Smart EPU Open API

code style black

Thermo Scientific™ **Smart EPU Software** is a Cryo-EM platform designed to increase efficiency, productivity, and ease of use in Cryo-EM. It comprises **EPU**, image evaluation programs (**EQM** and **Embedded CryoSPARC Live**), **Smart Plugins** and **Athena**.

Smart EPU Software offers a robust solution for automated screening and data acquisition in single particle analysis using cryo-EM (see Figure 1). It provides on-the-fly monitoring of data collection sessions (**EPU** runs), through AI-assisted decision-making **Smart Plugins**, eliminating the need for manual intervention in what becomes a tedious, time-consuming task.

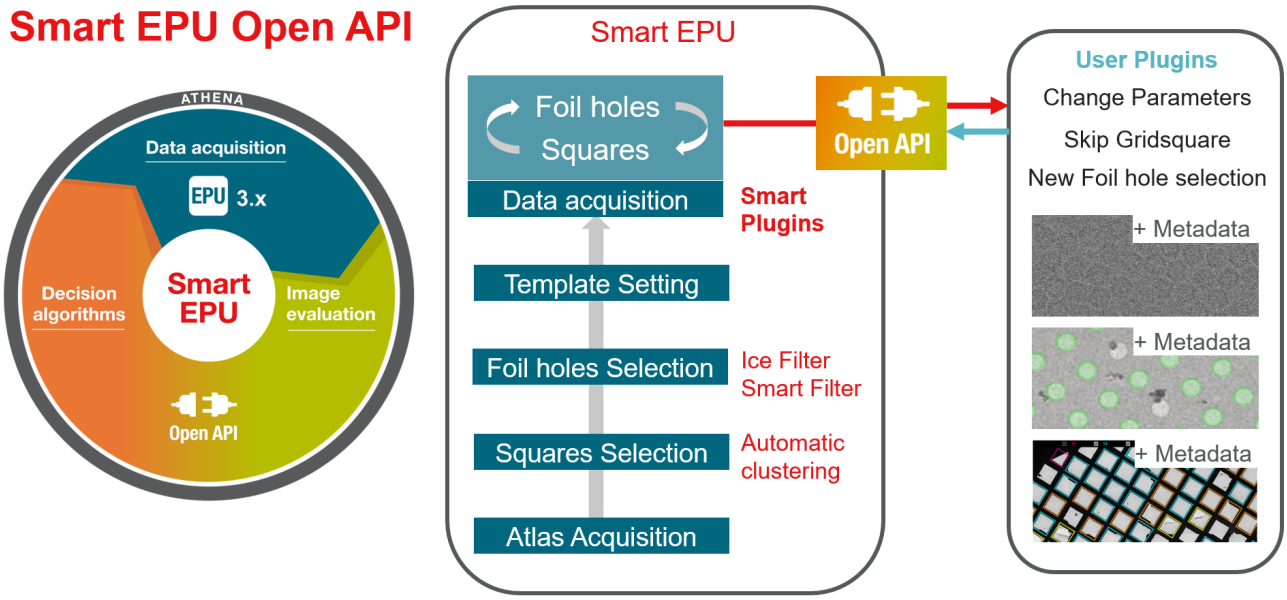

Figure 1: Smart EPU Software as a solution

A significant feature of Smart EPU Software is an Open Application Programming Interface (open API) that allows advanced users to develop their own plugins to customize certain critical steps in the preparation of a data acquisition session while still benefiting from the powerful capabilities of Smart EPU. These user-developed plugins can evaluate microscope outputs and influence Smart EPU Software according to specific needs. The open API facilitates innovation and the exchange of setups in the cryo-EM community.

High Level Description

When creating a plugin that needs to be connected to the Smart EPU solution, you must understand the **feedback loop** pattern as defined in Figure 2:

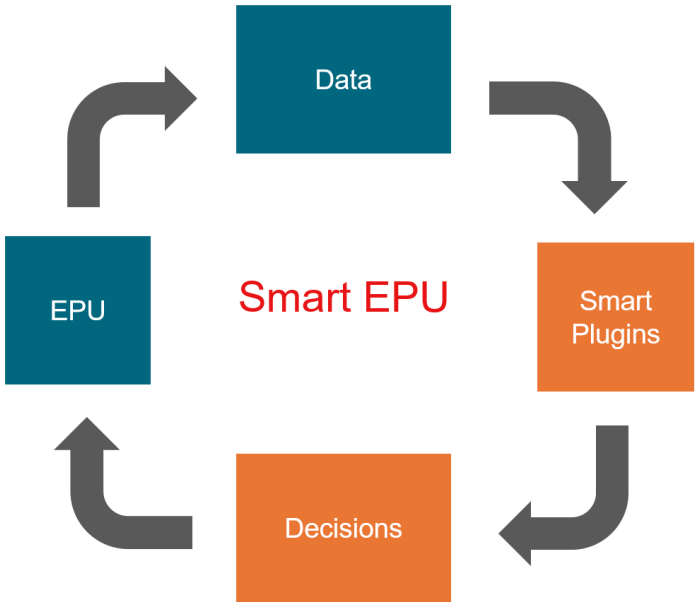

Figure 2: Representation of the feedback loop in Smart EPU

Where:

- **EPU** is the main application that supports the SPA workflow with a set of decision points that can be **influenced** by a set of EPU Smart Plugins or **User Plugins**.
- **Data** represents the image raw data, image metadata, and EPU-specific data models.
- **User plugins** are a set of (user-defined) algorithms or AI/ML models that consume **data** produced using the **EPU** application. The result of these plugins will be a Decision.
- **Decisions** are a set of algorithm (plugin) results that, if available, will be ingested by the **EPU** application and alter the EPU recipe.

