apiwrappers

Aleksei Maslakov

Jan 15, 2022

CONTENTS

apiwrappers is a library for building API clients that work both with regular and async code.

CHAPTER ONE

FEATURES

- **DRY** - support both regular and async code with one implementation
- **Flexible** - middleware mechanism to customize request/response
- **Typed** - library is fully typed and it’s relatively easy to get fully typed wrappers
- **Modern** - decode JSON with no effort using dataclasses and type annotations
- **Unified interface** - work with different python HTTP client libraries in the same way. Currently supported:
  - requests
  - aiohttp

CHAPTER TWO

INSTALLATION

```
pip install 'apiwrappers[aiohttp,requests]'
```

Note: extras are mainly needed for the final user of your API client

CHAPTER

THREE

GETTING STARTED

With *apiwrappers* you can bootstrap clients for different API pretty fast and easily.

Here is how a typical API client would look like:

```python
from __future__ import annotations

from dataclasses import dataclass
from typing import Awaitable, Generic, List, TypeVar, overload

from apiwrappers import AsyncDriver, Driver, Request, Url, fetch

T = TypeVar("T", Driver, AsyncDriver)


@dataclass
class Repo:
    id: int
    name: str


class Github(Generic[T]):
    def __init__(self, host: str, driver: T):
        self.url = Url(host)
        self.driver: T = driver

    @overload
    def get_repos(
        self: Github[Driver], username: str
    ) -> List[Repo]:
        ...

    @overload
    def get_repos(
        self: Github[AsyncDriver], username: str
    ) -> Awaitable[List[Repo]]:
        ...

    def get_repos(self, username: str):
        url = self.url("/users/{username}/repos", username=username)
        request = Request("GET", url)
        return fetch(self.driver, request, model=List[Repo])
```

This is small, but fully typed, API client for one of the api.github.com endpoints to get all user repos by username:

Here we defined Repo dataclass that describes what we want to get from response and pass it to the *fetch()* function. *fetch()* will then make a request and cast response to that type.

And here how we can use it:

```
>>> from apiwrappers import make_driver
>>> driver = make_driver("requests")
>>> github = Github("https://api.github.com", driver=driver)
>>> github.get_repos("unmade")
[Repo(id=47463599, name='am-date-picker'),
 Repo(id=231653904, name='apiwrappers'),
 Repo(id=144204778, name='conway'),
 ...
]
```

To use it with asyncio all we need to do is provide a proper driver and don't forget to `await` method call:

Use IPython or Python 3.8+ with python -m asyncio to try this code interactively

```
>>> from apiwrappers import make_driver
>>> driver = make_driver("aiohttp")
>>> github = Github("https://api.github.com", driver=driver)
>>> await github.get_repos("unmade")
[Repo(id=47463599, name='am-date-picker'),
 Repo(id=231653904, name='apiwrappers'),
 Repo(id=144204778, name='conway'),
 ...
]
```

3.1 Table of Contents

3.1.1 Building an API Client

This page will walk you through steps on how to build wrapper for API.

Making a Request

Each wrapper needs a HTTP client to make requests to the API.

You can easily use one of the *drivers* to make requests, but *Driver.fetch()* call returns a *Response* object, which is not always suitable for building good API clients.

For API client it can be better to return typed data, such as dataclasses, than let the final user deal with json.

apiwrappers provides a *fetch()* function, which takes driver as a first argument, and all other arguments are the same as with *Driver.fetch()*. Giving that, it behaves exactly like if you are working with driver:

```
>>> from apiwrappers import Request, fetch, make_driver
>>> driver = make_driver("requests")
>>> request = Request("GET", "https://example.org")
>>> response = fetch(driver, request)
<Response [200]>
```

You can also provide two additional arguments:

- `model` - a type or factory function that describes response structure.
- `source` - optional key name in the json, which value will be passed to the `model`. You may use dotted notation to traverse keys - `key1.key2`

