Backend.AI Client SDK for Python Documentation

Release 19.06.0a1

Lablup Inc.

Jun 19, 2019

TABLE OF CONTENTS

This is the documentation for the Python Client SDK which implements the Backend.AI API.

CHAPTER

ONE

REQUIREMENTS

Python 3.5.3 or higher is required.

You can download its official installer from python.org, or use a 3rd-party package/version manager such as homebrew, miniconda, or pyenv. It works on Linux, macOS, and Windows.

CHAPTER

TWO

GETTING STARTED

We recommend to create a virtual environment for isolated, unobtrusive installation of the client SDK library and tools.

```
$ python3 -m venv venv-backend-ai
$ source venv-backend-ai/bin/activate
(venv-backend-ai) $
```

Then install the client library from PyPI.

```
(venv-backend-ai) $ pip install -U pip setuptools
(venv-backend-ai) $ pip install backend.ai-client
```

Set your API keypair as environment variables:

```
(venv-backend-ai) $ export BACKEND_ACCESS_KEY=AKIA...
(venv-backend-ai) $ export BACKEND_SECRET_KEY=...
```

And then try the first commands:

```
(venv-backend-ai) $ backend.ai --help
...
(venv-backend-ai) $ backend.ai ps
...
```

Check out more details about *client configuration*, *command-line examples*, and *code examples*.

2.1 Getting Started

2.1.1 Installation

Linux/macOS

We recommend using pyenv to manage your Python versions and virtual environments to avoid conflicts with other Python applications.

Create a new virtual environment (Python 3.5.3 or higher) and activate it on your shell. Then run the following commands:

```
pip install -U pip setuptools
pip install -U backend.ai-client-py
```

Create a shell script `my-backendai-env.sh` like:

```
export BACKEND_ACCESS_KEY=...
export BACKEND_SECRET_KEY=...
export BACKEND_ENDPOINT=https://my-precious-cluster
```

Run this shell script before using `backend.ai` command.

Windows

We recommend using the Anaconda Navigator to manage your Python environments with a slick GUI app.

Create a new environment (Python 3.5.3 or higher) and launch a terminal (command prompt). Then run the following commands:

```
python -m pip install -U pip setuptools
python -m pip install -U backend.ai-client-py
```

Create a batch file `my-backendai-env.bat` like:

```
chcp 65001
set PYTHONIOENCODING=UTF-8
set BACKEND_ACCESS_KEY=...
set BACKEND_SECRET_KEY=...
set BACKEND_ENDPOINT=https://my-precious-cluster
```

Run this batch file before using `backend.ai` command.

Note that this batch file switches your command prompt to use the UTF-8 codepage for correct display of special characters in the console logs.

Verification

Run `backend.ai ps` command and check if it says "there is no compute sessions running" or something similar.

If you encounter error messages about "ACCESS_KEY", then check if your batch/shell scripts have the correct environment variable names.

If you encounter network connection error messages, check if the endpoint server is configured correctly and accessible.

2.1.2 Examples

Synchronous-mode execution

Query mode

This is the minimal code to execute a code snippet with this client SDK.

```
import sys
from ai.backend.client import Session

with Session() as session:
    kern = session.Kernel.get_or_create('python:3.6-ubuntu18.04')
    code = 'print("hello world")'
    mode = 'query'
    run_id = None
    while True:
        result = kern.execute(run_id, code, mode=mode)
        run_id = result['runId']  # keeps track of this particular run loop
        for rec in result.get('console', []):
            if rec[0] == 'stdout':
                print(rec[1], end='', file=sys.stdout)
            elif rec[0] == 'stderr':
```

(continues on next page)

(continued from previous page)

```
                print(rec[1], end='', file=sys.stderr)
            else:
                handle_media(rec)
        sys.stdout.flush()
        if result['status'] == 'finished':
            break
        else:
            mode = 'continued'
            code = ''
    kern.destroy()
```

You need to take care of `client_token` because it determines whether to reuse kernel sessions or not. Backend.AI cloud has a timeout so that it terminates long-idle kernel sessions, but within the timeout, any kernel creation requests with the same `client_token` let Backend.AI cloud to reuse the kernel.