EPU provides a defined set of decision points where algorithms can interact. In this initial phase, those decision points are:

- **Autofocus measurement**: a plugin can provide the focus values that normally the autofocus measurement in **EPU** would produce. If this value is present, then EPU will use it and skip performing the Autofocus measurement.
- **Stage waiting time**: only applicable in EPU's accurate acquisition mode, a plugin can provide this value (in seconds) that can be used to induce a delay in the next stage movement during the EPU image acquisition. This prevents image noise caused by mechanical vibrations.
- **Foil hole selection**: a plugin can specify in its resulting Decision a list of foil holes (based on area id) that will be selected for the EPU acquisition. This can alter or complement the **Find Holes** functionality during the Acquisition Run in an Automated Recipe.
- **Grid square selection**: a plugin can specify in its resulting Decisions whether it will skip the current grid square or not.

The Decision mechanism and most of the input source of data can be accessed from the Smart EPU Open API. The API specifications docs (**swagger-based documentation**) can be accessed locally from your Data

Management Platform (DMP) instance at the following link: `https://<athena-or-dmp-hostname>>/api/smartepu/docs/swagger/index.html`.

The Athena API docs can also be accessed via the following link: `https://<athena-or-dmp-hostname>>/api/aggregator/swagger/index.html`. This API offers the possibility to retrieve files and metadata.

Smart EPU Client

Smart EPU Client provides a convenience package for Python developers to connect **User Plugins** with **Smart EPU** solution offering. It abstracts the Smart EPU Open API and the Athena API, offering:

- Easy and Pythonic access to the EPU API to influence acquisition (`smartepu_client/decision_service/client.py`);
- Easy and Pythonic access to the Athena API in order to access data and metadata (`smartepu_client/athena/api.py`);
- Means to connect and integrate the user-developed plugins with the Smart EPU solution (`smartepu_client/algorithm_runner.py`);
- Frequently used patterns in algorithms (`smartepu_client/common/algorithm.py`);
- Logging (`smartepu_client/common/log.py`);
- Configuration functionality in (`smartepu_client/common/settings_utils.py`);
- Definition of internal data models (`smartepu_client/epu/data_structure.py`).

All of this is offered to simplify the development of User Plugins, minimizing the challenges to connect to the **Smart EPU** solution.

How does it work

The following diagram (Figure 3) presents how a **User Plugin** and a Smart EPU plugin use this **Smart EPU Client** package to interact with the Smart EPU solution:

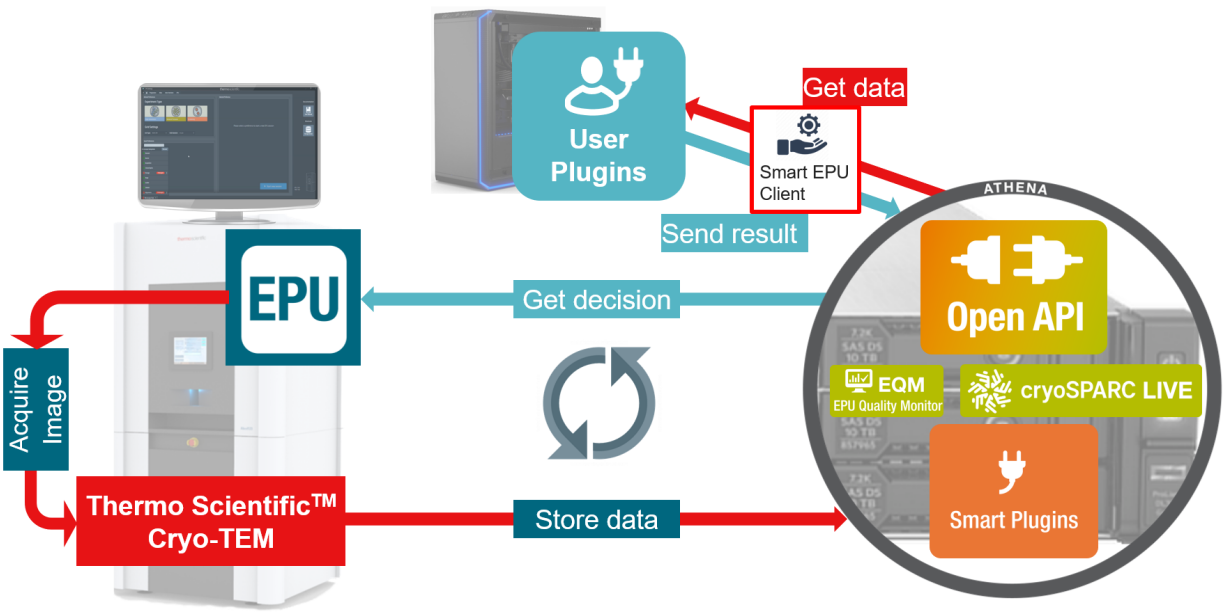

Figure 3: Representation of Smart EPU data flow in the context of User Plugins and Smart EPU plugins

Here, the data exchange with the Open API is abstracted by this **Smart EPU client** by calling the methods defined in the `smartepu_client/decision_service/client.py`.

Please note: *Currently, only polling is supported by the Smart EPU Client. A direct inference approach using events is scheduled for next releases.*

How to install the SmartEPU Client

In order to be able to use the full capabilities of this client, the following **wheel** package needs to be installed:

1. *smartepu_client*

If you have this package locally, it can be installed as follows:

```
pip install </path/to/>smartepu_client.whl
```

Please Note:

- It requires **Python 3.6.6+** to run and is still in beta phase.
- If you experience any issues, please contact us: epuopenapi@thermofisher.com.