With these arguments, *fetch()* function acts like a factory, returning new instance of the type provided to the `model` argument:

```
from dataclasses import dataclass
from typing import List

from apiwrappers import Request, fetch, make_driver

@dataclass
class Repo:
    name: str

url = "https://api.github.com/users/unmade/repos"
request = Request("GET", url)

driver = make_driver("requests")
fetch(driver, request, model=List[Repo])  # [Repo(name='am-date-picker'), ...]
fetch(driver, request, model=Repo, source="0")  # Repo(name='am-date-picker')
fetch(driver, request, model=str, source="0.name")  # 'am-date-picker'

driver = make_driver("aiohttp")
await fetch(driver, request, model=List[Repo])  # [Repo(name='am-date-picker'), ...]
await fetch(driver, request, model=Repo, source="0")  # Repo(name='am-date-picker')
await fetch(driver, request, model=str, source="0.name")  # 'am-date-picker'
```

Writing a Simple API Client

Now that we know how to make requests and how to get data from response, lets write API client class:

```
from dataclasses import dataclass
from typing import List

from apiwrappers import Request, Url, fetch


@dataclass
class Repo:
    id: int
    name: str


class GitHub:
    def __init__(self, host, driver):
        self.url = Url(host)
        self.driver = driver

    def get_repos(self, username):
```

(continues on next page)

(continued from previous page)

```
        url = self.url("/users/{username}/repos", username=username)
        request = Request("GET", url)
        return fetch(self.driver, request, model=List[Repo])
```

Here we defined `.get_repos` method to get all user's repos. Based on the driver this method returns either a `List[Repo]` or a coroutine - `Awaitable[List[Repo]]`

You never want to await the fetch call here, just return it immediately and let the final user await it if needed

Another thing to notice is how we create URL:

```
url = self.url("/users/{username}/repos", username=username)
```

Sometimes, it's useful to have an URL template, for example, for logging or for aggregating metrics, so instead of formatting immediately, we provide a template and replacement fields.

The wrapper above is good enough to satisfy most cases, however it lacks one of the important features nowadays - type annotations.

Adding Type Annotations

In the example above, we didn't add any type annotations for `.get_repos` method.

We can simply specify return type as:

```
Union[List[Repo], Awaitable[List[Repo]]
```

and that will be enough to have a good auto-completion, but what we want precise type annotations.

We want to tell mypy, that when driver corresponds to *`Driver`* protocol `.get_repos` has return type `List[Repo]` and for *`AsyncDriver`* protocol - `Awaitable[List[Repo]]`.

It can be done like that:

```
from __future__ import annotations

from dataclasses import dataclass
from typing import Awaitable, Generic, List, TypeVar, overload

from apiwrappers import AsyncDriver, Driver, Request, Url, fetch

T = TypeVar("T", Driver, AsyncDriver)


@dataclass
class Repo:
    id: int
    name: str


class GitHub(Generic[T]):
    def __init__(self, host: str, driver: T):
        self.url = Url(host)
        self.driver: T = driver
```

(continues on next page)

(continued from previous page)

```
    @overload
    def get_repos(
        self: GitHub[Driver], username: str
    ) -> List[Repo]:
        ...

    @overload
    def get_repos(
        self: GitHub[AsyncDriver], username: str
    ) -> Awaitable[List[Repo]]:
        ...

    def get_repos(self, username: str):
        url = self.url("/users/{username}/repos", username=username)
        request = Request("GET", url)
        return fetch(self.driver, request, model=List[Repo])
```

Here, we defined a `T` type variable, constrained to *`Driver`* and *`AsyncDriver`* protocols. Our wrapper is now a generic class of that variable. We also used `overload` with self-type to define return type based on the driver provided to our wrapper.

