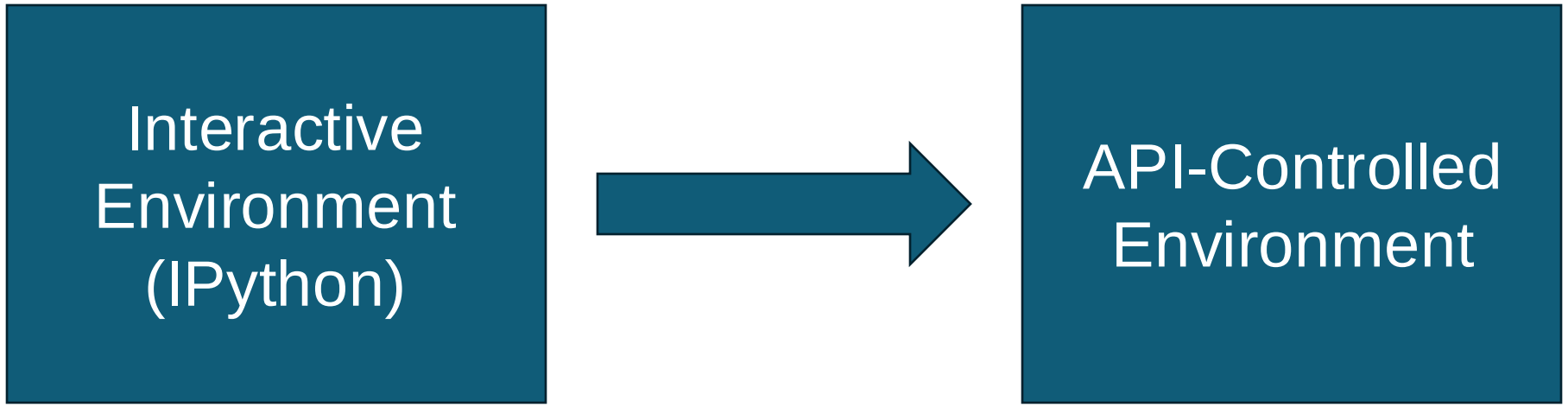
Bluesky Queue Server

Dmitri Gavrilov

September 20, 2024

Purpose of Queue Server



- **Remotely controlled experiments**
- **Autonomous control**
- **GUI**

Components of the QS Stack

- **Run Engine Manager** (RE Manager) – the ‘execution engine’, core component of the stack. 0MQ API.
  https://github.com/bluesky/bluesky-queueserver
- **HTTP Server** – REST API for communicating with RE Manager, authentication and access control.
  https://github.com/bluesky/bluesky-httpserver
- **Python API** – user-friendly Python API for communicating with RE Manager directly (over 0MQ) or via HTTP Server (REST API).
  https://github.com/bluesky/bluesky-queueserver-api
- **RE Widgets** are part of Bluesky-Widgets package: widgets for communicating with RE Manager directly or via the HTTP Server, ‘queue-monitor’ GUI application.
  https://github.com/bluesky/bluesky-widgets

Installation of QS (for evaluation)