Configuration

The package is configured through a configuration file (YAML) that needs to be accessible by the library. A template is provided under `config/template.yaml`.

```
# Decision service settings
decision_service:
  # base URL to decision service
  url: https://<dmp-host-name>/api/smartepu/api/v1
  # default Client ID for communication
  client_id: <decision-service-client-id>
  # default Client secret for communication
  client_secret: <decision-service-client-secret>
  # secrets file that can be used to overwrite default client id and secret
  path_to_secret_file: decision_service_secrets.json
  # maximum number of connections
  max_num_connections:

# Athena settings
athena:
  # base URL to athena
  url: https://<athena-host-name>
  # default Client ID for communication
  client_id: <athena-client-id>
  # default Client secret for communication
  client_secret: <athena-secret>
  # secrets file that can be used to overwrite default client id and secret
```

```
    path_to_secret_file: athena_secrets.json

  # Algorithm settings
  algorithm:
    # in seconds
    task_interval:
    # max number of concurrent tasks for execution
    max_number_of_parallel_tasks:
    # max number of areas. If infinity, leave as inf
    max_number_of_areas:

  # Communication configuration. This part is only needed if the user wants to
  switch to event driven architecture instead
  # of the default polling method.
  application_bus:
    # Host_IP:Port of the bootstrap kafka server. Typically, the port will be 9092
    bootstrap_server: dmp-components-kafka.ls-platform-apps:<port>
    # The Group ID that the consumer must register under
    group_id: <deployment>.smartepu-plugins.<plugin-name>
    # URL to Schema Registry. Typically, the port will be 8081
    schema_registry_url: http://ls-platform-apps-schema-registry.ls-platform-apps:
  <port>
    # White-space separated list of topics to be subscribed to
    topics: v1.decision_service.algorithm.input
    # Any optional configuration for the consumer can be listed below. It should be
  given as is described at
    # https://github.com/confluentinc/librdkafka/blob/master/CONFIGURATION.md
    # Example: auto offset reset becomes:
    auto.offset.reset: True
```

Ensure all entries present within the template are included for the file to be parsed correctly. If a field is not necessary, leave the value blank. The library can establish communication with the Decision Service and Athena using the provided information. Note the `path_to_secret_file` parameter in both `athena` and `decision_service` sections. By default, users can provide their ID and secret as part of this configuration file. However, if those are to be kept secure and distributed as secrets by K8s, users can provide the path where this file can be found. The algorithm will automatically read the file and overwrite the values in `client_id` and `client_secret`.

Please Note: To connect to the Smart EPU solution as a **User Plugin**, you will require security credentials (clientId, clientSecret) for both the **Decision Service API** and the **Athena API**. Please contact us at: epuopenapi@thermofisher.com so that we can help you by providing these values.

Once the configuration file is complete, ensure that the environment variable `SMARTEPU_CLIENT_CONFIG_PATH` is set. Example:

- On Linux, execute `export SMARTEPU_CLIENT_CONFIG_PATH=<path_to_config_file>` before you start your Python process. This **does not** modify the environment permanently, so you will **have to** provide it each time you close your terminal.
- In Python, add the following snippet before you instantiate the AlgorithmRunner:

```
import os

os.environ["SMARTEPU_CLIENT_CONFIG_PATH"] = "/path/to/config/file.yaml"
```

To create a Smart EPU auto function, create an AlgorithmRunner object and feed it the necessary functions. Then call `start()` on the created object. For example:

```
from smartepu.algorithm_runner import AlgorithmRunner
from smartepu.common.log import get_and_configure_logger
from smart_plugin_data import SmartPluginData
from smart_plugin_algo import SmartPluginAlgorithm

logger = get_and_configure_logger()
smart_plugin = SmartPluginAlgorithm(logger)
smart_plugin_data = SmartPluginData(logger)

runner = AlgorithmRunner(smart_plugin.create_autofocus_task,
smart_plugin_data.determine_next_inputs_for_autofocus,
                         logger)
runner.start()
# If running in event driven mode, and assuming listening for spa.eqm.micrograph
type events, change the above call to:
# runner.start(use_event_driven=True, select_type="spa.eqm.micrograph")
```

How to write a custom User Plugin using this client

To create a User Plugin, create an instance of the AlgorithmRunner class and parameterize it with:

- **Logger**: Conforms to the standard Python logger interface.
- **DataSource**: A class that retrieves data from the EPU and returns a data tuple when calling its main function. More details in the next section.
- **Algorithm**: A class that consumes the data tuple produced by DataSource. The Algorithm processes the data and posts Decisions to the DecisionService.

Using polling approach

- **IMPORTANT:** This approach requires **at least EPU 3.3** and the **corresponding Athena version (>= 1.19)**. Please contact service for an upgrade.
- **IMPORTANT:** The custom plugin shall be executed from a network that can access the DMP. (**by default**: the microscope network or the network that DMP is configured to be accessible).

The AlgorithmRunner and Logger classes are part of the **SmartEPU-Client** Python module. The **smartepu_client** module contains all the tools necessary to create a SMART EPU algorithm. Once the AlgorithmRunner is created and injected with the required components, calling the blocking method start() begins the algorithm. It never terminates but runs in a loop.

```
from smartepu.algorithm_runner import AlgorithmRunner
from smartepu.common.log import get_and_configure_logger
from smartepu_algorithm_data import SmartEpuAlgorithmData
from smartepu_algo import SmartEpuAlgorithm
from smartepu_client.models.smart_plugin_type import SmartPluginType

logger = get_and_configure_logger()
smartepu_algo = SmartEpuAlgorithm(logger)
smartepu_algorithm_data = SmartEpuAlgorithmData(logger)

runner = AlgorithmRunner(smartepu_algo.create_smartepu_task,
smartepu_algorithm_data.determine_next_inputs, logger,
SmartPluginType.CUSTOM_SMART_FOCUS)
runner.start()
```

When creating a Smart Plugins make sure you use the correct **SmartPluginType** (see `smartepu_client/common/smart_plugin_type.py`). A user-defined plugin should use a plugin type starting with "CUSTOM"