Using the API Client

Here is how we can use our client:

```
>>> from apiwrappers import make_driver
>>> driver = make_driver("requests")
>>> github = GitHub("https://api.github.com", driver=driver)
>>> github.get_repos("unmade")
[Repo(id=47463599, name='am-date-picker'),
 ...
]
```

Or to use it asynchronously:

```
>>> from apiwrappers import make_driver
>>> driver = make_driver("aiohttp")
>>> github = GitHub("https://api.github.com", driver=driver)
>>> await github.get_repos("unmade")
[Repo(id=47463599, name='am-date-picker'),
 ...
]
```

3.1.2 Drivers

Drivers are essentially adapters for different python HTTP client libraries.

This page will walk you through the concept of drivers in the *apiwrappers* library.

Basic Usage

Out of the box *apiwrappers* provides drivers for requests and aiohttp libraries.

You can create them with a `make_driver()` factory. Let's learn how to make a simple request using a driver for requests library:

```
>>> from apiwrappers import Request, make_driver
>>> driver = make_driver("requests")
>>> request = Request("GET", "https://example.org")
>>> response = driver.fetch(request)
>>> response
<Response [200]>
>>> response.status_code
200
>>> response.headers["content-type"]
'text/html; charset=UTF-8'
>>> response.text()
'<!doctype html>\n<html>\n<head>\n<title>Example Domain...'
```

Or using driver for aiohttp:

Use IPython or Python 3.8+ with python -m asyncio to try this code interactively

```
>>> from apiwrappers import Request, make_driver
>>> driver = make_driver("aiohttp")
>>> request = Request("GET", "https://example.org")
>>> response = await driver.fetch(request)
>>> response
<Response [200]>
```

As you see, some drivers can be used regularly, while others - asynchronously. It is also better to think of what structural protocol particular driver follows, rather than what library it uses underneath.

Driver protocols

All drivers should follow either `Driver` or `AsyncDriver` protocols, depending on which HTTP client is used.Protocols also help to abstract away from concrete driver implementations and ease type checking and annotation.

Timeouts

You can set timeouts in seconds when creating a driver or when making a request. The later will take precedences over driver settings.

By default timeout is `30 seconds`.

Here is how you can change it:

```python
from datetime import timedelta

from apiwrappers import make_driver


driver = make_driver("requests", timeout=5)

# making a request with timeout set to 5 seconds
driver.fetch(request)

# making a request with timeout set to 2.5 seconds
driver.fetch(request, timeout=2.5)

# or more explicitly
driver.fetch(request, timeout=timedelta(minutes=1))

# timeout is disabled, wait infinitely
driver.fetch(request, timeout=None)
```

In case timeout value is exceeded `Timeout` error will be raised

SSL Verification

You can enable/disable SSL verification or provide custom SSL certs upon driver instantiation. Default CA bundle provided by certifi library.

By default SSL verification is enabled.

Here is how you can change it:

```python
from apiwrappers import make_driver

# disable SSL verification
driver = make_driver("requests", verify=False)

# custom SSL with trusted CAs
driver = make_driver("requests", verify="/path/to/ca-bundle.crt")

# custom Client Side Certificates
certs = ('/path/to/client.cert', '/path/to/client.key')
driver = make_driver("requests", cert=certs)
```

Writing your own driver

To write a driver you don't need to subclass anything and have a lot of freedom. You can write however you want, the key thing is to follow one of the protocols.

3.1.3 Authentication

This page describes how you can use various kinds of authentication with *apiwrappers*.

Basic Authentication

Making request with HTTP Basic Auth is rather straightforward:

```
from apiwrappers import Request

Request(..., auth=("user", "pass"))
```

Token Authentication

To make a request with a Token Based Authentication:

```
from apiwrappers import Request
from apiwrappers.auth import TokenAuth

Request(..., auth=TokenAuth("your_token", kind="JWT"))
```

Api key Authentication

To make a request with a Api key Based Authentication:

```
from apiwrappers import Request
from apiwrappers.auth import ApiKeyAuth

Request(..., auth=ApiKeyAuth("your_key", header="X-Api-Key"))
```

Custom Authentication

You can add your own authentication mechanism relatively easy.