Batch mode

You first need to upload the files after creating the session and construct a `opts` struct.

```
import sys
from ai.backend.client import Session

with Session() as session:
    kern = session.Kernel.get_or_create('python:3.6-ubuntu18.04')
    kern.upload(['mycode.py', 'setup.py'])
    code = ''
    mode = 'batch'
    run_id = None
    opts = {
        'build': '*',  # calls "python setup.py install"
        'exec': 'python mycode.py arg1 arg2',
    }
    while True:
        result = kern.execute(run_id, code, mode=mode, opts=opts)
        opts.clear()
        run_id = result['runId']
        for rec in result.get('console', []):
            if rec[0] == 'stdout':
                print(rec[1], end='', file=sys.stdout)
            elif rec[0] == 'stderr':
                print(rec[1], end='', file=sys.stderr)
            else:
                handle_media(rec)
        sys.stdout.flush()
        if result['status'] == 'finished':
            break
        else:
            mode = 'continued'
            code = ''
    kern.destroy()
```

Handling user inputs

Inside the while-loop for `kern.execute()` above, change the if-block for `result['status']` as follows:

```
...
if result['status'] == 'finished':
    break
```

(continues on next page)

(continued from previous page)

```
elif result['status'] == 'waiting-input':
    mode = 'input'
    if result['options'].get('is_password', False):
        code = getpass.getpass()
    else:
        code = input()
else:
    mode = 'continued'
    code = ''
...
```

A common gotcha is to miss setting `mode = 'input'`. Be careful!

Handling multi-media outputs

The `handle_media()` function used above examples would look like:

```
def handle_media(record):
    media_type = record[0]  # MIME-Type string
    media_data = record[1]  # content
    ...
```

The exact method to process `media_data` depends on the `media_type`. Currently the following behaviors are well-defined:

- For (binary-format) images, the content is a dataURI-encoded string.
- For SVG (scalable vector graphics) images, the content is an XML string.
- For `application/x-sorna-drawing`, the content is a JSON string that represents a set of vector drawing commands to be replayed the client-side (e.g., Javascript on browsers)

Asynchronous-mode Execution

The async version has all sync-version interfaces as coroutines but comes with additional features such as `stream_execute()` which streams the execution results via websockets and `stream_pty()` for interactive terminal streaming.

```
import asyncio
import json
import sys
import aiohttp
from ai.backend.client import AsyncSession

async def main():
    async with AsyncSession() as session:
        kern = await session.Kernel.get_or_create('python:3.6-ubuntu18.04',
                                                  client_token='mysession')
        code = 'print("hello world")'
        mode = 'query'
        async with kern.stream_execute(code, mode=mode) as stream:
            # no need for explicit run_id since WebSocket connection represents it!
            async for result in stream:
                if result.type != aiohttp.WSMsgType.TEXT:
                    continue
                result = json.loads(result.data)
                for rec in result.get('console', []):
                    if rec[0] == 'stdout':
                        print(rec[1], end='', file=sys.stdout)
```

(continues on next page)

(continued from previous page)

```
                    elif rec[0] == 'stderr':
                        print(rec[1], end='', file=sys.stderr)
                    else:
                        handle_media(rec)
                sys.stdout.flush()
                if result['status'] == 'finished':
                    break
                elif result['status'] == 'waiting-input':
                    mode = 'input'
                    if result['options'].get('is_password', False):
                        code = getpass.getpass()
                    else:
                        code = input()
                    await stream.send_text(code)
                else:
                    mode = 'continued'
                    code = ''
        await kern.destroy()

loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(main())
finally:
    loop.close()
```

New in version 1.5.

2.1.3 Client Session

This module is the first place to begin with your Python programs that use Backend.AI API functions.

The high-level API functions cannot be used alone – you must initiate a client session first because each session provides *proxy attributes* that represent API functions and run on the session itself.

To achieve this, during initialization session objects internally construct new types by combining the *BaseFunction* class with the attributes in each API function classes, and makes the new types bound to itself. Creating new types every time when creating a new session instance may look weird, but it is the most convenient way to provide *class-methods* in the API function classes to work with specific *session instances*.

When designing your application, please note that session objects are intended to live long following the process' lifecycle, instead of to be created and disposed whenever making API requests.

class ai.backend.client.session.**BaseSession**(*, *config=None*)
The base abstract class for sessions.

class ai.backend.client.session.**Session**(*, *config=None*)
An API client session that makes API requests synchronously. You may call (almost) all function proxy methods like a plain Python function. It provides a context manager interface to ensure closing of the session upon errors and scope exits.

Admin
The *Admin* function proxy bound to this session.

Agent
The *Agent* function proxy bound to this session.

Domain
The Domain function proxy bound to this session.

Group
The Group function proxy bound to this session.

