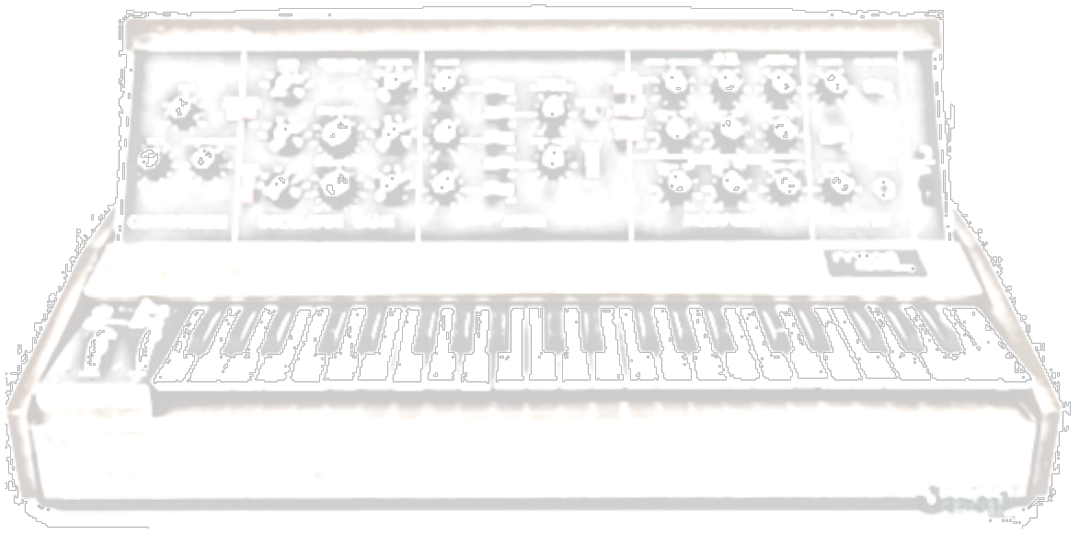
Synth

by DevicePilot™



1. Introduction

Connected devices are being deployed in every market sector in increasing numbers. A significant challenge of the Internet of Things is *how to test IoT services?* Services must typically be brought-up ahead of the widespread availability of the devices they will serve, yet testing a service requires an estate of devices to test it with – a chicken-and-egg problem. And as a proposition grows in the market, operators will wish to continually test services at a scale perhaps an order-of-magnitude greater than the number of devices currently in use, to find bottlenecks and weaknesses in the service.

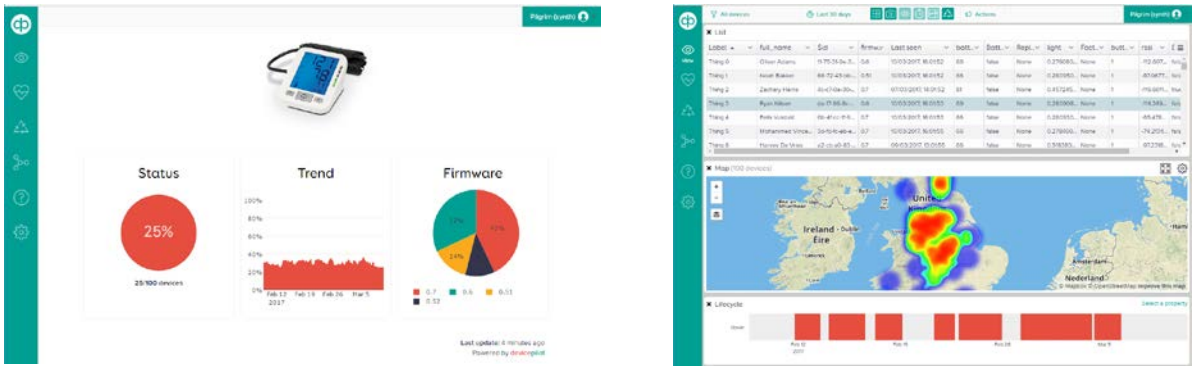
- Manually testing a service with physical devices is far too slow and error-prone and simply doesn't fit within a modern CI/CD/TDD framework
- Maintaining an estate of physical devices for automated testing has some merits – but is impractical at a scale of more than 100 or so devices, and doesn't scale well across many developers
- Therefore there is a need for a tool capable of *synthesising* virtual devices, at scale. Such emulated devices then comprise a virtual device 'estate' which can be thrown at a service to prove that it works correctly at any scale

Synth is such a tool. It has the unusual capability to create data both in batch and interactive modes, and seamlessly to morph from one to the other, making it useful for several purposes including:

- Generating realistic device datasets: both static and historical time-series
- Dynamic testing of services
- Demonstrating IoT services interactively, having first generated a plausible history to this moment, without requiring access to real user data which is often subject to data confidentiality.