If you don't need to make any external calls, then you can define a callable that returns a dictionary with authorization headers.

For example, this is simple authentication class:

```
from typing import Dict


class ProxyAuth:
    def __call__(self) -> Dict[str, str]:
        return {"Proxy-Authorization": "<type> <credentials>"}
```

Authentication Flows

Sometimes we need to make additional calls to get credentials.

apiwrappers allows you to do just that:

```
from typing import Generator, Dict

from apiwrappers import Request, Response


class CustomAuthFlow:
    def __call__(self) -> Generator[Request, Response, Dict[str, str]]:
        # you can issue as many request as you needed
        # this is how you issue a request
        response = yield Request(...)

        # response is available immediately for processing
        return {"Authorization": response.json()["token"]}
```

Note, that a function now is generator function and you can yield as many request as you needed, but you should always return a dictionary with authentication headers.

3.1.4 Middleware

Middleware is a light “plugin” system for altering driver’s request/response/exception processing.

This page will walk you through the concept of middleware in the *apiwrappers* library.

Writing your own middleware

A middleware factory is a callable that takes a callable and returns a middleware. A middleware is a callable that takes same argument as `Driver.fetch()` and returns a response.

The most simple way is to write a middleware as function:

```
from apiwrappers.structures import NoValue


def simple_middleware(handler):

    def middleware(request, timeout=NoValue()):
        # Code to be executed before request is made
        response = handler(request, timeout)
        # Code to be executed after request is made
        return response

    return middleware
```

Since middleware is used by drivers, and the one we’ve written can be used only by regular driver, we also need to provide an async implementation:

```python
from apiwrappers.structures import NoValue


def simple_async_middleware(handler):

    async def middleware(request, timeout=NoValue()):
        # Code to be executed before request is made
        response = await handler(request, timeout)
        # Code to be executed after request is made
        return response

    return middleware
```

As you can see, the only difference is that in async middleware we have to await the handler call.

To help us reduce this code duplication *apiwrappers* provides a `BaseMiddleware` class. Subclassing one you can then override it hook methods like that:

```python
from typing import NoReturn

from apiwrappers import Request, Response
from apiwrappers.middleware import BaseMiddleware


class SimpleMiddleware(BaseMiddleware):
    def process_request(self, request: Request) -> Request:
        # Code to be executed before request is made
        return request

    def process_response(self, response: Response) -> Response:
        # Code to be executed after request is made
        return response

    def process_exception(
        self, request: Request, exception: Exception
    ) -> NoReturn:
        # Code to be executed when any exception is raised
        raise exception
```

Using middleware

Middleware are used by drivers and each driver accepts a list of middleware.

Although, middleware we defined earlier literally does nothing, it still can be used like that:

```python
>>> from apiwrappers import make_driver
>>> driver = make_driver("requests", SimpleMiddleware)
>>> driver
# RequestsDriver(Authorization, SimpleMiddleware, ...
```

Note, that even we provide only ``SimpleMiddleware`` the driver also has ``Authorization`` middleware. That's because some drivers have middleware that should always be present.

You can also change driver middleware after creation by simply reassigning *`Driver.middleware`* attribute:

```
>>> driver.middleware = []
>>> driver
# RequestsDriver(Authorization, ...
```

The order of the default middleware can be overridden by explicitly specifying it:

```
>>> driver.middleware = [SimpleMiddleware, Authorization]
>>> driver
# RequestsDriver(SimpleMiddleware, Authorization, ...
```

Middleware order

The order of middleware matters because a middleware can depend on other middleware.

Before making actual request, middleware are executed in the order they are defined. After getting the response middleware are executed in the reverse order.