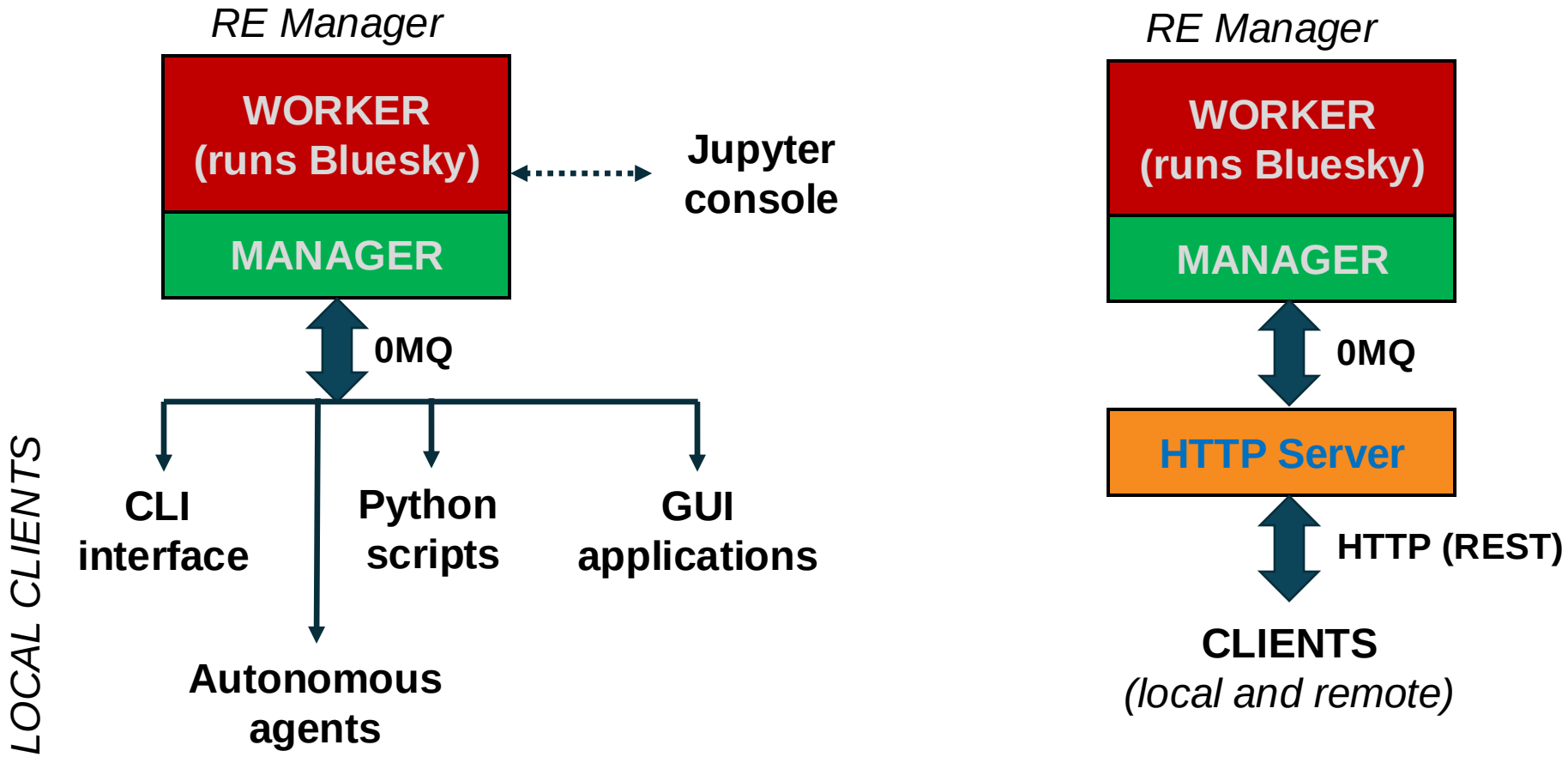
- QS was designed for Linux. It is expected to work correctly on Mac.
- Install Redis: https://redis.io/docs/latest/operate/oss_and_stack/install/install-redis/
- Installation of QS using *conda* (*mamba*):

```
conda create -n bs-qserver python=3.11 pip -c conda-forge
conda activate bs-qserver
conda install bluesky-queueserver bluesky-httpserver bluesky-queueserver-api -c conda-forge
conda install bluesky-widgets pyqt qtpy -c conda-forge
```

- Installation of QS using *pip* (e.x. in activated Conda environment)

```
pip install bluesky-queueserver bluesky-httpserver bluesky-queueserver-api
pip install bluesky-widgets qtpy pyqt5
```