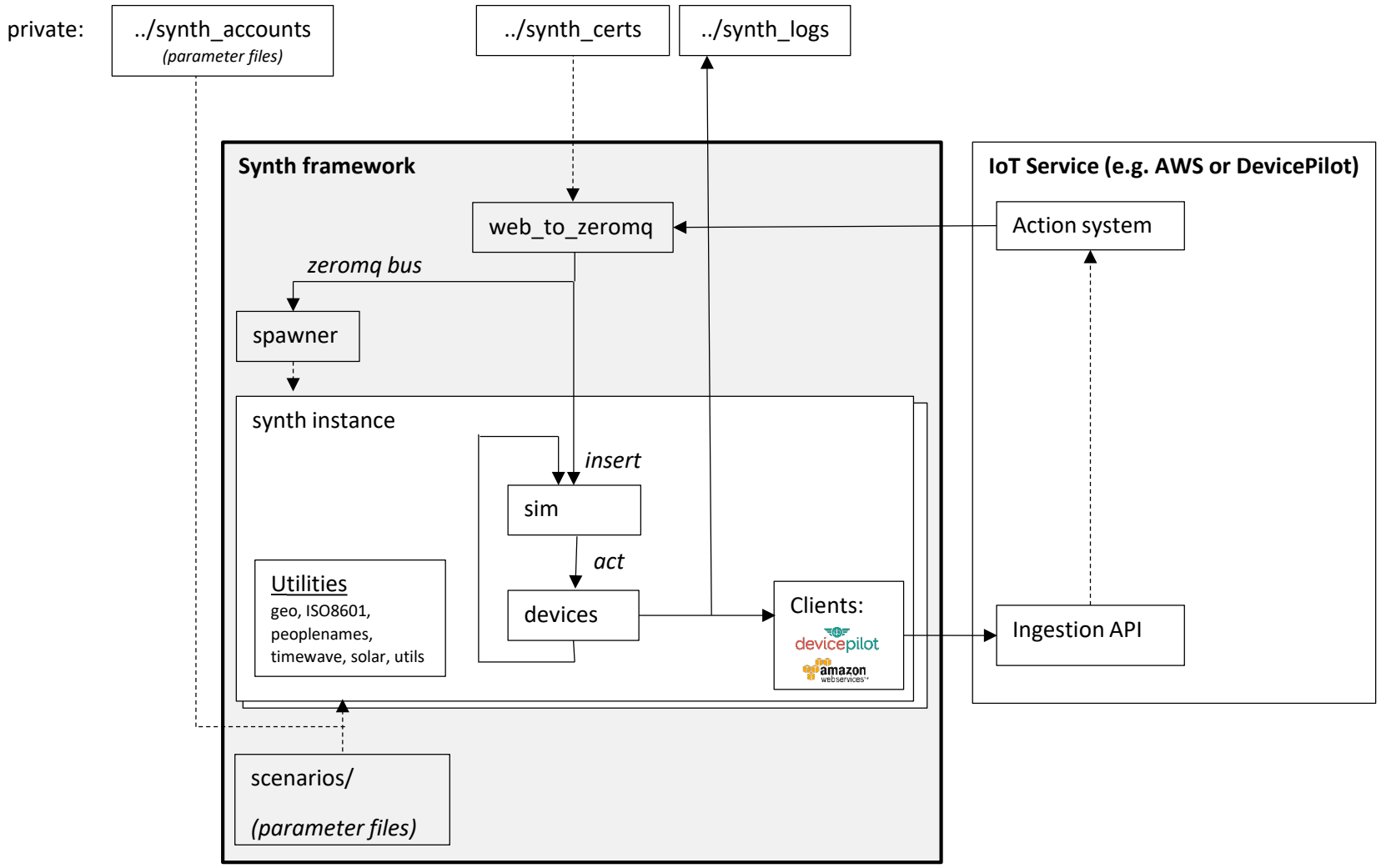
Synth was developed as a tool to help test and demonstrate DevicePilot, the cloud service for managing IoT devices, but it's independent of DevicePilot and can be used with any framework - it comes with an AWS-IoT client too, for example.

