2019-05-29 Rob documents how notifications may work and asks for reviews:
existing features in blue
new features in black

2019-05-30 Define API and classes, update system diagram and link here.
Updated v5 architecture diagram.
Updated v5 architecture description.

2019-06-05 Add future image processing class / service design.

Each PFC publishes a stream of messages:

- sensor values when they change (and images)
- event: recipe start
  - (modify the UI code that sends the message to the device so the message it saves into BQ is "<blah> *sent* to the device")
  - the ground truth is when the device publishes a message saying "I have started <name> recipe".
- event: recipe stopped (manually by a user, or implicitly by starting a new recipe)
- event: recipe finished (it ended itself by running to its programmed end)
- event: errors and alerts (in the future)

MQTT service parses all the messages from all devices:

- **all** messages written to BQ
- **sensor** data:
  - write to: datastore.DeviceData<device_ID>.<sensor name>
  - each <sensor name> property is a queue of the most recent 100 values
- **event** data:
  - Notifications.publish(device_ID: str, type: str, value: str)
- **image** data: (future)
  - ImageProcessor.publish(device_ID: str, type: str, value: str)
    - sends the public URL of the image

EDU UI

- Get the list of unacknowledged notifications:
  - notifications_list = Notifications.get_for_device(device_ID)
- Displays the unacknowledged notifications and allows the user to acknowledge them with an OK/Yes/Done button.
- When the user ACKs the notification, the UI will call:
  - Notifications.ack(notification_ID)

Notification service

Subscribes to the *notifications* topic and handles these message types:

- Notifications.parse(message) will handle these messages:

- Notifications.recipe_start
  - Scheduler.add(device_ID, Scheduler.check_fluid, 48)
  - Scheduler.add(device_ID, Scheduler.take_measurements, 24 * 7)
  - Runs.start(device_ID, value)

- Notifications.recipe_stop
  - Scheduler.remove_all(device_ID)
  - Runs.stop(device_ID)

- Notifications.recipe_end
  - Scheduler.remove_all(device_ID)
  - Scheduler.add(device_ID, Scheduler.harvest_plant)
  - Runs.stop(device_ID)

- for all messages received, call:
  - Scheduler.check(device_ID)
  - (in the context of the message processing callback)

- finally, do an iot.ack() of the message.

Notifications class

- data stored in datastore.DeviceData<device_ID>.notifications as a dict
  - queue of the most recent 100 notifications per device
  - notification_ID: str
  - message: str
  - created: str (TS in UTC)
  - acknowledged: str (TS in UTC)

- notification constants that we initially handle:
  - recipe_start
  - recipe_stop
  - recipe_end

- publish(device_ID: str, type: str, value: str) -> None
  - if the type is one of the Notifications constants:
    - publish a message with a value to the *notifications* topic.

- parse(data: Dict[str, str]) -> None
  - Parse the dict and handle valid messages.
  - See the description of the notification service for messages and actions.

- get_for_device(device_ID: str) -> List[Dict[str, str]]
  - returns a list of **unacknowledged notifications** dicts

- ack(device_ID: str, notification_ID: str) -> None
  - find in notification by ID and update the acknowledged timestamp to now()

- add(device_ID: str, notification_type: str, message: str) -> str
  - add a new notification for this device, set created TS to now()
  - return notification_ID

Scheduler class

- data stored in datastore.DeviceData<device_ID>.schedule as a dict
  - command: str <command>
  - timestamp: str <timestamp to run on>
  - repeat: int <number of hours, can be 0 for a one time command>
  - count: int <execution count>

- command constants that we initially handle:
  - check_fluid
    - message: str = 'Check your fluid level'
    - default_repeat_hours: int = 48
  - take_measurements
    - message: str = 'Record your plant measurements'
    - default_repeat_hours: int = 24
  - harvest_plant
    - message: str = 'Time to harvest your plant'
    - default_repeat_hours: int = 0
    - removes itself from the schedule once it has fired.

- add(device_ID: str, command: str, repeat_hours: int = 0) -> None
  - creates entry above, setting timestamp = now() + hours…, count = 0

- remove_all(device_ID: str) -> None
  - removes **all** commands for this device.

- check(device_ID: str) -> None
  - iterate the schedule entries for device_ID acting upon entries that have a timestamp <= now()

  - if a command has a repeat_hours value > 0, then update its timestamp when executing it.
  - update the count of times the command has been executed.
  - write notifications the UI will render
    - Notifications.add(..)
      - notification_ID: UUID generated when notification created
      - message: <yada>
      - timestamp: <TS in UTC>
      - acknowledged: <TS in UTC>
  - Handle init. and term. logic, such as:
    - if we are repeating the take_measurements command and count == 1, then set the repeat interval to 24 hours.

Runs class

- data stored in datastore.DeviceData<device_ID>.runs as a list of dicts
  - queue of the most recent 100 runs per device
  - start: str <timestamp in UTC>
  - end: str <timestamp in UTC>
  - value: str <name of recipe>

- get_all(device_ID: str) -> List[Dict[str, str]]
  - returns a list of dicts of the runs for this device as:
    - { start: str, end: str, value: str }
    - start may be None if a recipe has never been run.
    - end may be None if the run is in progress.

- get_latest(device_ID: str) -> Dict[str, str]
  - returns a dict of:
    - { start: str, end: str, value: str }
    - start may be None if a recipe has never been run.
    - end may be None if the run is in progress.

- start(device_ID: str, value: str)
  - start a new run for this device starting now.
  - push onto the queue:
    - { start: now(), end: None, value: value }

- stop(device_ID: str)
  - stop an existing run for this device, now.
  - if top item on the queue has a end == None
    - { start: TS, end: now() }

ImageProcessor class (future)

- publish(device_ID: str, type: str, value: str) -> None
  - publish an image_frob message, such as "update timeline"

ImageProcessor service (future)

- Subscribes to the image_process topic and processes messages.
- Also subscribes to the notifications topic to track recipe start/end (or uses the datatore?).
- First feature will be to start, update, end an image time lapse that matches the recipe timeline.
  - How to implement making a video from stills with delays? OpenCV?
  - Updates the time lapse video in a public gstorage bucket.
    - Filename something like <device_ID>_<recipe_name>_<recipe_start_TS>.mp4
  - Replaces the URL to the time lapse video in datastore.DeviceData<device_ID>.timelapse

Code organization

- Common classes, utils, etc will be here:
  - https://github.com/OpenAgricultureFoundation/cloud_common

- The Notification service gcloud app engine project will be here:
  - https://github.com/OpenAgricultureFoundation/notification-service