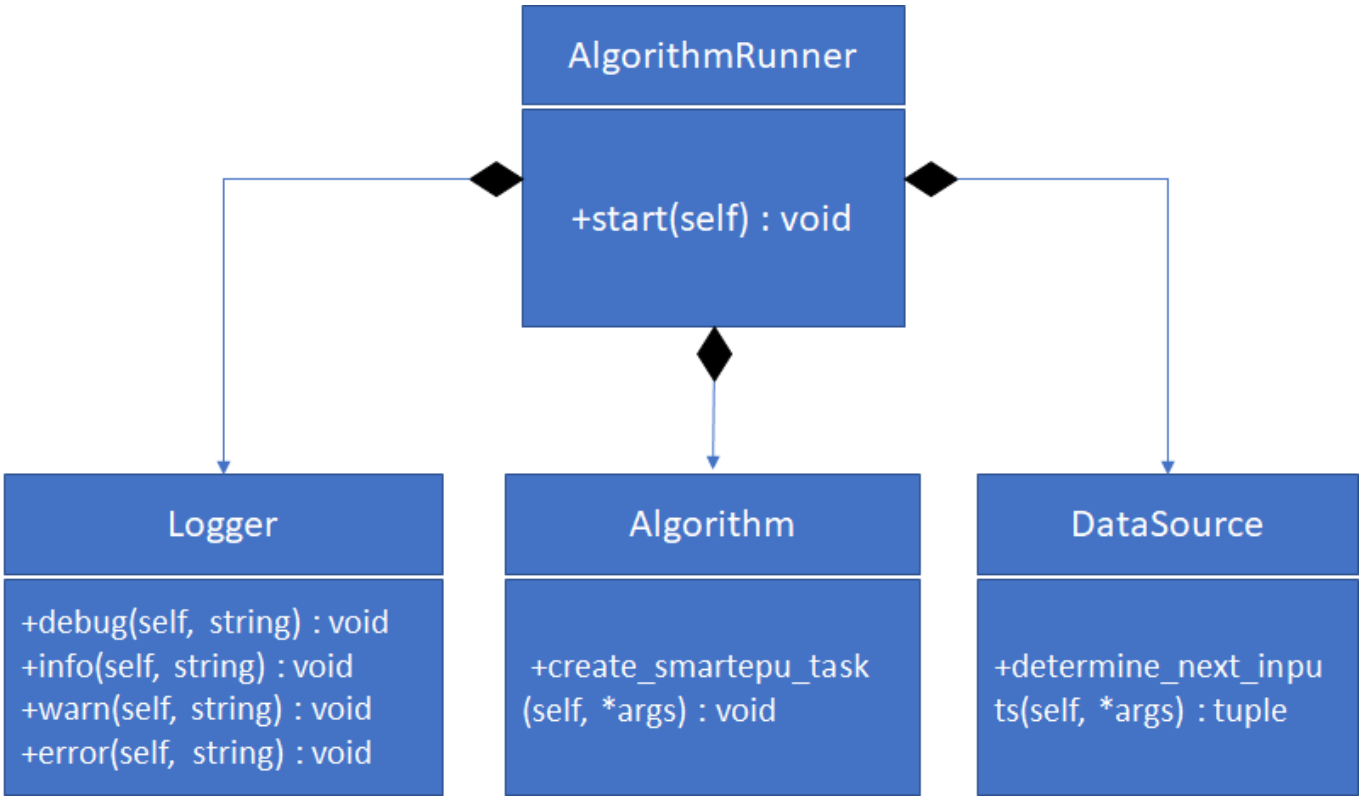

Figure 4: AlgorithmRunner class diagram

DataSource

This class obtains data from the DecisionService (EPU API) and returns a data tuple consumed by the Algorithm class. The class should accept the logger object and create an EpuStateProcessor object, which monitors the state of the EPU. Below is an example code snippet.

```
from requests.exceptions import ConnectionError, HTTPError
from smartepu.common.algorithm import handle_errors
from smartepu.epu.state_processor import EpuStateProcessor
from smartepu.epu.data_structure import EpuDataStructure


class SmartEpuAlgorithmData:
    def __init__(self, logger):
        self.logger = logger
        self.epu_processor = EpuStateProcessor(self.logger)

    @handle_errors((HTTPError, ConnectionError), action="log")
    def determine_next_inputs(self):
        """Function that provides a models tuple to the Algorithm"""
        pass
```

Next, the function `determine_next_inputs` needs to do something, like getting data from the EPU/Athena. The code snippet below shows how to obtain EPU data via the EpuDataStructure. This function, polling the DecisionService, runs every 5 seconds via the AlgorithmRunner. The polling time is configurable through environmental variables (see below). First, `self.epu_processor` gets the current session. If valid, it checks if there is a valid current grid square. If so, an `EpuDataStructure` object is created and populated. The EpuDataStructure is modeled as a tree. In this case, the current grid square is populated with all the foil hole area, and all the foil holes are populated with all the particles. Next, various metadata is populated, such as foil coordinates, CTF, and (particle) image metadata. Motion correction data is also available but not used in this example. Below is an illustration of the EpuDataStructure tree modeling the EPU data hierarchy.

```
  @handle_errors((HTTPError, ConnectionError), action="log")
  def determine_next_inputs_for(self):
    """Function that provides a models tuple to the Algorithm"""
    try:
        sess = self.epu_processor.get_current_session()
        if sess is None:
            return

        cur_grid_sq_id, cur_fh_id =
self.epu_processor.get_current_grid_and_foilhole(sess['sessionId'])
        if cur_grid_sq_id is None:
            return

        epu_ds = EpuDataStructure(sess['sessionId'], self.logger)

        epu_ds.add_grid_square(cur_grid_sq_id, cur_fh_id)
        epu_ds.update_foil_holes(cur_grid_sq_id)
        epu_ds.update_coordinates()
        epu_ds.update_particles()
        epu_ds.update_metadata()
        if not epu_ds.update_ctfs(cur_grid_sq_id):
            self.logger.debug("no CTF results found, aborting")
            return
```

```
    except Exception as e:
        self.logger.error(f"Error in EPU models: aborting {e}")
        return

      .... # code that traverses the EpuDataStructure
```

EPU data structure: Each node may or may not have metadata such as CTF, X,Y coordinates, etc., associated with it.