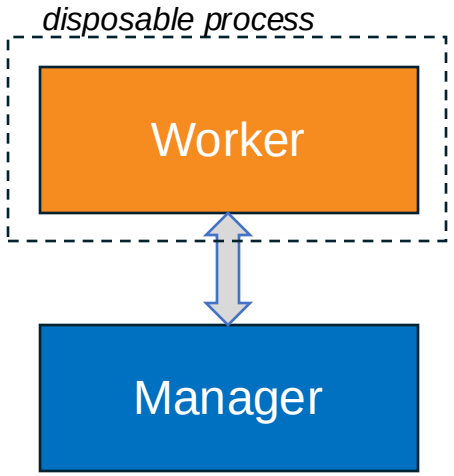
System Configurations



Examples of QS Clients

- **'qserver'**: simple CLI tool (0MQ, 'bluesky-queueserver' package). Access to all RE Manager API, except batch queue operations.
- **'qserver-console-monitor'**: simple CLI tool (0MQ, 'bluesky-queueserver' package). Displays streamed console output of RE Manager.
- **'queue-monitor'** GUI application (0MQ, REST, 'bluesky-widgets' package).
- **Custom GUI based on RE Widgets** (0MQ, REST, 'bluesky-widgets' package).
- **Custom applications or scripts based on Python API** (0MQ, REST, 'bluesky-queueserver-api' package), such as GUI applications, autonomous agents, control scripts etc. The API are friendly enough to be called manually from IPython environment if necessary.
- **Web clients** (REST). Communicate with QS via HTTP server.

RE Manager (Execution Engine)



- RE Manager internally runs two processes: continuously running Manager process and disposable Worker process.
- RE Worker Environment is created in a disposable 'Worker' process.
- Main API:
  - **'environment_open'** - creates the process and loads the startup code into the environment;
  - **'environment_close'** - closes the idle environment (exits the process in orderly way);
  - **'environment_destroy'** - terminates the process, used if the environment becomes unresponsive after crash.

- **The environment** consists of the namespace with objects created by the startup code (including 'RE' instance) and means of executing existing plans ('RE(…)'), functions and uploaded scripts to add/modify objects in the namespace.
- The **state of environment** and executed plans **can be monitored** by remote clients.

Sources of Startup Code

- A directory with alphabetically ordered code files (**IPython-style startup code**). The code files are executed in the worker environment namespace one by one. Configuration parameters: a path to the directory or the name of IPython profile (if the startup code is in the standard IPython location).
- **Python script.** Configuration parameter: a path to the script.
- **Python module.** The module namespace is copied to the environment namespace. Configuration parameter: module name.
- The source is selected using RE Manager parameters.

Startup code can be reloaded by closing then opening the environment (‘*environment_close*’ followed by *‘environment_open*’ requests). This operation opens a new worker process and loads the startup code the clean environment namespace.

IPython Kernel

RE Manager can be configured to start **IPython kernel in the worker process**. Loading of the startup code and execution of plans, functions and scripts are performed in the kernel, which gives access to all IPython features. Users may **connect to the kernel directly using Jupyter console** (0MQ, see ‘qserver-console’ and ‘qserver-qtconsole’ CLI applications). This mode is extremely useful in the following cases:

- **Transition from IPython-based REPL workflow**. Only minor changes to startup code are necessary. Dual workflow (API based access and REPL via Jupyter console) is possible to ease the transition.
- **Development and testing of plans.** Dual workflow allows easy access to the execution environment via Jupyter console to run plans and inspect variables.

Monitoring the Status of RE Manager

```
$ qserver status
12:59:49 - MESSAGE:
{'msg': 'RE Manager v0.0.19.post42+g3d99532',
'items_in_queue': 0,
'items_in_history': 19,
'running_item_uid': None,
'manager_state': 'idle',
'queue_stop_pending': False,
'queue_autostart_enabled': False,
'worker_environment_exists': True,
'worker_environment_state': 'idle',
'worker_background_tasks': 0,
're_state': 're_idle',
'ip_kernel_state': 'disabled',
'ip_kernel_captured': True,
'pause_pending': False,
'run_list_uid': 'd0b8f60b-de68-48f5-a4b8-f14413d2d3f5',
'plan_queue_uid': 'e0de02d4-706c-478c-8100-d57591e3f1fe',
'plan_history_uid': 'fee97e6b-3eb6-460e-be24-9fd84aa3ecc6',
'devices_existing_uid': '4a79b1b2-b223-4da0-8157-bedfe4ae8459',
'plans_existing_uid': '99ad05d3-fbb8-4a52-a489-efe87dc25303',
'devices_allowed_uid': '66d07c68-ea08-48d2-a485-5f5a6ec2a315',
'plans_allowed_uid': '969ae7a0-c349-4f53-a1a8-d10dcd0fcf1b',
'plan_queue_mode': {'loop': False, 'ignore_failures': False},
'task_results_uid': '14f0a382-404d-4d53-884e-b2a442c36838',
'lock_info_uid': 'a8aff45c-73ad-4ccb-9636-84a81808bb9b',
'lock': {'environment': False, 'queue': False}}
```