Synth was released under the permissive open-source MIT license in 2017.



Examples of synthesised device data (shown on DevicePilot UI)

2. Architecture



3. Synth framework

The architecture is pure Python 2.7 so should run on most OSes. A Linux server is its natural home but it has been tested on Windows too. Installation of all required packages can be done with apt-get and pip.

The framework consists of three processes communicating within a host server via ZeroMQ:

- A Web server (web_to_zeromq)
- A process for spawning new Synth instances (spawner.py)
- Zero or more running Synth instances (synth.py)

The `runSynth` script runs exactly one instance of a given Synth (killing any existing one first), logging its output to ../logs/*.out

The `runWeb` and `runSpawner` scripts do the same for the other processes.

synth.py

A new Synth instance is started by running

```
python synth.py {parameters|parameter_files}
```

Depending on the parameter files supplied, an instance may exit once its job is done, or keep running to interactively simulate devices in real time, perhaps forever. Parameters are supplied as arguments. Any argument containing an "=" sign is taken to be a parameter assignment. Other parameters are taken to be the name of a JSON file in the `data/` directory which contains a set of parameters.

`synth.py` also connects various callbacks between lower-level modules so they don't need to know about each other.

sim.py

Maintains a simulation queue, and the concept of a current simulation time, with which log messages are stamped. The `injectEvent()` function inserts callbacks into the queue for later execution. Periodically calling the `nextEvent()` function from a main loop then executes each callback when its time has come. Thread-safe locking is used so that `injectEvent()` may be called asynchronously. If started with a historical time, sim detects when it hits real-time and automatically drops into an interactive mode.

devices.py

A device behaviour simulator which maintains a list of device instances. Devices can call `sim.injectEvent()` to create future events on them, so for example a battery method might decrement the battery property by 1% and then call `sim.injectEvent()` to run again in a month. Whenever device properties are changed, a callback posts device updates into an external service such as DevicePilot or AWS.

Some modelling of realistic comms failure is included.

Todo: In future `devices.py` should probably be split into a generic device class, which specific device definitions could inherit. Support multiple device types simultaneously.

web_to_zeromq.py

A standalone Flask Web server which accepts incoming Web requests and translates them into messages on a ZeroMQ bus. Its principle use is to inject external events asynchronously into any Synth instance (identified by its `instance_name` parameter). Needs to be run with sufficient privileges to be able to open SSL port 440.

spawner.py

A standalone process which creates new Synth instances on demand when it receives instruction to do so from the Web server. In combination with web_to_zeromq means, new Synth instances can be spawned and monitored with a simple Web request.

Clients

Clients are IoT systems to which Synth can post its simulated device data. Synth already contains two such clients and it would be relatively easy to add more [todo: add extensible client library]

aws/

A basic client for AWS-IoT, updating the Device Shadow "reported" state. Has some limitations:

- Not very efficient as it uses the HTTP interface to AWS rather than MQTT or Websockets.
- Cannot yet load existing device state

If no AWS-IoT account is defined in the parameters, it can be configured using the `aws configure` command of the AWS CLI.

devicepilot/

A client for DevicePilot. For efficiency, implements a queue so that messages are batch-posted. Different clients can be targeted on different runs according to the `devicepilot_*` parameters.

Utilities

There are various utilities including:

- **geo:** Can place devices realistically anywhere on the Earth according to typical technology density (so they appear close to roads, cities etc.). Can place devices within a circle of given radius, as specified by placenames.
- **ISO8601:** Reads and writes time in the format YYYY-MM-DDTHH:MM:SSZ with timezones, relying on pytz for this
- **peoplenames:** Generates somewhat-plausible names for device owners
- **timewave:** Creates sequences over human time, e.g. "every Tuesday".
- **solar:** Understands the light level at any time of day anywhere on the Earth. Todo: clouds!

A word on Security

Synth assumes it is running on a secured server, e.g. a Linux box on AWS. So for example the three Synth processes which intercommunicate via ZeroMQ do so without security measures (a miscreant with access into the server could easily spoof messages between them).

