Backend.AI Client SDK for Python Documentation

Release 19.03.0b4

Lablup Inc.

Feb 14, 2019

Table of contents

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

CHAPTER 1

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 2

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('lua')
```

(continues on next page)

(continued from previous page)

```
    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':
                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')
    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
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 aiohttp
from ai.backend.client import AsyncSession

async def main():
    async with AsyncSession() as session:
        kern = await session.Kernel.get_or_create('lua5', 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
```

(continues on next page)

(continued from previous page)

```
                result = json.loads(result.data)
                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
                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.

abstractmethod close()

Terminates the session and releases underlying resources.

closed

Checks if the session is closed.

Return type bool

config

The configuration used by this session object.

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.

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.

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.

worker_thread
The thread that internally executes the asynchronous implementations of the given API functions.

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.

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.

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

await close()
Terminates the session and releases underlying resources.

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*, *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[URL, str, 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.
- **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 = {'endpoint': 'https://api.backend.ai', 'hash_type': 'sha256', 'version':
The default values except the access and secret keys.

endpoint
The configured endpoint URL prefix.

Return type URL

user_agent
The configured user agent string.

Return type str

access_key
The configured API access key.

Return type str

secret_key
The configured API secret key.

Return type str

version
The configured API protocol version.

Return type str

hash_type
The configured hash algorithm for API authentication signatures.

Return type str

vfolder_mounts
The configured auto-mounted vfolder list.

Return type Tuple[str, ...]

skip_sslcert_validation
Whether to skip SSL certificate validation for the API gateway.

Return type bool

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.

classmethod await query(*query*, *variables=None*)
Sends the GraphQL query and returns the response.

Parameters

- **query** (str) – The GraphQL query string.
- **variables** (Optional[Mapping[str, Any]]) – An optional key-value dictionary to fill the interpolated template variables in the query.

Return type Any

Returns The object parsed from the response JSON string.

2.3.2 Agent Functions

class ai.backend.client.agent.**Agent**(*agent_id*)
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.

classmethod await list(*status='ALIVE'*, *fields=None*)
Fetches the list of agents with the given status.

Parameters

- **status** (str) – An upper-cased string constant representing agent status (one of 'ALIVE', 'TERMINATED', 'LOST', etc.)
- **fields** (Optional[Iterable[str]]) – Additional per-agent query fields to fetch.

Return type Sequence[dict]

await info(*fields=None*)
Returns the agent's information including resource capacity and usage.

New in version 18.12.

Return type dict

2.3.3 Kernel Functions

class ai.backend.client.kernel.**Kernel**(*kernel_id*)
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.

classmethod await get_or_create(*lang*, ***, *client_token=None*, *mounts=None*, *envs=None*, *resources=None*, *cluster_size=1*, *tag=None*)
Get-or-creates a compute session. If *client_token* is None, it creates a new compute session as long as the server has enough resources and your API key has remaining quota. If *client_token* is a valid string and there is an existing compute session with the same token and the same *lang*, then it returns the *Kernel* instance representing the existing session.

Parameters

- **lang** (str) – The image name and tag for the compute session. Example: python:3.6-ubuntu. Check out the full list of available images in your server using (TODO: new API).
- **client_token** (Optional[str]) – A client-side identifier to seamlessly reuse the compute session already created.
- **mounts** (Optional[Iterable[str]]) – The list of vfolder names that belongs to the currrent API access key.
- **envs** (Optional[Mapping[str, str]]) – The environment variables which always bypasses the jail policy.
- **resources** (Optional[Mapping[str, int]]) – The resource specification. (TODO: details)
- **cluster_size** (int) – The number of containers in this compute session. Must be at least 1.
- **tag** (Optional[str]) – An optional string to annotate extra information.

Return type *Kernel*

Returns The *Kernel* instance.

await destroy()
Destroys the compute session. Since the server literally kills the container(s), all ongoing executions are forcibly interrupted.

await restart()
Restarts the compute session. The server force-destroys the current running container(s), but keeps their temporary scratch directories intact.

await interrupt()
Tries to interrupt the current ongoing code execution. This may fail without any explicit errors depending on the code being executed.

await complete(*code*, *opts=None*)
Gets the auto-completion candidates from the given code string, as if a user has pressed the tab key just after the code in IDEs.

Depending on the language of the compute session, this feature may not be supported. Unsupported sessions returns an empty list.

Parameters

- **code** (str) – An (incomplete) code text.
- **opts** (Optional[dict]) – Additional information about the current cursor position, such as row, col, line and the remainder text.