'status' API returns **information on the state of the RE Manager and the Worker**:

- manager_state – state of RE Manager;
- worker_environment_exists – indicates if the worker exists;
- worker_environment_state – state of the worker;
- re_state – state of Run Engine.

Status can be requested at any time.

Plan Queue and Plan History

- Editable plan queue:
  - Queue items are **plans and instructions**. Only one instruction ('queue_stop') currently exists.
  - Queue can be **modified at any time** (open/closed environment, plan/function/script is executed, etc.)
  - Queue is **stored in Redis** and persists between QS restarts.
  - Each queue item is assigned a **unique UID**.
  - Once queue execution is started, QS is executing the plans one by one until **all the plans are completed, one of the plans fails or is stopped/aborted/halted or the queue is stopped** ('queue_stop' API).
  - If the **'queue_stop' request** is pending, the currently running plan **runs to completion**. The request **can be cancelled** until the queue is stopped.
  - **Use of the queue is optional**. Plans can be submitted by the client one by one using 'queue_item_execute' API.
- Plan history:
  - Contains information on all **executed plans**. The information includes plan parameters and execution results.
  - Plans retain their original **'item_uid'.**
  - The history is not mutable, but it can be **cleared at any time**.

API for Controlling the Queue

Operations on a single item

- queue_item_get
- queue_item_add
- queue_item_update
- queue_item_remove
- queue_item_move

Batch operations (atomic)

- queue_item_add_batch
- queue_item_remove_batch
- queue_item_move_batch

Operations on all elements

- queue_get
- queue_clear

Queue execution

- queue_start
- queue_stop
- queue_stop_cancel
- queue_autostart

Closer look at 'queue_item_add'

Parameters:

- **item: *dict* -** plan parameters.
- **user_group: *str*** – name of the existing user group (e.g. 'primary').
- **user: *str*** – user name (arbitrary string).
- **pos: *int*, 'front', 'back' *(optional)*** - positive or negative int or a string.
- **before_uid, after_uid: *str (optional)*** - insert before or after an item with the given UID.
- **lock_key: *str (optional)*** - perform operation with the locked queue (if key is known).

Example

```
{
  "method": "queue_item_add",
  "params": {
    "item": {"name": "count", "args": [["det1", "det2"]], "kwargs": {"num": 5, "delay": 1}, "item_type": "plan"},
    "pos": "front",
    "user": "Sample User",
    "user_group": "primary",
  },
}
```

Executing Plans

- **'queue-item-execute' API returns immediately.** Other API could be called while the plan is running.
- Plan execution **starts immediately**, otherwise the API call **fails** ('success': False).
- Relevant **error message** is returned ('msg') in case of failure.
- **'item_uid'** is assigned automatically.
- The information on completed plan is added to the **plan history**. If the plan fails, the result contains the error message ('msg') and full traceback.

$ qserver queue execute plan '{"name": "count", "args": [["det1", "det2"]], "kwargs": {"num": 10, "delay": 1}}'

```
13:17:29 - MESSAGE:
{'success': True,
'msg': '',
'qsize': 0,
'item': {'name': 'count',
         'args': [['det1', 'det2']],
         'kwargs': {'num': 10, 'delay': 1},
         'item_type': 'plan',
         'user': 'qserver-cli',
         'user_group': 'primary',
         'item_uid': '3b5d907a-d954-4682-b311-19e03ded5ef9'}}
```