Synth attempts to keep the perimeter of the server secure. All inbound and outbound communications are over HTTPS, incoming requests must present keys, and inbound requests are sanity-checked to avoid injection attacks.

4. Directories

Some of the supporting directories are located in the directory *above* the sourcecode, so that they don't accidentally get checked-in and can hold your personal/private data.

Parameter files

Any command-line argument passed to Synth which does not include the "=" character it is taken as the name of a JSON parameter file to read. Parameter files contain a JSON set of `{ parameter : value, parameter : value }` pairs. Python-style `# comments` may also be used. See the Parameters section for possible values.

All parameter files have the same syntax and can contain any parameters, but by convention we distinguish two types of file by naming and location:

- Account files (named by convention "On*" as in "OnServiceAccountUser") which provide keys to enable communication with a host system such as DevicePilot, in both directions. These also define the instance of DevicePilot (by URL) and the user key (thus identifying the DevicePilot account). These files will typically be stored in the private ../accounts folder. Account files typically contain only the following parameters:
  - `instance_name`
  - `web_key`
  - `devicepilot_api` `-or-` `aws_access_key_id`
  - `devicepilot_key` `aws_secret_access_key`
  - `aws_region`
- Scenario files. These contain parameters to set up particular simulation scenarios, e.g. "Germany10devices", and are stored in the scenarios folder.

scenarios/

The first place that Synth looks for parameter files. Put your simulation scenarios here.

../synth_accounts/

The second place that Synth looks for parameter files. Put your account credentials here in files named "On*".

../logs/

Files emitted by Synth to help in diagnosis. They are named by instance_name and always append.

- *.evt files are a complete record of the simulator output, i.e. a record of every parameter update from every device. Timestamps are simulation-time.
- *.out files are the stdout+stderr output from a Synth instance

../certs/

Contains private keys and certificates:

- ssl.crt and ssl.csl: the two SSL certificate files necessary to enable Flask to securely accept and make HTTPS:// connections:
- webkey: a single line of characters which for security must be presented as an argument during incoming GET requests to the Synth webserver "/" path (see Web endpoints)
- googlemapskey: a single line of characters which will be presented to the Google Maps API when a "placename to (long,lat)' lookup is done (only if the area_centre/radius parameters are used)

5. Web endpoints

Synth simulates not just device output, but also device input too. An external service can "prod" Synth to generate asynchronous events on devices, via its Web server, at scale. These events are synchronised via the event queue in `sim.py` and then handled in `device.py`.

Some general URL queries allow the status of the Synth framework to be checked, and new simulations to be spawned. And a POST interface allows events to be sent to specific devices running in specific Synth instances.

`web_to_zeromq.py` hosts a Flask web server at HTTPS port 443.

Valid requests are:

GET /?<magickey>

Return a basic page listing all running Synth processes and free memory. For security this must be accompanied by a magic key matching the contents of the file ../synth_certs/magickey

GET /spawn?devicepilot_key=XXX&devicepilot_api=staging

Spawn a new instance of Synth, with these two specific parameters set. The UserDemo scenario is run. The instance_name is set to be "devicepilot_key=XXX" since that is assumed to be unique.

GET /is_running?devicepilot_key=XXX&devicepilot_api=staging

Find whether a specific instance of Synth (identified by its key) is still running. This returns:

```
{ "active" : true }   # if it is still running
{ "active" : false }  # if it has finished
```

POST /event

This causes an event to be asynchronously generated for a specific device on a specific Synth instance.

The header of the web request must contain the following:

```
Key : <web_key parameter>
Instancename : <instance_name parameter>
```

The body of the web request must contain a JSON set include the following elements:

```
"deviceId" : "theid"                (matching a device's $id field)
"eventName" :   "replace_battery" |
                "upgradeFirmware" |
                "factoryReset"
"arg" : "0.7"                       (optional – used only by upgradeFirmware)
```

DevicePilot Actions

If defining a Webhook Action in DevicePilot to create a Synth event, the device ID will be automatically filled-out if you define it as `{device.$id},` resulting in an action specification which looks something like this:

Upgrade firmware to 0.7	method: POST
	url: https://synthservice. com/event
	headers: { "Key":"mywebkey", "Instancename" : "OnProductionAccountUser" }
	body: { "deviceId" : "{device.$id}", "eventName" : "upgadeFirmware", "arg":"0.7"}

6. Parameters

The parameters for every Synth run are supplied either directly on the command-line, or in JSON files named on the command-line. Thus with a sensible convention on file naming, one can see which scenarios are running by doing:

```
ps uax | grep python | grep –v grep
```

Below, each parameter name is followed by one or more possible example values. The parameter names are shown here in a logical order of introduction, not sorted alphabetically.