Image
The `Image` function proxy bound to this session.

Kernel
The *Kernel* function proxy bound to this session.

KeyPair
The *KeyPair* function proxy bound to this session.

Manager
The *Manager* function proxy bound to this session.

Resource
The `Resource` function proxy bound to this session.

KeypairResourcePolicy
The `KeypairResourcePolicy` function proxy bound to this session.

User
The `User` function proxy bound to this session.

VFolder
The *VFolder* function proxy bound to this session.

close()
Terminates the session. It schedules the `close()` coroutine of the underlying aiohttp session and then enqueues a sentinel object to indicate termination. Then it waits until the worker thread to self-terminate by joining.

class ai.backend.client.session.**AsyncSession**(*, *config=None*)
An API client session that makes API requests asynchronously using coroutines. You may call all function proxy methods like a coroutine. It provides an async context manager interface to ensure closing of the session upon errors and scope exits.

Admin
The *Admin* function proxy bound to this session.

Agent
The *Agent* function proxy bound to this session.

Group
The `Group` function proxy bound to this session.

Image
The `Image` function proxy bound to this session.

Kernel
The *Kernel* function proxy bound to this session.

KeyPair
The *KeyPair* function proxy bound to this session.

Manager
The *Manager* function proxy bound to this session.

Resource
The `Resource` function proxy bound to this session.

KeypairResourcePolicy
The `KeypairResourcePolicy` function proxy bound to this session.

User
The `User` function proxy bound to this session.

VFolder
The *VFolder* function proxy bound to this session.

2.1.4 Client Configuration

The configuration for Backend.AI API includes the endpoint URL prefix, API keypairs (access and secret keys), and a few others.

There are two ways to set the configuration:

1. Setting environment variables before running your program that uses this SDK.
2. Manually creating *APIConfig* instance and creating sessions with it.

The list of supported environment variables are:

- BACKEND_ENDPOINT
- BACKEND_ACCESS_KEY
- BACKEND_SECRET_KEY
- BACKEND_VFOLDER_MOUNTS

Other configurations are set to defaults.

Note that when you use our client-side Jupyter integration, BACKEND_VFOLDER_MOUNTS is the only way to attach your virtual folders to the notebook kernels.

ai.backend.client.config.**get_env**(*key*, *default=None*, *, *clean=<function <lambda>>*)
Retrieves a configuration value from the environment variables. The given *key* is uppercased and prefixed by "BACKEND_" and then "SORNA_" if the former does not exist.

Parameters

- **key** (str) – The key name.
- **default** (Optional[Any]) – The default value returned when there is no corresponding environment variable.
- **clean** (Callable[[str], Any]) – A single-argument function that is applied to the result of lookup (in both successes and the default value for failures). The default is returning the value as-is.

Returns The value processed by the *clean* function.

ai.backend.client.config.**get_config**()
Returns the configuration for the current process. If there is no explicitly set *APIConfig* instance, it will generate a new one from the current environment variables and defaults.

ai.backend.client.config.**set_config**(*conf*)
Sets the configuration used throughout the current process.

class ai.backend.client.config.**APIConfig**(*, *endpoint=None*, *domain=None*, *group=None*, *version=None*, *user_agent=None*, *access_key=None*, *secret_key=None*, *hash_type=None*, *vfolder_mounts=None*, *skip_sslcert_validation=None*)
Represents a set of API client configurations. The access key and secret key are mandatory – they must be set in either environment variables or as the explicit arguments.

Parameters

- **endpoint** (Union[str, URL, None]) – The URL prefix to make API requests via HTTP/HTTPS.
- **version** (Optional[str]) – The API protocol version.
- **user_agent** (Optional[str]) – A custom user-agent string which is sent to the API server as a User-Agent HTTP header.

- **access_key** (Optional[str]) – The API access key. If deliberately set to an empty string, the API requests will be made without signatures (anonymously).
- **secret_key** (Optional[str]) – The API secret key.
- **hash_type** (Optional[str]) – The hash type to generate per-request authentication signatures.
- **vfolder_mounts** (Optional[Iterable[str]]) – A list of vfolder names (that must belong to the given access key) to be automatically mounted upon any Kernel.get_or_create() calls.

DEFAULTS = {'domain': 'default', 'endpoint': 'https://api.backend.ai', 'group': 'de
The default values except the access and secret keys.

2.2 Command-line Interface

2.2.1 Examples

Note: Please consult the detailed usage in the help of each command (use -h or --help argument to display the manual).