3.1.5 Experimental Features

As experiment, there is also a `Fetch` descriptor, that helps reduce boilerplate and lets you write wrappers in almost declarative way:

```python
from __future__ import annotations

from dataclasses import dataclass
from typing import Any, Generic, List, Mapping, TypeVar

from apiwrappers import AsyncDriver, Driver, Request, Url
from apiwrappers.xfeatures import Fetch

T = TypeVar("T", Driver, AsyncDriver)


@dataclass
class Repo:
    id: int
    name: str

class Github(Generic[T]):
    get_repos = Fetch(List[Repo])

    def __init__(self, host: str, driver: T):
        self.url = Url(host)
        self.driver: T = driver

    @get_repos.request
    def get_repos_request(self, username: str) -> Request:
        url = self.url("/users/{username}/repos", username=username)
        return Request("GET", url)
```

Here we did the following:

1. First, we defined Repo dataclass that describes what we want to get from response

2. Next, we used `Fetch` descriptor to declare API method
3. Each `Fetch` object also needs a so-called request factory. We provide one by using `get_repos.request` decorator on the `get_repos_request` method
4. `get_repos_request` is a pure function and easy to test
5. No need to use overload - mypy will understand the return type of the `.get_repos` call

There are several trade-offs using this approach:

- no auto-completion when calling a method, which is a really huge flaw.
- mypy won't check the signature of the method due to limited support of the callable argument
- for end user it will be hard to understand what's going on and where to look in case of any problem

3.1.6 API Reference

This page is reference for the public API. Each class or function can be imported directly from **apiwrappers**.

`apiwrappers.`**`fetch`**(*driver:* apiwrappers.protocols.Driver, *request:* apiwrappers.entities.Request, *timeout: Union[int, float, None, datetime.timedelta, apiwrappers.structures.NoValue] = NoValue()*, *model:* None = None, *source: Optional[str] = None*) → *apiwrappers.entities.Response*

`apiwrappers.`**`fetch`**(*driver:* apiwrappers.protocols.AsyncDriver, *request:* apiwrappers.entities.Request, *timeout: Union[int, float, None, datetime.timedelta, apiwrappers.structures.NoValue] = NoValue()*, *model:* None = None, *source: Optional[str] = None*) → Awaitable[*apiwrappers.entities.Response*]

`apiwrappers.`**`fetch`**(*driver:* apiwrappers.protocols.Driver, *request:* apiwrappers.entities.Request, *timeout: Union[int, float, None, datetime.timedelta, apiwrappers.structures.NoValue] = NoValue()*, *model: Union[Callable[[...], apiwrappers.shortcuts.T], Type[apiwrappers.shortcuts.T]] = None*, *source: Optional[str] = None*) → apiwrappers.shortcuts.T

`apiwrappers.`**`fetch`**(*driver:* apiwrappers.protocols.AsyncDriver, *request:* apiwrappers.entities.Request, *timeout: Union[int, float, None, datetime.timedelta, apiwrappers.structures.NoValue] = NoValue()*, *model: Union[Callable[[...], apiwrappers.shortcuts.T], Type[apiwrappers.shortcuts.T]] = None*, *source: Optional[str] = None*) → Awaitable[apiwrappers.shortcuts.T]

Makes a request and returns response from server.

This is shortcut function for making requests. Prefer this over *`Driver.fetch()`* and *`AsyncDriver.fetch()`*. It also has extended behaviour and can parse JSON if `model` arg provided.

Parameters

- **`driver`** – driver that actually makes a request.
- **`request`** – request object.
- **`timeout`** – how many seconds to wait for the server to send data before giving up. If set to `None` waits infinitely. If provided, will take precedence over the `driver.timeout`.
- **`model`** – parser for a json response. This can be either type, e.g. List[int], or a callable that accepts json.
- **`source`** – name of the key in the json, which value will be passed to the model. You may use dotted notation to traverse keys, e.g. `key1.key2`.

Returns

- **Response** if regular driver is provided and `model` is not.
- **Awaitable[Response]** if asynchronous driver is provided, `model` is not.
- **T** if regular driver and model is provided. The `T` corresponds to `model` type.

- **Awaitable[T]** if asynchronous driver and model is provided. The `T` corresponds to `model` type.