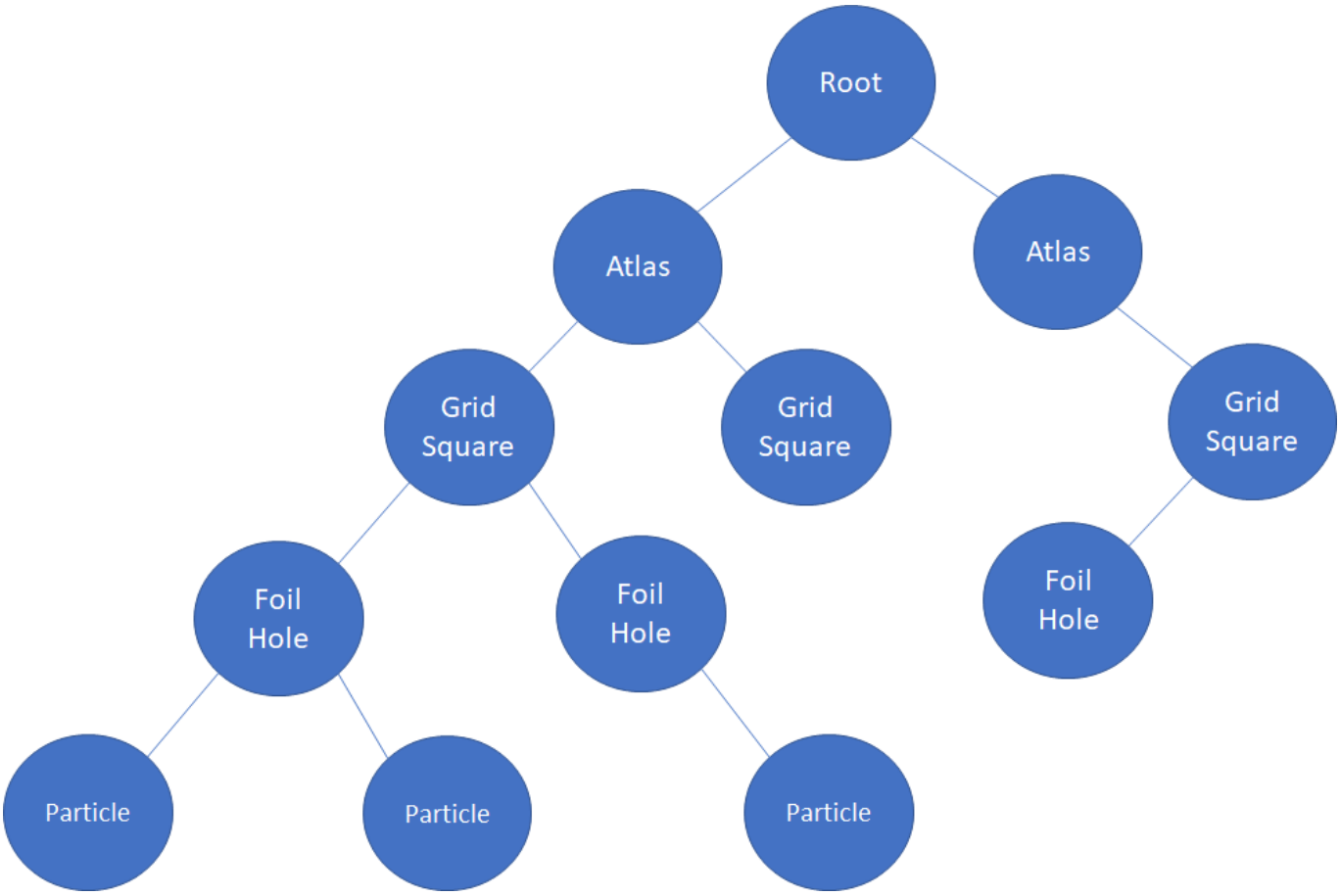

Figure 5: Representation of the EPU area data structure

Once the `EpuDataStructure` is populated, it is necessary to traverse the tree to extract the desired data. The extracted data is then returned and consumed by the Algorithm. See the code snippet below:

```
@handle_errors((HTTPError, ConnectionError), action="log")
def determine_next_inputs(self):
  """Function that provides a models tuple to the Algorithm"""

  ...

  # Extract the models from the EPU models structure by traversing the tree
  coords_dict = {}
  defocus_dict = defaultdict(list)
  fhs_in_cur_grid_sq = []

  grid_area = epu_ds.areas[cur_grid_sq_id]
```

```
    foil_hole_ids, _ = grid_area.get_ordered_children_ids()
    for fh_id in foil_hole_ids:
        fh = grid_area.children[fh_id]

        fhs_in_cur_grid_sq.append(fh)
        coords_dict[fh_id] = fh.coords

        for particle_id, particle in fh.children.items():
            if fh.ctf is None:
                continue
            defocus_entry = {"ctf": fh.ctf[0]}

            if particle.image_meta_data is not None:
                defocus_entry["opt_defocus_m"] =
particle.image_meta_data["microscopeData"]["optics"][
                    "Defocus"
                ]
            defocus_dict[fh_id].append(defocus_entry)

    next_fhs = self.epu_processor.get_next_foilholes(fhs_in_cur_grid_sq,
cur_fh_id, fhs_ordered)

    return sess, fhs_in_cur_grid_sq, next_fhs, coords_dict, ctfs_dict,
defocus_dict
```

Additionally, you can download the raw image files for particles (here, micrographs) and grid squares. The `EpuDataStructure` provides these utilities in the form of the `download_micrograph_image` and `update_gridsquare_metadata` methods, respectively. These can be useful when extra postprocessing or decoding is required:

```
import base64
import numpy as np
import cv2
from mrc_str_reader import MrcStream
from smartepu.models.area_image_type import AreaImageType
from smartepu.epu.data_structure import EpuDataStructure as epu_ds


def decode_image(img_string, bit_depth):
    if bit_depth != 8:
        raise ValueError("Bit depth of %d not supported" % bit_depth)

    base64_decoded = base64.b64decode(img_string)
    npimg = np.fromstring(base64_decoded, np.uint8)

    return cv2.imdecode(npimg, cv2.IMREAD_GRAYSCALE)


def get_micrograph_image(areaId: int, area_image_type: AreaImageType):
    # Grab image from Athena
    try:
```

```
        response = epu_ds.download_micrograph_image(areaId, area_image_type)
        mrc_reader = MrcStream(response.read(None))

        return mrc_reader.data
    except Exception as e:
        raise ValueError("Cannot access MRC resource")


def get_gridsquare_image(areaId: int):
    # Grab image from Athena
    try:
        response = epu_ds.download_gridsquare_image(areaId)
        mrc_reader = MrcStream(response.read(None))

        return mrc_reader.data
    except Exception as e:
        raise ValueError("Cannot access MRC resource")
```

Please Note: In the example above, the MRC reader/stream is not provided with this package. It serves as an illustrative example. If you are interested in such utilities, please contact R&D support.