Listing currently running sessions

```
backend.ai ps
```

This command is actually an alias of the following command:

```
backend.ai admin sessions
```

Running simple sessions

The following command spawns a Python session and executes the code passed as -c argument immediately. --rm option states that the client automatically terminates the session after execution finishes.

```
backend.ai run --rm -c 'print("hello world")' python
```

The following command spawns a Python session and execute the code passed as ./myscript.py file, using the shell command specified in the --exec option.

```
backend.ai run --rm --exec 'python myscript.py arg1 arg2' \
           python ./myscript.py
```

Running sessions with accelerators

The following command spawns a Python TensorFlow session using a half of virtual GPU device and executes ./mygpucode.py file inside it.

```
backend.ai run --rm -r gpu=0.5 \
           python-tensorflow ./mygpucode.py
```

Terminating running sessions

Without `--rm` option, your session remains alive for a configured amount of idle timeout (default is 30 minutes). You can see such sessions using the `backend.ai ps` command. Use the following command to manually terminate them via their session IDs. You may specifcy multiple session IDs to terminate them at once.

```
backend.ai rm <sessionID>
```

Starting a session and connecting to its Jupyter Notebook

The following command first spawns a Python session named "mysession" without running any code immediately, and then executes a local proxy which connects to the "jupyter" service running inside the session via the local TCP port 9900. The `start` command shows application services provided by the created compute session so that you can choose one in the subsequent `app` command.

```
backend.ai start -t mysession python
backend.ai app -p 9900 mysession jupyter
```

Once executed, the `app` command waits for the user to open the displayed address using appropriate application. For the jupyter service, use your favorite web browser just like the way you use Jupyter Notebooks. To stop the `app` command, press `Ctrl+C` or send the `SIGINT` signal.

Running sessions with vfolders

The following command creates a virtual folder named "mydata1", and then uploads `./bigdata.csv` file into it.

```
backend.ai vfolder create mydata1
backend.ai vfolder upload mydata1 ./bigdata.csv
```

The following command spawns a Python session where the virtual folder "mydata1" is mounted. The execution options are omitted in this example. Then, it downloads `./bigresult.txt` file (generated by your code) from the "mydata1" virtual folder.

```
backend.ai run --rm -m mydata1 python ...
backend.ai vfolder download mydata1 ./bigresult.txt
```

In your code, you may access the virtual folder via `/home/work/mydata1` (where the default current working directory is `/home/work`) just like a normal directory.

Running parallel experiment sessions

(TODO)

2.3 High-level Function Reference

2.3.1 Admin Functions

class ai.backend.client.admin.**Admin**

Provides the function interface for making admin GrapQL queries.

> **Note:** Depending on the privilege of your API access key, you may or may not have access to querying/mutating server-side resources of other users.

session = None
The client session instance that this function class is bound to.

2.3.2 Agent Functions

class ai.backend.client.agent.**Agent**
Provides a shortcut of `Admin.query()` that fetches various agent information.

Note: All methods in this function class require your API access key to have the *admin* privilege.

session = None
The client session instance that this function class is bound to.

2.3.3 Kernel Functions

class ai.backend.client.kernel.**Kernel**(*kernel_id*, *owner_access_key=None*)
Provides various interactions with compute sessions in Backend.AI.

The term 'kernel' is now deprecated and we prefer 'compute sessions'. However, for historical reasons and to avoid confusion with client sessions, we keep the backward compatibility with the naming of this API function class.

For multi-container sessions, all methods take effects to the master container only, except `destroy()` and `restart()` methods. So it is the user's responsibility to distribute uploaded files to multiple containers using explicit copies or virtual folders which are commonly mounted to all containers belonging to the same compute session.

session = None
The client session instance that this function class is bound to.

stream_pty()
Opens a pseudo-terminal of the kernel (if supported) streamed via websockets.

Return type *StreamPty*

Returns a *StreamPty* object.

stream_execute(*code=''*, ***, *mode='query'*, *opts=None*)
Executes a code snippet in the streaming mode. Since the returned websocket represents a run loop, there is no need to specify *run_id* explicitly.

Return type *WebSocketResponse*

class ai.backend.client.kernel.**StreamPty**(*session*, *underlying_ws*)
A derivative class of *WebSocketResponse* which provides additional functions to control the terminal.

2.3.4 KeyPair Functions

class ai.backend.client.keypair.**KeyPair**(*access_key*)
Provides interactions with keypairs.

session = None
The client session instance that this function class is bound to.