$ qserver history get

```
13:21:23 - MESSAGE:
{'success': True,
'msg': '',
'items': [{'name': 'count',
           'args': [['det1', 'det2']],
           'kwargs': {'num': 10, 'delay': 1},
           'item_type': 'plan',
           'user': 'qserver-cli',
           'user_group': 'primary',
           'item_uid': '3b5d907a-d954-4682-b311-19e03ded5ef9',
           'result': {'exit_status': 'completed',
                      'run_uids': ['bbac7fa3-7bae-46f4-8284-8fbf843c7885'],
                      'scan_ids': [1],
                      'time_start': 1701713849.0494175,
                      'time_stop': 1701713859.3225343,
                      'msg': '',
                      'traceback': ''}}],
'plan_history_uid': 'a9ab3aab-21e5-4253-8c98-5cbea873a223'}
```

Additional Features

Execution of Functions

'function_execute' API is very similar to **'queue_item_execute'** API. Use cases: custom monitoring features for the environment or a plan; communication between running plan and a client application.

- The API parameters include function name and function *args* and *kwargs*.
- The **function must exist in the worker namespace** (it is typically defined in the startup code).
- Functions can be executed in the **foreground thread** (blocks execution of plans and other foreground task) or **background thread**.
- **Background tasks** can be started while a plan is running. The total number of background tasks is reported as part of status ('*worker_background_tasks*').
- **'function_execute' returns immediately**. If API call is successful, the function execution is started in the worker.
- The returned **'task_uid'** can be used to **check the task status** ('task_status' API) and download the result when execution is completed ('task_result' API).

Execution of Scripts

'script_upload' API. Use cases: add objects (e.g. plans or functions) to the environment, modify the existing objects (e.g. edited plans).

- The API accepts a **Python script as text**.
- The script is uploaded to RE Worker and executed in the environment. The script is handled the same as a startup script.
- The scripts can be executed in the **foreground thread** (blocks execution of plans and other foreground task) or **background thread**.
- **'script_upload' returns immediately**. If API call is successful, the script execution is started in the worker.
- The returned **'task_uid'** can be used to **check the task status** ('task_status' API) and download the result when execution is completed ('task_result' API).
- The result contains the full traceback if script execution failed with an exception.

User Group Permissions

- The feature allows to **set filters for the names of plans, devices and functions**. The filters are applied to plans and devices from the worker namespace to generate lists of allowed plans and devices.
- The filters could be tuned to restrict access to plans and devices for different user groups, e.g. beamline scientists and users.
- This feature is designed to **reduce clutter and prevent operator errors**, not as a security feature. It could be used for as part of access control in conjunction with other security features.
- The filters are based on **lists of names** and/or **regular expressions**.
- Initially, the filters are **loaded from disk** (‘user_group_permissions.yaml’ file). The permissions could be **modified by the client** using ‘permissions_get’ and ‘permissions_set' API (e.g. if permissions are managed by a web server).
- The ‘permissions_reload’ API **restores the original permissions** loaded from disk.

Example: ‘user_group_permissions.yaml’

```
user_groups:
    root:  # Defines the rules for preliminary filtering of plan/device/function names for all groups.
        allowed_plans:
            - null  # Allow all
        forbidden_plans:
            - ":^_"  # All plans with names starting with '_'
        allowed_devices:
            - null  # Allow all
        forbidden_devices
            - ":^_:?.*"  # All devices with names starting with '_'
        allowed_functions:
            - null  # Allow all
        forbidden_functions:
            - ":^_"  # All functions with names starting with '_'
    primary:  # Default group. The group can be renamed or other groups may be created.
        allowed_plans
            - ":.*"  # Different way to allow all plans.
        forbidden_plans
            - null  # Nothing is forbidden
        allowed_devices
            - ":?.*:depth=5"  # Allow all device and subdevices. Maximum deepth for subdevices is 5.
        forbidden_devices
            - null  # Nothing is forbidden
        allowed_functions
            - "function_sleep"  # Explicitly listed name
            - ":^func_for_test"
```

Brookhaven National Laboratory

Queue Server API

RE Manager API (0MQ)