Raises

- *`Timeout`* – the request timed out.
- `ssl.SSLError` – An SSL error occurred.
- *`ConnectionFailed`* – a connection error occurred.
- *`DriverError`* – in case of any other error in driver underlying library.

Simple Usage:

```
>>> from apiwrappers import Method, Request, fetch, make_driver
>>> driver = make_driver("requests")
>>> request = Request(Method.GET, "https://example.org")
>>> response = fetch(driver, request)
<Response [200]>
```

To use it in asynchronous code just use proper driver and don't forget to await:

```
>>> from apiwrappers import Method, Request, fetch, make_driver
>>> driver = make_driver("aiohttp")
>>> request = Request(Method.GET, "https://example.org")
>>> response = await fetch(driver, request)
<Response [200]>
```

If you provide `model` argument the JSON response will be parsed:

```
>>> from dataclasses import dataclass
>>> from typing import List
>>> from apiwrappers import Method, Request, fetch, make_driver
>>> @dataclass
... class Repo:
...     name: str
>>> driver = make_driver("requests")
>>> Request(
...     Method.GET,
...     "https://api.github.com/users/unmade/repos",
... )
>>> fetch(driver, request, model=List[Repo])
[Repo(name='am-date-picker'), ...]
```

Do note, it's highly discourage to use `Optional` if a `fetch` call, because mypy can't infer proper type for that case and the return type will be `object`

`apiwrappers.`**`make_driver`**(*driver_type: typing_extensions.Literal[requests], *middleware: Type[apiwrappers.protocols.Middleware], timeout: Union[int, float, None, datetime.timedelta] = 'DEFAULT_TIMEOUT', verify: Union[bool, str] = 'True', cert: Union[str, None, Tuple[str, str]] = 'None'*) → *apiwrappers.protocols.Driver*

`apiwrappers.`**`make_driver`**(*driver_type: typing_extensions.Literal[aiohttp], *middleware: Type[apiwrappers.protocols.AsyncMiddleware], timeout: Union[int, float, None, datetime.timedelta] = 'DEFAULT_TIMEOUT', verify: Union[bool, str] = 'True', cert: Union[str, None, Tuple[str, str]] = 'None'*) → *apiwrappers.protocols.AsyncDriver* `apiwrappers.`**`make_driver`**(*driver_type: str, *middleware: Union[Type[apiwrappers.protocols.Middleware], Type[apiwrappers.protocols.AsyncMiddleware]], timeout: Union[int, float, None, datetime.timedelta] = 'DEFAULT_TIMEOUT', verify: Union[bool, str] = 'True', cert: Union[str, None, Tuple[str, str]] = 'None'*) → Union[*apiwrappers.protocols.Driver*, *apiwrappers.protocols.AsyncDriver*]

Creates driver instance and returns it

This is a factory function to ease driver instantiation. That way you can abstract from specific driver class - no need to import it, no need to know how the class is called.

Parameters

- **`driver_type`** – specifies what kind of driver to create. Valid choices are `request` and `aiohttp`.
- **`*middleware`** – *middleware* to apply to driver. Dependant on `driver_type` it should be of one kind - either `Type[Middleware]` for regular drivers and `Type[AsyncMiddleware]` for asynchronous ones.
- **`timeout`** – how many seconds to wait for the server to send data before giving up. If set to `None` waits infinitely.
- **`verify`** – Either a boolean, in which case it controls whether to verify the server's TLS certificate, or a string, in which case it must be a path to a CA bundle to use.
- **`cert`** – Either a path to SSL client cert file (.pem) or a ('cert', 'key') tuple.

Returns

- **Driver** if `driver_type` is `requests`.
- **AsyncDriver** if `driver_type` is `aiohttp`.

Raises `ValueError` – if unknown driver type specified

Usage:

```
>>> from apiwrappers import make_driver
>>> make_driver("requests")
RequestsDriver(timeout=300, verify=True)
```

Driver Protocols

`class apiwrappers.`**`AsyncDriver`**(**args, **kwds*)

Protocol describing asynchronous driver.