2.3.5 Manager Functions

class ai.backend.client.manager.**Manager**
Provides controlling of the gateway/manager servers.

New in version 18.12.

session = None
The client session instance that this function class is bound to.

2.3.6 Virtual Folder Functions

class ai.backend.client.vfolder.**VFolder**(*name*)

session = None
The client session instance that this function class is bound to.

2.4 Low-level SDK Reference

2.4.1 Base Function

This module defines a few utilities that ease complexities to support both synchronous and asynchronous API functions, using some tricks with Python metaclasses.

Unless your are contributing to the client SDK, probably you won't have to use this module directly.

class ai.backend.client.base.**APIFunctionMeta**(*name*, *bases*, *attrs*, ***kwargs*)
Converts all methods marked with *api_function()* into session-aware methods that are either plain Python functions or coroutines.

mro() → list
return a type's method resolution order

class ai.backend.client.base.**BaseFunction**
The class used to build API functions proxies bound to specific session instances.

@ai.backend.client.base.**api_function**(*meth*)
Mark the wrapped method as the API function method.

2.4.2 Request API

This module provides low-level API request/response interfaces based on aiohttp.

Depending on the session object where the request is made from, *Request* and *Response* differentiate their behavior: works as plain Python functions or returns awaitables.

class ai.backend.client.request.**Request**(*session*, *method='GET'*, *path=None*, *content=None*, ***, *content_type=None*, *params=None*, *reporthook=None*)
The API request object.

with async with fetch(***kwargs*) **as Response**
Sends the request to the server and reads the response.

You may use this method either with plain synchronous Session or AsyncSession. Both the followings patterns are valid:

```
from ai.backend.client.request import Request
from ai.backend.client.session import Session

with Session() as sess:
  rqst = Request(sess, 'GET', ...)
  with rqst.fetch() as resp:
    print(resp.text())
```

```
from ai.backend.client.request import Request
from ai.backend.client.session import AsyncSession

async with AsyncSession() as sess:
  rqst = Request(sess, 'GET', ...)
  async with rqst.fetch() as resp:
    print(await resp.text())
```

Return type *FetchContextManager*

async with connect_websocket(***kwargs*) **as WebSocketResponse or its derivatives**

Creates a WebSocket connection.

> **Warning:** This method only works with *AsyncSession*.

Return type *WebSocketContextManager*

set_content(*value*, *, *content_type=None*)

Sets the content of the request.

set_json(*value*)

A shortcut for set_content() with JSON objects.

attach_files(*files*)

Attach a list of files represented as AttachedFile.

class ai.backend.client.request.**Response**(*session*, *underlying_response*, *, *async_mode=False*)

Represents the Backend.AI API response. Also serves as a high-level wrapper of `aiohttp.ClientResponse`.

The response objects are meant to be created by the SDK, not the callers.

`text()`, `json()` methods return the resolved content directly with plain synchronous Session while they return the coroutines with AsyncSession.

class ai.backend.client.request.**WebSocketResponse**(*session*, *underlying_ws*)

A high-level wrapper of `aiohttp.ClientWebSocketResponse`.

class ai.backend.client.request.**FetchContextManager**(*session*, *rqst_ctx*, *, *response_cls=<class 'ai.backend.client.request.Response'>*, *check_status=True*)

The context manager returned by *Request.fetch()*.

It provides both synchronouse and asynchronous contex manager interfaces.

class ai.backend.client.request.**WebSocketContextManager**(*session*, *ws_ctx*, *, *on_enter=None*, *response_cls=<class 'ai.backend.client.request.WebSocketResponse'*

The context manager returned by *Request.connect_websocket()*.

class ai.backend.client.request.**AttachedFile**(*filename*, *stream*, *content_type*)

A struct that represents an attached file to the API request.

Parameters

- **filename** (*str*) – The name of file to store. It may include paths and the server will create parent directories if required.
- **stream** (*Any*) – A file-like object that allows stream-reading bytes.

- **content_type** (*str*) – The content type for the stream. For arbitrary binary data, use "application/octet-stream".

count(*value*) → integer – return number of occurrences of value

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

2.4.3 Exceptions

class ai.backend.client.exceptions.**BackendError**
Exception type to catch all ai.backend-related errors.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class ai.backend.client.exceptions.**BackendAPIError**(*status*, *reason*, *data*)
Exceptions returned by the API gateway.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class ai.backend.client.exceptions.**BackendClientError**
Exceptions from the client library, such as argument validation errors.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX