

If the database does not already exist, we still execute the above collection create, this will implicitly create the database.

And that’s section 4, another long section, but at the same time not. Some more MVP examples and then optimization as we go along.

For now, this will be it… See you in part 5, this will be our final “part”

Good luck, this is all fraught with rabbit holes, as always, so many and you can disappear so easily… but then that’s ½ the fun.



**About Me**

I’m a techie, a technologist, always curious, love data, have for as long as I can remember always worked with data in one form or the other, Database admin, Database product lead, data platforms architect, infrastructure architect hosting databases, backing it up, optimizing performance, accessing it. Data data data… it makes the world go round.

In recent years, pivoted into a more generic Technology Architect role, capable of full stack architecture.

[George Leonard](https://www.linkedin.com/in/george-leonard-945b502/)

[georgelza@gmail.com](mailto:georgelza@gmail.com)

<https://www.linkedin.com/in/george-leonard-945b502/>

<https://www.mongodb.com/community/creators/>

<https://medium.com/@georgelza>

# Generating a realistic IoT data stream using Python and storing into MongoDB Timeseries Collection. – Part 5

A Tutorial (of sort) where we take the reader through the creation of an IoT timeseries data stream using Python and store the data into an MongoDB Timeseries Collection.

(22 November 2024)

**Overview**

So, at this point we are done, we’re now polishing…

In this last section I’m going to change 1 function, logger and then add code to accomplish some timing/instrumentation, lastly, we will do a short recap…

See app5/README.md

**Firstly Objective**.

You would have noticed we output *siteId* when writing to the console and to the file (this was because we included *siteId* in the message body), well, this value was written for every line to in the file, even though the filename also already the file name. This extra tag and value per every line quickly adds up, so what we will do now (with hindsight of the above) is to modify the file logger function in app5/utils.py

**Function**: logger

Previously we only had one formatter line, we split this into separate formatters, namely *console\_formatter* and *file\_formatter*.

When we now call the logger function to create a logger it will create our console logger with the *siteId* included (this is due to the fact that when we run multiple sites we have many sites outputting messages to the console and we want to know which line pertains to white site), but when we write logging messages to file we don’t require this.

```

*def logger(filename, console\_debuglevel, file\_debuglevel):*

*logger = logging.getLogger(\_\_name\_\_)*

*logger.setLevel(logging.DEBUG)*

*# Create a formatter*

*console\_formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(processName)s - %(message)s')*

*# create console handler*

*ch = logging.StreamHandler()*

*# Set file log level*

*if console\_debuglevel == 0:*

*ch.setLevel(logging.DEBUG)*

*elif console\_debuglevel == 1:*

*ch.setLevel(logging.INFO)*

*elif console\_debuglevel == 2:*

*ch.setLevel(logging.WARNING)*

*elif console\_debuglevel == 3:*

*ch.setLevel(logging.ERROR)*

*elif console\_debuglevel == 4:*

*ch.setLevel(logging.CRITICAL)*

*ch.setFormatter(console\_formatter)*

*logger.addHandler(ch)*

*# Create a formatter*

*file\_formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')*

*fh = logging.FileHandler(filename)*

*# Set file log level*

*if file\_debuglevel == 0:*

*fh.setLevel(logging.DEBUG)*

*elif file\_debuglevel == 1:*

*fh.setLevel(logging.INFO)*

*elif file\_debuglevel == 2:*

*fh.setLevel(logging.WARNING)*

*elif file\_debuglevel == 3:*

*fh.setLevel(logging.ERROR)*

*elif file\_debuglevel == 4:*

*fh.setLevel(logging.CRITICAL)*

*else:*

*fh.setLevel(logging.INFO) # Default log level if undefined*

*fh.setFormatter(file\_formatter)*

*logger.addHandler(fh)*

*return logger*

*# end logger*

```

Next up, to use the above, see *app5/main.py line 90.* You will notice we now add *name=siteLabel* to our call to *multiprocessing.Process*

This is *name* variable is picked up by the logger and interpreted as *processName*.