- RE Manager creates **two 0MQ sockets**: **control socket** (REQ/REP) and data socket (PUB/SUB). The control socket is used by clients to send requests. Clients may also subscribe to the data socket to receive streamed console output.
- Control socket supports **encryption** (CurveZMQ).
- API are designed to **process the request and send the response** to client almost **immediately**. Quick operations (e.g. 'status' or 'queue_item_add') are executed before the response is sent. For longer operations (e.g. 'environment_open' and 'queue_start'), the response is sent once the operation is initiated. Clients are expected to monitor RE Manager status to wait until long operation is completed.
- **Block queue operations** 'queue_item_add_batch', 'queue_item_remove_batch' and 'queue_item_move_batch' **are atomic**: the queue remains unchanged if the operation fails for any item in the batch and batch processing can not be interrupted by any other queue operation before it completes.
- Detailed documentation of API is available: https://blueskyproject.io/bluesky-queueserver/re_manager_api.html

API Groups

- plans_allowed
- devices_allowed
- plans_existing
- devices_existing

- permissions_get
- permissions_set
- permissions_reload

- script_upload
- function_execute
- task_status
- task_result

- kernel_interrupt

- environment_open
- environment_close
- environment_destroy
- environment_update

- queue_get
- queue_item_add
- queue_item_update
- queue_item_get
- queue_item_remove
- queue_item_move
- queue_item_execute
- queue_clear

- queue_start
- queue_stop
- queue_stop_cancel
- queue_autostart
- queue_mode_set

- status
- config_get
- re_runs

- lock
- lock_info
- unlock

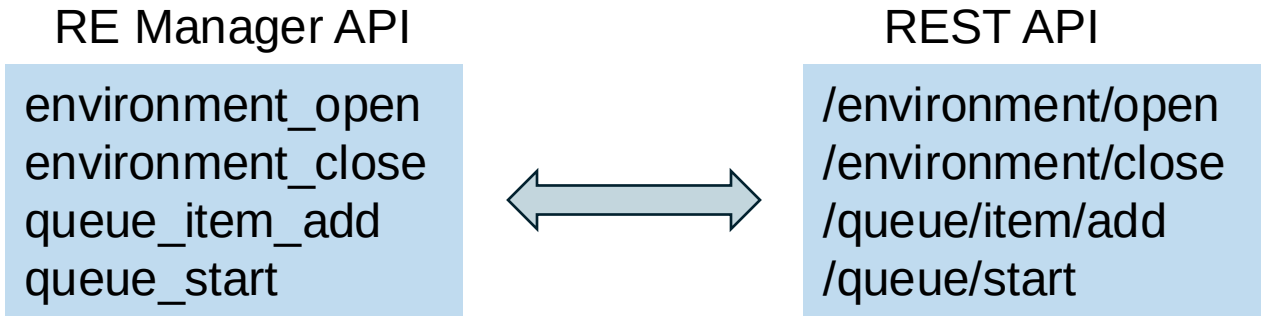
- queue_item_add_batch
- queue_item_remove_batch
- queue_item_move_batch

- history_get
- history_set

- re_pause
- re_resume
- re_stop
- re_abort
- re_halt

HTTP Server API (REST)

- In the current implementation, the **HTTP Server is forwarding all requests to RE Manager**, so the basic set of API is identical.
- The HTTP Server is using Queue Server API package for 0MQ communication that performs caching of status, queue, history etc.).
- HTTP Server provides additional **API for authentication and access control**.



Queue Server API Package

Issues with native RE Manager API:

- Writing each API request as JSON may be **inconvenient and prone to errors**.
- Deeper understanding of RE Manager internals is necessary to write efficient code.
- Two versions of code need to be maintained if an application need to communicate with RE Manager using **both 0MQ and REST API**.

Problem solved by the API from ‘bluesky-queueserver-api’ package:

- Python library: **access to all API by calling class methods**, the parameters are passed as method parameters.
- The same **code can be used to communicate over 0MQ and REST API**.
- The library contains **synchronous and asyncio versions** API.
- The API class stores global settings (configuration) and temporary data. Data downloaded from RE Manager (such as status, queue, lists of allowed plans and devices) is **cached locally and reloaded only if expired or changed at the server**.
- Additional API for **higher level functionality** (e.g. ‘wait_for_...’ API).