Account parameters

By convention the following parameters are defined in "On*" files, as they are the only parameters which change according to which IoT platform (and account on that platform) Synth is interacting with.

"instance_name" : "OnProductionSynthPilgrim"

An arbitrary unique name used to identify this Synth instance (in log files, and also to determine which instance incoming web requests should be sent to)

"on_aws" : <anything>

[not recommended] If defined, causes device property updates to be sent to AWS-IoT, using whatever account has been previously configured locally on the server running Synth, using `aws configure`

"aws_access_key_id" : "<AWSaccesskey>"
"aws_secret_access_key" : "<AWSsecretkey>"
"aws_region" : "us-west-1"

These parameters define a specific AWS account to send device property updates into. This allows you to configure various "OnAWSxxx" files in your ../synth_accounts folder, and set instances to run on each independently. To create these parameters:

1. Log on to your AWS account
2. Click IAM / Users / <username>
3. In the Security tab, click "Create Access Key"

"devicepilot_api" : "https://api.devicepilot.com"
"devicepilot_key" : "12345678901234567890123456789012"

If defined, specifies the DevicePilot API endpoint to which to send outgoing device property update, and the key that the DevicePilot client uses to authenticate when sending device property updates. To find your key:

1. Log on to your DevicePilot account
2. Click Settings / My User and find your key in the API Key section

"web_key" : "AveryLongAndUniqueString"

A key that Synth uses to authenticate incoming web requests, when receiving spontaneous external events

Scenario parameters

These determine what happens during a simulation run.

```
"initial_action" :      "deleteExisting"  # Delete all devices
                        "deleteDemo"      # Delete only demo devices
"loadExisting"    # Load existing devices
```

This is what happens before the simulation run begins. If you want a clean sweep, use `deleteExisting` (which is very fast on DevicePilot). But if there might be other data and you want to be sure you're only deleting demo devices previously created by Synth, use `deleteDemo`. Or if you want Synth to load its internal device state *from* the external service, i.e. to resume a simulation, then specify `loadExisting`.

```
"device_count" : 10
```

The number of devices to create and simulate.

```
"start_time" :    "now"
                  "2017-01-01T00:00:00"
                  "-120"
```

The simulation time at the start of the simulation run. If it is "now" then simulation starts at the current realtime, suitable for an interactive simulation which doesn't need to create any historical data. Or it can be an absolute datetime in ISO8601 format. Or if it's a negative number then that is interpreted as "days before now", allowing a certain length of historical record to be created before now.

```
"end_time" :      null
                  "2017-02-01T00:00:00"
                  "now"
```

The simulation time at which the simulation ends. If `null` then it never ends. If an ISO8601 datetime then it will end at that time. And if "now" then it will end when simulation time reaches the current real time.

Except in this last case, when simulation time catches-up with real time, Synth automatically drops into a realtime interactive mode where it will wait as necessary to ensure that simulation time never gets ahead of realtime.

```
"install_timespan" : 2592000
```

The time, in seconds, over which all the devices are installed. If zero then all the devices are created at the start_time, but it is often more realistic to install them gradually.

```
"queue_criterion" :     "interactive"
                        "messages"
                        "time"
"queue_limit" : 1
```

The DevicePilot client queues synthesised device data and then posts it into DevicePilot in batches for efficiency. This parameter sets the criterion for when the queue is flushed, and the queue_limit parameter below provides the argument for whichever criterion is chosen.

- `"interactive"` means that the queue will be flushed at least every `queue_limit` seconds.
- `"messages"` means that the queue will be flushed when `queue_limit` messages are enqueued.
- `"time"` means that the time-stamp of the messages in the queue will never span more than `queue_limit` seconds.

```
"area_centre" : "Berlin, Germany"
"area_radius" : "Hamburg, Germany"
```

If these parameters are not specified, then devices are placed randomly over the entire globe, scattered according to typical technology-density. If they are specified then they are taken to be the address of the centre and radius respectively of a circle within which all devices will be scattered.

```
"battery_life_mu" : 300
"battery_life_sigma" : 60
"battery_replace" : true
```

These determine the battery-life of devices. Battery-life is expressed as a percentage and once it reaches zero the device stops working. Batteries can be replaced with the "replace_battery" web request. These parameters determine the average (mu) and standard deviation (sigma) of battery-life. If no randomness is required, set `battery_life_sigma` to 0. If the `battery_replace` parameters is specified, then batteries are automatically replaced when they become depleted.

```
"comms_reliability" : 1.0
                      "[[0,0.20],[30,0.65],[60,0.50],[90,0.95],[120,0.98]]"
```

Sets the reliability of device communication. If 1.0 then it's completely reliable, if it's 0.0 then messages never get through. If it's a string then it's interpreted as a trajectory specification of the form `"[[day,fraction],[day,fraction],...]"` – with linear interpolation. At present the period of communications going up or down is fixed at 0.5 days.

```
"web_response_min" : 3
"web_response_max" : 10
```

The response of Synth to asynchronous incoming web events (such as a request to change a device battery) can be delayed by an interval (in seconds) which is randomised between these two parameters, to provide a more realistic response for demo purposes.

```
"setup_demo_filters" : false
```

This DevicePilot-specific feature sets-up specific filters on a DevicePilot account, for demo-purposes.

7. Example scenarios

Interactive Demo

```
{
    "initial_action" : "deleteExisting",    # "deleteExisting" deletes all existing devices from external service
    "device_count" : 10,
    "start_time" : "now",
    "end_time" : null,
    "install_timespan" : 60,
    "queue_criterion" : "interactive",
    "queue_limit" : 1,
    "battery_life_mu" : 300,  # Extremely short battery life of 5 minutes – good for demonstrating battery replacement
    "battery_life_sigma" : 60
}
```

Interactive Demo with history

```
{
    "initial_action" : "deleteExisting",
    "device_count" : 100,
    "start_time" : "-120",           # -ve number "-N" means "N days before now"
    "end_time" : null,               # Never terminate (enters interactive mode when it catches-up with now)
    "install_timespan" : 60,         # How many seconds it takes at start of simulation to install all devices
    "queue_criterion" : "messages",  # Send updates to DevicePilot/AWS every N "messages"
    "queue_limit" : 1000,            # N for above
    "area_centre" : "Manchester, UK",# Centre of a circle of possible positions
    "area_radius" : "London, UK",    # Edge of circle
    "battery_life_mu" : 31536000,    # Average battery life of a year
    "battery_life_sigma" : 0,        # Variation in battery life
    "comms_reliability" : "[[0,0.20],[30,0.65],[60,0.50],[90,0.95],[120,0.99]]"    # A trajectory specification "[[day,fraction],…]
}
```

Historical only

```
{
    "initial_action" : "deleteExisting",
    "device_count" : 100,
    "start_time" : "-120",                  # Generate 4 months of data up to now
    "end_time" : "now",                     # Then quit
```

```
}
```

8. Example account files

Example sets of parameters, each of which would be stored in a file in your ../synth_accounts directory with a filename like “OnDevicePilot”

DevicePilot account

```
{
      "instance_name" : "OnProductionSynthPilgrim",
      "web_key" : "12323123123123123",
      "devicepilot_api" : "https://api.devicepilot.com",
      "devicepilot_key" : "12345678901234567890123456789012"
}
```

AWS account

```
{
      "instance_name" : "OnAWS",
      "web_key" : "1234567",
      "aws_access_key_id" : "ABCDEFGHIHJKLMNOPQRT",
      "aws_secret_access_key" : "AbCDeF+GHIjkLMN2OP00qrstUvw/AAbCDeFG+0aB",
      "aws_region" : "us-west-2"
}
```

9. Getting started

Initially try running Synth from the command-line, e.g.:

```
python synth.py OnAWS World100Live
```

To get this to work you'll have to:

A. Install required Python libraries (e.g. `pytz` and `requests`)
B. Create the following files (see examples in the previous section):
   - a scenario file called `World100Live` in the `scenarios/` directory
   - an account definition called `OnAWS` in your `../synth_accounts` directory.

Once it's working on the command-line, you can use:

```
./runSynth OnAWS World100Live
```

to ensure that any existing instance is killed-off first, and log the output

10. Synth in use

See Synth in use (without having to set it up yourself) by signing-up for a DevicePilot account and hitting the "Create demo data" button.