`middleware`

list of *middleware* to be run on every request.

Type MiddlewareChain

`timeout`

how many seconds to wait for the server to send data before giving up. If set to `None` should wait infinitely.

Type *Timeout*

`verify`

Either a boolean, in which case it controls whether to verify the server's TLS certificate, or a string, in which case it must be a path to a CA bundle to use.

Type Verify

`cert`
Either a path to SSL client cert file (.pem) or a ('cert', 'key') tuple.

Type ClientCert

`async fetch`(*request*, *timeout=NoValue()*)
Makes actual request and returns response from the server.

Parameters

- **request** (`apiwrappers.entities.Request`) – a request object with data to send to server.
- **timeout** (*Union[int, float, None, datetime.timedelta, apiwrappers.structures.NoValue]*) – how many seconds to wait for the server to send data before giving up. If set to `None` waits infinitely. If provided, will take precedence over the *AsyncDriver.timeout*.

Return type *apiwrappers.entities.Response*

Returns: response from the server.

Raises

- ***Timeout*** – the request timed out.
- **ssl.SSLError** – An SSL error occurred.
- ***ConnectionFailed*** – a connection error occurred.
- ***DriverError*** – in case of any other error in driver underlying library.

Parameters

- **request** (`apiwrappers.entities.Request`) –
- **timeout** (*Union[int, float, None, datetime.timedelta, apiwrappers.structures.NoValue]*) –

Return type *apiwrappers.entities.Response*

`class apiwrappers.Driver`(**args*, ***kwds*)
Protocol describing regular synchronous driver.

`middleware`
list of *middleware* to be run on every request.

Type MiddlewareChain

`timeout`
how many seconds to wait for the server to send data before giving up. If set to `None` should wait infinitely.

Type *Timeout*

`verify`
Either a boolean, in which case it controls whether to verify the server's TLS certificate, or a string, in which case it must be a path to a CA bundle to use.

Type Verify

`cert`
Either a path to SSL client cert file (.pem) or a ('cert', 'key') tuple.

Type ClientCert

`fetch`(*request*, *timeout=NoValue()*)
Makes actual request and returns response from the server.

Parameters

- **`request`** (`apiwrappers.entities.Request`) – a request object with data to send to server.
- **`timeout`** (*`Union[int, float, None, datetime.timedelta, apiwrappers.structures.NoValue]`*) – how many seconds to wait for the server to send data before giving up. If set to `None` waits infinitely. If provided, will take precedence over the *`Driver.timeout`*.

Return type *apiwrappers.entities.Response*

Returns: response from the server.

Raises

- *`Timeout`* – The request timed out.
- `ssl.SSLError` – An SSL error occurred.
- *`ConnectionFailed`* – A Connection error occurred.
- *`DriverError`* – In case of any other error in driver underlying library.

Parameters

- **`request`** (`apiwrappers.entities.Request`) –
- **`timeout`** (*`Union[int, float, None, datetime.timedelta, apiwrappers.structures.NoValue]`*) –

Return type *apiwrappers.entities.Response*

Request and Response

`class apiwrappers.`**`Method`**(*value*)

A subclass of enum.Enum that defines a set of HTTP methods

The available methods are:

- DELETE
- HEAD
- GET
- POST
- PUT
- PATCH

Usage:

```
>>> from apiwrappers import Method
>>> Method.GET
<Method.GET: 'GET'>
>>> Method.POST == 'POST'
True
```

`class apiwrappers.`**`Request`**(*method, url, query_params=None, headers=None, cookies=None, auth=None, data=None, files=None, json=None*)