Example: Queue Server API

```
from bluesky_queueserver_api.zmq import REManagerAPI
# from bluesky_queueserver_api.http import REManagerAPI
RM = REManagerAPI()  # Parameters depend on configuration
RM.environment_open()
RM.wait_for_idle()
```

```
from bluesky_queueserver_api.aio.zmq import REManagerAPI
# from bluesky_queueserver_api.aio.http import REManagerAPI
RM = REManagerAPI()  # Parameters depend on configuration
await RM.environment_open()
await RM.wait_for_idle()
```

Queue Server API: ‘item_add’

- ‘item_add’ accepts plan parameters as JSON (dictionary) or as an instance of **BPlan or BInst helper classes**.
- There are three helper classes: **BPlan, BInst and BFunc**. The classes accept plan, instruction or function parameters in natural form and represent them as a dictionary.
- The **first parameter is the name of a plan**, instruction or function.

```
try:
    # Add a plan to the back of the queue
    response = RM.item_add(BPlan(“count”, [“det1”], num=10, delay=1))
    item_uid = response[“item”][“item_uid”]

    # Insert another plan before the previously added plan
    RM.item_add(BPlan(“count”, [“det1”] , num=10, delay=1), before_uid=item_uid)
except RM.RequestFailedError as ex:
    # < process the error >
```

Queue Server API: ‘wait_for_...’

- **‘wait_for_...’ API are monitoring status** and returns if predefined condition is met. If the condition is not met within specified timeout period, ‘WaitTimeoutError’ is raised.
- The functions accept **additional ‘monitor’ parameter**, which can be used to monitor wait time or cancel the wait by calling ‘monitor.cancel()’ (exception ‘WaitCancelError’ is raised.
- **Multiple ‘wait_for_...’ functions running in different threads are sharing status data.** No additional requests are sent to the server.

```
# Synchronous code (0MQ, HTTP)
RM.queue_start()
try:
    RM.wait_for_idle(timeout=120)  # Wait for 2 minutes
    # The queue is completed or stopped, RE Manager is idle.
except RM.WaitTimeoutError:
    # < process timeout error, RE Manager is probably not idle >

# Aynchronous code (0MQ, HTTP)
await RM.queue_start()
try:
    await RM.wait_for_idle(timeout=120)  # Wait for 2 minutes
    # The queue is completed or stopped, RE Manager is idle.
except RM.WaitTimeoutError:
    # < process timeout error, RE Manager is probably not idle >
```

Available ‘wait_for_...’ functions to support common tasks:

- ‘wait_for_idle’
- ‘wait_for_idle_or_paused’ (waiting for the queue or plan to complete)
- ‘wait_for_idle_or_running’ (opening environment with *autostart mode* enabled)

The function ‘wait_for_condition’ allows to implement **custom wait functions**.

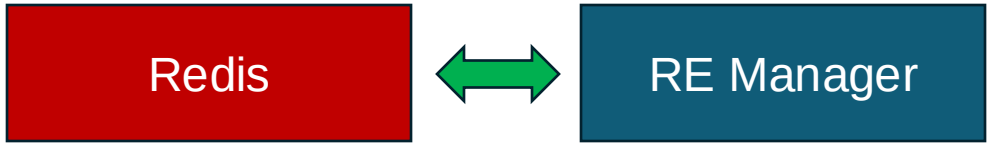
```
def condition(status):
    return status[“manager_state”] == “idle”

RM.wait_for_condition(condition, timeout=60)
```

Deployment and Maintenance

Use of Redis

- RE Manager relies on **Redis** for **persistent storage of temporary data**. The data includes contents of the queue, history and some internal parameters.
- Queue Server communicates with Redis during each queue operation, so the access is expected to be **fast and reliable**. It is recommended that Redis is installed on the same machine as the Queue Server.
- Queue Server **automatically creates all necessary Redis variables**. It successfully start with 'empty' Redis.
- Queue Server can be **directly installed on the host** (e.g. from PyPI) or **deployed in a container**.