Function: *main* – extract.

```

*try:*

*for site in my\_sites:*

*siteLabel = "SiteId: " + str(site["siteId"])*

*logger.info("Calling simulate.run\_simulation for {siteLabel}".format(*

*siteLabel=siteLabel*

*))*

*# Create a process for each site*

*p = multiprocessing.Process(target=simulate.run\_simulation, name=siteLabel, args=(site, current\_time, config\_params))*

*processes.append(p)*

*p.start()*

*# end for*

*for p in processes:*

*p.join()*

*# end for*

```

**Second Objective:**

Instrumentation, or well, in our case it’s more just some simple timing to be able to report how long a step took, and the throughput.

See *app5/simulate.py line 238*

```

*# Historical phase*

*if "historic\_data\_start\_datetime" in site and site["historic\_data\_start\_datetime"]:*

*step1starttime = datetime.now()*

*step1start = perf\_counter()*

```

Firstly, we add the above 2 variables (*step1starttime & step1start*).

We then do whatever we want to do, this includes a counter of how many loops, or records are generated, used below as *historical\_record\_counter.*

And then call:

```

*step1endtime = datetime.now()*

*step1end = perf\_counter()*

*step1time = step1end - step1start*

*histrate = str( round(historical\_record\_counter/step1time, 2))*

```

We then report, print to file/console these values using the below code:

This allows us to see the start time, end time, run time, how many records we processed and the records/second / throughput / rate.

See *app5/simulate.py* line 441

```

*logger.info("simulate.run\_simulation - Historical Record Process Stats - St: {start}, Et: {end}, Rt: {runtime}, Recs: {historical\_record\_counter} docs, Rate: {histrate} docs/s".format(*

*start = str(step1starttime.strftime("%Y-%m-%d %H:%M:%S.%f")),*

*end = str(step1endtime.strftime("%Y-%m-%d %H:%M:%S.%f")),*

*runtime = str(round(step1time, 4)),*

*historical\_record\_counter = historical\_record\_counter,*

*histrate = histrate*

*))*

```

**In Summary**

Well, that’s it… That’s our entire application. From nothing to I hope something usefull.

I hope my process makes sense, I hope the various version’s shown by starting with a MVP version A, then following up with a version B & C etc. as we improve/optimize/expand functionality made sense.

The intend here was to show a realistic IoT data generation and pushing the generated data into a MongoDB Atlas TimeSeries specific collection.

In the real world, we’re more likely to have our data coming out of the sensor, collected by a PLC controller (our app) onto say a MQTT broker, which will be configured to sink that payload onto a Kafka Cluster onto a Topic. We will then in turn configure a Kafka sink Connector to save/insert the payloads into our collection.

By starting with the above we can generate our data, store that in our collection, then build our analytics, and Mongo Charts. Once we know that’s working, we can modify our collection functions to publish to a MQTT broker or Kafka Topic and see the larger flow in action.

In closing, my original idea actually started with wanting to explore Machine Learning (ML) on sensor data, on an IoT data stream. However, to do that I require realistic data, with realistic sensor readings, with normal measurements and then measurements that deviate from the norm seen…

Thus, enabling us to see if our ML model find our the little easter eggs… but that’s for next time.

Hope this was useful. Till next time.



**About Me**

I’m a techie, a technologist, always curious, love data, have for as long as I can remember always worked with data in one form or the other, Database admin, Database product lead, data platforms architect, infrastructure architect hosting databases, backing it up, optimizing performance, accessing it. Data data data… it makes the world go round.

In recent years, pivoted into a more generic Technology Architect role, capable of full stack architecture.

[George Leonard](https://www.linkedin.com/in/george-leonard-945b502/)

[georgelza@gmail.com](mailto:georgelza@gmail.com)

<https://www.linkedin.com/in/george-leonard-945b502/>

<https://www.mongodb.com/community/creators/>

<https://medium.com/@georgelza>