Return type Iterable[str]

Returns An ordered list of strings.

await get_info()
Retrieves a brief information about the compute session.

await get_logs()
Retrieves the console log of the compute session container.

await execute(*run_id=None*, *code=None*, *mode='query'*, *opts=None*)
Executes a code snippet directly in the compute session or sends a set of build/clean/execute commands to the compute session.

For more details about using this API, please refer the official API documentation.

Parameters

- **run_id** (Optional[str]) – A unique identifier for a particular run loop. In the first call, it may be None so that the server auto-assigns one. Subsequent calls must use the returned runId value to request continuation or to send user inputs.
- **code** (Optional[str]) – A code snippet as string. In the continuation requests, it must be an empty string. When sending user inputs, this is where the user input string is stored.
- **mode** (str) – A constant string which is one of "query", "batch", "continue", and "user-input".
- **opts** (Optional[dict]) – A dict for specifying additional options. Mainly used in the batch mode to specify build/clean/execution commands. See the API object reference for details.

Returns An execution result object

await upload(*files*, *basedir=None*, *show_progress=False*)
Uploads the given list of files to the compute session. You may refer them in the batch-mode execution or from the code executed in the server afterwards.

Parameters

- **files** (Sequence[Union[str, Path]]) – The list of file paths in the client-side. If the paths include directories, the location of them in the compute session is calculated from the relative path to *basedir* and all intermediate parent directories are automatically created if not exists.

For example, if a file path is `/home/user/test/data.txt` (or `test/data.txt`) where *basedir* is `/home/user` (or the current working directory is `/home/user`), the uploaded file is located at `/home/work/test/data.txt` in the compute session container.

- **basedir** (Union[str, Path, None]) – The directory prefix where the files reside. The default value is the current working directory.
- **show_progress** (bool) – Displays a progress bar during uploads.

await download(*files*, *dest='.'*, *show_progress=False*)
Downloads the given list of files from the compute session.

Parameters

- **files** (Sequence[Union[str, Path]]) – The list of file paths in the compute session. If they are relative paths, the path is calculated from `/home/work` in the compute session container.
- **dest** (Union[str, Path]) – The destination directory in the client-side.
- **show_progress** (bool) – Displays a progress bar during downloads.

await list_files(*path='.'*)
Gets the list of files in the given path inside the compute session container.

Parameters **path** (Union[str, Path]) – The directory path in the compute session.

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.

classmethod await create(*user_id*, *is_active=True*, *is_admin=False*, *resource_policy=None*, *rate_limit=None*, *concurrency_limit=None*, *fields=None*)
Creates a new keypair with the given options. You need an admin privilege for this operation.

Return type dict

classmethod await list(*user_id=None*, *is_active=None*, *fields=None*)
Lists the keypairs. You need an admin privilege for this operation.

Return type Sequence[dict]

await info(*fields=None*)
Returns the keypair's information such as resource limits.

Parameters **fields** (Optional[Iterable[str]]) – Additional per-agent query fields to fetch.

New in version 18.12.

Return type dict

await activate()
Activates this keypair. You need an admin privilege for this operation.

await deactivate()
Deactivates this keypair. Deactivated keypairs cannot make any API requests unless activated again by an administrator. You need an admin privilege for this operation.

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.

classmethod await status()
Returns the current status of the configured API server.

classmethod await freeze(*force_kill=False*)
Freezes the configured API server. Any API clients will no longer be able to create new compute sessions nor create and modify vfolders/keypairs/etc. This is used to enter the maintenance mode of the server for unobtrusive manager and/or agent upgrades.

Parameters **force_kill** (bool) – If set True, immediately shuts down all running compute sessions forcibly. If not set, clients who have running compute session are still able to interact with them though they cannot create new compute sessions.

classmethod await unfreeze()
Unfreezes the configured API server so that it resumes to normal operation.

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.

classmethod await create(*name*, *host=None*)

classmethod await list()

await info()

await delete()

await upload(*files*, *basedir=None*, *show_progress=False*)

await mkdir(*path*)

await delete_files(*files*, *recursive=False*)

await download(*files*, *show_progress=False*)

await list_files(*path='.'*)

await invite(*perm*, *emails*)

classmethod await invitations()

classmethod await accept_invitation(*inv_id*, *inv_ak*)

classmethod await delete_invitation(*inv_id*)

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.

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*

content

Retrieves the content in the original form. Private codes should NOT use this as it incurs duplicate encoding/decoding.

Return type Union[bytes, bytearray, str, StreamReader, IOBase, None]

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".

2.4.3 Exceptions

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

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

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

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index