CLI Tools

The following CLI tools are installed as **part of 'bluesky-queueserver' package**:

- start-re-manager - start RE Manager.
- qserver - communicate with RE Manager over 0MQ.
- qserver-list-plans-devices - generate list of existing plans and devices, validate startup code.
- qserver-zmq-keys - generate key pair for encryption of 0MQ control channel.
- qserver-console-monitor - simple monitor of RE Manager console output.
- qserver-clear-lock - unlock RE Manager if the lock key is lost.
- qserver-console - start Jupyter Console connected to IPython kernel running in the worker.
- qserver-qtconsole - start Jupyter Qt Console connected to IPython kernel running in the worker.

Starting the Queue Server

Developers and small users may want to start RE Manager in the terminal. Following examples illustrate how to use the most common options.

- QS (RE Manager) is started by running '***start-re-manager***'. By default, RE Manager is started in demo mode and uses startup code with simulated plans and devices distributed with the package.
- Display the available CLI options:

  start-re-manager -h
- Start in IPython kernel mode:

  start-re-manager –use-ipython-kernel=ON
- Point to a directory with startup code:

  start-re-manager –startup-dir=~/.ipython/profile_collection/startup
- Enable streaming of console output:

  start-re-manager –zmq-publish-console=ON

Running RE Manager as a Service

Instructions on how to run a Queue Server are included in documentation:
https://blueskyproject.io/bluesky-queueserver/using_queue_server.html#running-re-manager-as-a-service.

Service configuration file:

```
# File: /home/<user-name>/.config/systemd/user/queue-server.service

[Unit]
Description=Bluesky Queue Server

[Service]
ExecStart=/usr/bin/bash /home/<user-name>/queue-server.sh

[Install]
WantedBy=default.target
Alias=queue-server.service
```

Script for starting RE Manager:

```
# File: /home/<user-name>/queue-server.sh

source "/home/dgavrilov/miniconda3/etc/profile.d/conda.sh"
conda activate bs-qserver
start-re-manager --zmq-publish-console ON --console-output OFF
```

RE Manager Configuration

- Configuration options: environment variables, configuration file (YML) or CLI parameters.
- Settings from config file override environment variables, CLI parameters override all other settings.
- RE Manager attempts to load the config file if the location is specified using CLI parameter or environment variable:

start-re-manager –config=<path-to-config>
QSERVER_CONFIG=<path-to-config> start-re-manager

```
# config.yml
network:
  zmq_control_addr: tcp://*:60615
  zmq_info_addr: tcp://*:60625
  zmq_publish_console: true,
  redis_addr: localhost:6379
startup:
  keep_re: true,
  startup_dir: ~/.ipython/profile_collection/startup
  existing_plans_and_devices_path: ~/.ipython/profile_collection/startup,
  user_group_permissions_path: ~/.ipython/profile_collection/startup
  device_max_depth: 3,
operation:
  print_console_output: true,
  console_logging_level: NORMAL
  update_existing_plans_and_devices: ENVIRONMENT_OPEN,
  user_group_permissions_reload: ON_REQUEST
  emergency_lock_key: some_lock_key
```

Questions?

Starting Queue Server for the Demo

- Activate the environment with installed Queue Server packages (e.g. *bs-qserver*):

  $ conda activate bs-qserver

- Start the Run Engine Manager, enable IPython kernel mode and publishing of console output:

  $ start-re-manager –use-ipython-kernel=ON –zmq-publish-console=ON

- Open another terminal and activate the environment. Now you can explore different API requests using ***qserver*** CLI application to send requests, for example:

  $ qserver environment open
  $ qserver queue add plan '{"name": "count", "args": [["det1", "det2"]], "kwargs": {"num": 10, "delay": 1}}'
  $ qserver queue start
  $ qserver environment close

Starting *queue-monitor* GUI Application

- Open another terminal and activate the environment. Start ***queue-monitor*** application:

  $ queue-monitor

- Click 'Connect' button to start communication with RE Manager.