Using event driven approach

As an alternative, you can opt to use an event-driven approach. In this way, the plugin's `determine_next_inputs` function will be triggered not every X seconds (as in the default polling approach), but will be triggered by an event on an application bus. To switch between, the last call of the above snippet `runner.start()` needs to be changed to `runner.start(use_event_driven=True, select_type="<type_of_plugin>")`, replacing `<type_of_plugin>` with a string to be used to filter through events (to ensure the plugin is only called for the correct decisions).

- **IMPORTANT:** This approach requires **at least EPU 3.4** and the **corresponding Athena version (>= 1.19)**. Please contact service for an upgrade.
- **IMPORTANT:** The custom plugin shall be executed from a network that can access the DMP. (**by default**: the microscope network or the network that DMP is configured to be accessible).
- **IMPORTANT:** To make use of the event driven approach, make sure that you populate the `application_bus` section of the configuration file.
- **IMPORTANT:** When using event driven, you have to make sure that the `determine_next_inputs` function can accept an argument called `message_body`, This argument will be a dictionary which includes the contents of the event (the message). The mechanism through which the developer handles this (be that argument, `args` or `kwargs`) is up to the developer.

Here is an example snippet of how to trigger the event driven approach:

```
from smartepu.algorithm_runner import AlgorithmRunner
from smartepu.common.log import get_and_configure_logger
from smart_plugin_data import SmartPluginData
from smart_plugin_algo import SmartPluginAlgorithm

logger = get_and_configure_logger()
```

```
smart_plugin = SmartPluginAlgorithm(logger)
smart_plugin_data = SmartPluginData(logger)

runner = AlgorithmRunner(smart_plugin.create_autofocus_task,
smart_plugin_data.determine_next_inputs_for_autofocus,
                         logger)

# If running in event driven mode, and assuming listening for spa.eqm.micrograph
type events, change the above call to:
runner.start(use_event_driven=True, select_type="spa.eqm.micrograph")
```

The second point above is the main difference between how the two approaches operate - while in polling the algorithm itself is responsible for determining if it should perform a run (are all necessary inputs present?), the kafka-enabled event driven approach allows the system to tell the plugin that it has a set of inputs that it should operate on. In addition, while in polling the process of building the input structure may take many requests sent to Decision Service, in event driven the majority of this information will be available through the event's message body. A message's body is passed along with the `message_body` argument to the `determine_next_inputs` function. The `message_body` itself is a JSON structure that has already been deserialized (and content-checked against a schema provided from a Schema Registry) and converted to a nested dictionary. The custom plugin is responsible for extracting necessary useful information from the dictionary and using it further inside its operations.

IMPORTANT: As of the time of writing, the types of events (and corresponding data) that are available for plugins to consume are: Gridsquare, Micgrograph, Motion Correction and CTF:

- Gridsquare events are thrown when "Detect Holes" is used with enabled Smart Filter (wthin the EPU tab). The corresponding `select_type` is `spa.gridqsquare`.
- Micrograph events are thrown during an EPU run. They are triggered by Athena when data is offloaded from Camera Offload Service. The corresponding `select_type` is `spa.micrograph`.
- Motion Correction events are thrown during an EPU run. They are triggered by eqm/ecl activity. The corresponding `select_type` are `spa.ecl.motion.correction` and `spa.eqm.processor.motion.correction`.
- CTF events are thrown during an EPU run. They are also triggered by eqm/ecl activity. The corresponding `select_type` are `spa.ecl.ctf` and `spa.eqm.processor.ctf`.
-

As mentioned above, the `message_body` needs to comply to a pre-defined schema. This means that its high-level key-value pairs must conform to the following schema (as of time of writing):

```
{"$schema":"http://json-schema.org/draft-04/schema#",
"title":"AlgorithmResultRecord",
"type":"object",
"description":"Represents an algorithm result record.",
"additionalProperties":false,
"properties":
    {
        "SessionId":
            {
                "type":"string",
```

```
                "description":"The id that uniquely identifies the session.",
                "format":"guid"},
        "AreaId":
            {
                "type":"integer",
                "description":"The id that uniquely identifies the area.",
                "format":"int32"},
        "Name":
            {
                "type":["null","string"],
                "description":"The name of the algorithm result."},
        "Result":
            {
                "type":["null","object"],
                "description":"The algorithm result represented as a collection of
string-object pairs.",
                "additionalProperties":{}},
        "Timestamp":
            {
                "type":"string",
                "description":"The time that the algorithm result was generated.",
                "format":"date-time"}
    }
}
```

Algorithm

This class consumes data from the DataSource and posts decisions. Below is a code snippet showing the creation of such a class. Note how the functions `create_smartepu_task` and `smartepu_task_fn` consume the data from the DataSource. The AlgorithmRunner uses `create_smartepu_task` to create a task every poll and pushes it onto a job queue. When run, the task calls `smartepu_task_fn`.

```
from smartepu.decision_service.client_provider import
DecisionServiceClientProvider
from smartepu.common.algorithm import Task


class SmartEpuAlgorithm:
    def __init__(self, logger):
        self.logger = logger

    def create_smartepu_task(self, sess, foil_holes, next_foil_holes, coords_dict,
defocus_dict):
        smartepu_task = Task(
            self.smartepu_task_fn,
            args=(sess, foil_holes, next_foil_holes, coords_dict, defocus_dict),
            kwargs={},
            use_gpu=False,
        )
        return smartepu_task
```

```
    def smartepu_task_fn(self, sess, foil_holes, next_foil_holes, coords_dict,
defocus_dict, timeout=None):
        """Function that consumes that models tuple and posts decisions"""
        pass
```

Next, the function needs to process the data and then post a decision(s). The code snippet below shows this in action:

```
  def smartepu_task_fn(self, sess, foil_holes, next_foil_holes, coords_dict,
defocus_dict, timeout=None):
      """Function that consumes that models tuple and posts decisions"""

      smart_values_list = []
      ... # analyze the models

      decisions = DecisionServiceClientProvider().client.post_decisions(
          sess["sessionId"],
          area_ids=area_ids_list,
          dec_types=["smart_value"] * len(smart_values_list),
          dec_vals=smart_values_list,
          dec_bys=["smartepu_client algorithm"] * len(smart_values_list),
      )

      return decisions
```

Example of a FoilHole Selection algorithm using the event-driven approach

To speed up development and better understand how to use this Smart EPU Client, we included an example algorithm that is based on event containing GridSquare metadata. This example algorithm will:

1. Download the raw GridSquare image via Athena Client
2. Infer the GridSquare metadata and image to a fictive AI Model.
3. Push results to Smart EPU microservices via Smart EPU Client

As defined in the previous sections, the first step is to define your main run.py:

```
from smartepu_client.algorithm_runner import AlgorithmRunner
from smartepu_client.common.log import get_and_configure_logger
from smartepu_client.common.settings_utils import get_configuration
from smartepu_client.models.smart_plugin_type import SmartPluginType
from app.foil_hole_selection_algo import foil_hole_selection
from app.foil_hole_selection_data import FoilHoleSelectionData


def run():
    """
    A wrapper QOA function that takes care of parsing necessary configurations and
executing the AlgorithmRunner for
    the foil-hole-selection algorithm as an event_driven application.
```

```
    Returns
    -------
    None
    """
    logger = get_and_configure_logger()
    config = get_configuration(logger)
    foil_hole_selection_data = FoilHoleSelectionData(logger,
                                                     config['algorithm']
['max_number_of_areas'],
                                                     config['athena']
['url'],
                                                     config['athena']
['client_id'],
                                                     config['athena']
['client_secret'],
                                                     )

    runner = AlgorithmRunner(
        foil_hole_selection,

foil_hole_selection_data.determine_next_inputs_for_auto_foil_hole_selection,
        logger,
        # This will ensure you can use the Smart Filter (FoilHole) action
functionality in EPU in custom mode.
        # By running this you will notice a change in the UI of the toggle button.
        SmartPluginType.SMART_FOIL_HOLE_FILTER,
    )
    # the spa.gridsquare sets the event-driven mechanism to listen for events that
match with this tag.
    # This event will include the GridSquare metadata
    runner.start(use_event_driven=True, select_type="spa.gridsquare")
```

Then we define the algorithm specific data structures:

```
import json
from requests.exceptions import ConnectionError, HTTPError
from smartepu_client.common.algorithm import handle_errors
from smartepu_client.epu.state_processor import EpuStateProcessor
from app.utils import get_image


class FoilHoleSelectionData:
    """
    A class meant to handle the preparation of the inputs for the algorithm. This
mostly handles the parsing of the event
    message coming from the application message bus, but also fetches necessary
image(s) from Athena.
    """
    def __init__(self, logger, max_number_of_areas, athena_url, athena_client_id,
athena_client_secret):
        self.logger = logger
```

```
        self.epu_processor = EpuStateProcessor(self.logger, max_number_of_areas)
        self.athena_url = athena_url
        self.athena_client_id = athena_client_id
        self.athena_client_secret = athena_client_secret

    @handle_errors((HTTPError, ConnectionError), action="log")
    def determine_next_inputs_for_foil_hole_selection(self, message_body):
        """
        Parses the contents of the message inside the
v1.decision_service.algorithm.input event. This also
        fetches the image via `app.utils.get_image` from Athena (underlying
mechanism is SmartEPU-Client).

        Parameters
        ----------
        message_body: dict
            Parsed json event message as a dictionary

        Returns
        -------
        tuple
            Inputs required for the execution of the algorithm. This consists of:

            logger: logging.Logger()
                The logger to be used during the execution
            sess_id: str
                The session ID from the event's message
            area_id:
                The id of the area of interest
            grid_type: str
                The type of grid being used
            img: numpy array (r x c)
                The grid square image
            foil_hole_coords: numpy array (n x 2)
                The list of 'n' coordinates for foil hole locations
            pixel_size: 2-tuple
                The pixel size of the grid square image
            hole_spacing:float
                The spacing between foil holes

        """
        hole_spacing = float(message_body['Result']
['application.epu.gridsquare.foilhole.space'])
        pixel_size = json.loads(message_body['Result']
['application.epu.gridsquare.image.pixel.size'])
        grid_type = message_body['Result']['application.epu.gridsquare.grid.type']
        pixel_size = (pixel_size['X'], pixel_size['Y'])
        area_id = message_body['AreaId']
        sess_id = message_body['SessionId']
        foil_hole_coords = json.loads(message_body['Result']
['application.epu.gridsquare.foilholes'])
        foil_hole_coords = [[int(i['X'] + 0.5), int(i['Y'] + 0.5), int(i['Id'])]
for i in foil_hole_coords['FoilHoles']]
```

```
        # To fetch raw image we need the file data id. This is provided in the
body of the spa.gridsquare event.
        img = get_image(message_body['Result']
['application.epu.gridsquare.file.data']['Id'],
                        athena_url=self.athena_url,
                        athena_client_id=self.athena_client_id,
                        athena_client_secret=self.athena_client_secret,
                        logger=self.logger)

        return (self.logger,
                sess_id,
                area_id,
                grid_type,
                img,
                foil_hole_coords,
                pixel_size,
                hole_spacing)
```

Where *get_image* is defined as follows:

```
import logging

from app.mrc_str_reader import MrcStream
from smartepu_client.athena.client import AthenaClient

def get_image(_id: str, athena_url: str, athena_client_id: str,
athena_client_secret: str, logger: logging.Logger):
    # Grab image from Athena. For this we will use the Athena Client embedded in
the tar ball for Smart EPU Client
    athena_client = AthenaClient(url=athena_url, client_id=athena_client_id,
client_secret=athena_client_secret)

    try:
        response = athena_client.get_client().get_file_binary(file_id=_id)
        mrc_reader = MrcStream(response.read(None))
        return mrc_reader.data
    except Exception as e:
        logger.error(f"MRC Resource not available or corrupt due to {str(e)}")
        raise ValueError("Cannot access MRC resource")
```

Last, but not least we define the algorithm. Here:

1. Determine the valid coordinates. This needs to be implemented according to your uses-case. In this example, it will also be used for defining the result of inference and encapsulate this in a decision. This decision will be used in EPU to render the foil hole selection. This means this array should include at least the valid FoilHole AreaIds
2. Execute a "fictive" AI inference.
3. Push results

```
import numpy as np
from smartepu_client.decision_service.client_provider import
DecisionServiceClientProvider

class FoilHoleSelectionAlgo:
    """Class to encapsulate the Foil Hole Selection algorithm/model

    Attributes
    ----------
    logger: Logger
        The logger. Must implement the standard Python Logger interface

    Methods
    ----------
    foil_hole_selection(self, sess, grid_square_img, coords, pixel_size,
hole_spacing, timeout=None)
        performs inference on all foil hole crops and push accept/keep decision to
DS
    """

    def __init__(self, logger):

        self.logger = logger
        self.fh_window_size_factor = DEFINE_YOUR_WINDOW_SIZE
        self.classifier_threshold = DEFINE_YOUR_THREASHOLD
        self.batch_size = DEFINE_YOUR_BATCH_SIZE

    @property
    def ds_client(self):
        return DecisionServiceClientProvider().client

    def foil_hole_selection(self, sess_id, grid_square_img, coords, area_id,
pixel_size, hole_spacing,
                                        timeout=None):
        """ Predict foil hole quality and decide to reject/accept foilhole. Push
results to Decision Service

        Parameters
        ----------
        sess_id : str or uuid
            the EPU session
        grid_square_img : numpy array (r x c)
            the grid square image
        coords : numpy array (n x 2)
            the list of 'n' coordinates for foil hole locations
        area_id : int
            the grid id
        pixel_size : 2-tuple
            the pixel size of the grid square image
        hole_spacing : float
            the spacing between foil holes

        Returns
```

```
        -------
        list
            selections
        """
        self.logger.info(f"Starting AI model inference on grid id {area_id}.")

        # Define your valid coordinates. This table should include at least the
foile hole area Ids
        valid_coords = get_valid_coords(grid_square_img, coords, pixel_size,
hole_spacing)

        # Perform your AI Model inference. To do so you need to provide a
execute_inference method.
        preds = execute_inference(grid_square_img, valid_coords)

        selected = np.asarray(preds >= self.classifier_threshold)

        fh_decisions = [{"areaId": valid_coords[i], "result": str(c[0]).lower()}
for i, c in enumerate(selected)]

        self.logger.info(f"Stopped executing AI model inference. Computed decision
for {len(fh_decisions)} foil holes:")

        self.push_decisions(sess_id, area_id, fh_decisions)

        return selected, [{"areaId": valid_coords[i], "result": c[0]} for i, c in
enumerate(selected)]

    def push_decisions(self, sess_id, area_id, fh_decisions):
        try:
            self.ds_client.post_decision(
                sess_id,
                area_id,
                dec_type="foilHoleSelection",
                dec_val=fh_decisions,
                dec_by="foil hole selection algorithm",
                is_custom_plugin=False
            )

            self.logger.info("pushed decision to Decision Service")
        except Exception as e:
            self.logger.error(f"Could not push decision to Decision Service
because {str(e)}")
```

Glossary

Smart EPU Software: Thermo Scientific platform to increase efficiency, productivity, and ease of use for Cryo-EM, composed of EPU, EPU Multigrid, image evaluation programs (EQM and Embedded CryoSPARC Live), Smart Plugins, and Athena.

EPU: Component of Smart EPU Software facilitating automated screening and data acquisition for Single Particle Analysis (SPA).

EQM: EPU quality monitor, Smart EPU Software image evaluation program capable of executing motion correction and CTF-estimation of CryoEM data, providing image quality metrics.

Embedded CryoSPARC Live: Version of Structura Biotechnology Inc. CryoSPARC Live fully integrated into Smart EPU. It performs data processing on-the-fly for image and sample quality evaluation.

Smart Plugins: Thermo ScientificTM algorithms capable of monitoring data acquisition on-the-fly executed by Smart EPU based on images and associated metadata.

Athena: Thermo ScientificTM visual data management platform enabling Smart EPU Software with a web-based UI accessible on the local network.

Smart EPU Open API: Open Application Programming Interface (API) of Smart EPU Software that allows development of User Plugins for Smart EPU Software. It is a REST API offered through the DMP server.

User Plugins: User-developed algorithms capable of monitoring and influencing data acquisition on-the-fly executed by Smart EPU based on images and associated metadata.

EPU run: Session of data collection after all imaging presets are defined.

MPC: Microscope PC, computer associated with Thermo Scientific cryoTEMs.

DMP: Data Manager Platform, refers to a server connected to Thermo Scientific cryoTEMs for managing the output data and providing processing capabilities. Athena is configured to run on this server.

Smart EPU-Client: A Python library to assist developers in writing User Plugins using the Smart EPU Open API.

EPU Data: Raw images recorded by a camera attached to Thermo Scientific cryoTEMs and processed Smart EPU images by EQM whenever available.

EPU Metadata: Data providing information about the EPU Data, e.g., acquisition parameters, and EQM whenever available.

EPU Data Models: Identified EPU areas of interest (e.g., squares, foil holes) and their status.

Decisions: Results of the algorithms (plugins) that can influence the EPU run.