A container holding a request information

Parameters

- **method** (apiwrappers.entities.Method) – HTTP Method to use.
- **url** (apiwrappers.structures.Url) – URL to send request to.
- **query_params** (*Mapping[str, Optional[Iterable[str]]]*) – dictionary or list of tuples to send in the query string. Param with None values will not be added to the query string. Default value is empty dict.
- **headers** (*MutableMapping[str, str]*) – headers to send.
- **cookies** (*MutableMapping[str, str]*) – cookies to send.
- **auth** (*Optional[Union[Tuple[str, str], Callable[[], Dict[str, str]], Callable[[], Generator[*Request*, *Response*, Dict[str, str]]]]]*) – Auth tuple to enable Basic Auth or callable returning dict with authorization headers, e.g. '{"Authorization": "Bearer …"}'
- **data** (*Union[str, None, Mapping[str, Any], Iterable[Tuple[str, Any]]]*) – the body to attach to the request. If a dictionary or list of tuples `[(key, value)]` is provided, form-encoding will take place.
- **files** (*Optional[Dict[str, Union[BinaryIO, Tuple[str, BinaryIO], Tuple[str, BinaryIO, str]]]]*) – Dictionary of `'name': file-like-objects` (or `{'name': file-tuple}`) for multipart encoding upload. `file-tuple` can be a 2-tuple `('filename', fileobj)`, 3-tuple `('filename', fileobj, 'content_type')`, where `'content-type'` is a string defining the content type of the given file.
- **json** (*Union[str, int, float, bool, None, Mapping[str, Any], List[Any]]*) – json for the body to attach to the request (mutually exclusive with `data` arg).

Raises `ValueError` – If both `data` or `files` or `json` args provided.

Usage:

```
>>> from apiwrappers import Request
>>> Request(Method.GET, 'https://example.org')
Request(method=<Method.GET: 'GET'>, ...)
```

class apiwrappers.**Response**(*request*, *status_code*, *url*, *headers*, *cookies*, *content*, *encoding*)

A container holding a response from server.

Parameters

- **request** (apiwrappers.entities.Request) – request object to which this is a response.
- **status_code** (*int*) – integer Code of responded HTTP Status, e.g. 404 or 200.
- **url** (*str*) – final URL location of Response
- **headers** (*apiwrappers.structures.CaseInsensitiveDict[str]*) – case-insensitive dict of response headers. For example, `headers['content-encoding']` will return the value of a `'Content-Encoding'` response header.
- **cookies** (*http.cookies.SimpleCookie*) – cookies the server sent back.
- **content** (*bytes*) – content of the response, in bytes.
- **encoding** (*str*) – encoding or the response.

Return type None

json()

Returns the json-encoded content of the response.

Raises `ValueError` – if the response body does not contain valid json.

Return type Union[str, int, float, bool, None, Mapping[str, Any], List[Any]]

text()

Returns content of the response, in unicode.

If server response doesn't specified encoding, `utf-8` will be used instead.

Return type str

class apiwrappers.**Url**(*template*, ***replacements*)

Class to work with formatted string URLs and joining urls and path.

Sometimes it useful to keep original format string in place, for example, for logging or metrics. This class stores original format string and its replacements fields, substituting it when needed.

Parameters

- **template** (*str*) – a URL as format string, e.g. "https://example.org/users/{id}".
- **replacements** (*Any*) – values to format template with.

Usage:

```
>>> from apiwrappers import Url
>>> url = Url("https://example.org")
>>> url("/users/{id}", id=1)
Url('https://example.org/users/{id}', id=1)
>>> str(url("/users/{id}", id=1))
'https://example.org/users/1'
```

__call__(*path*, ***replacements*)

Joins path with current URL and return a new instance.

Parameters

- **path** (*str*) – a path as format string, e.g. "/users/{id}".
- **replacements** (*Any*) – values to path with.

Return type *apiwrappers.structures.Url*

Returns: New instance with a url joined with path.

Exceptions

exception apiwrappers.**DriverError**

Base class for driver-specific errors.

exception apiwrappers.**ConnectionFailed**

A Connection error occurred.

exception apiwrappers.**Timeout**

The request timed out.

PYTHON MODULE INDEX

INDEX