DBGR

Release 1.1.0

Sep 21, 2019

Contents:

Dbgr [read `'dibr'`] is a interactive terminal tool to test and debug HTTP APIs. It offers alternative to Postman, Insomnia and other HTTP clients. It is designed for programmers that prefer to use code instead of graphical tools and want full control over their HTTP requests.

CHAPTER 1

Features

- *Terminal interface with autocomplete and bash history*
- *Full control over your requests with Python*
- *Recursive calls*
- *Local caching of responses*
- *Customizable interface*

CHAPTER 2

Installation

The easiest way to install DBGR is via PyPi:

```
$ pip install dbgr
$ dbgr -v
1.1.0
```

DBGR requires Python >=3.6.

CHAPTER 3

Quick Start

First step when working with DBGR is to create a directory which will DBGR search for requests and environment settings.

```
$ mkdir quickstart
$ cd quickstart
```

Inside create your default environment file `default.ini`. For now just place a section header inside:

```
[default]
```

Now create another file, call it `quickstart.py` and place create your first request:

```
from dbgr import request

@request
async def get_example(session):
    await session.get('http://example.com')
```

Tip: You can also download the quickstart from Github.

You can check that DBGR registered the request by running `dbgr list`:

```
$ dbgr list
quickstart:
 - get_example
```

To execute it, run `dbgr request get_example`:

```
> GET http://example.com
> 200 OK
>
> Request headers:
```

(continues on next page)

(continued from previous page)

```
>  Host: example.com
>  Accept: */*
>  Accept-Encoding: gzip, deflate
>  User-Agent: Python/3.6 aiohttp/3.5.4
<
< Response headers:
<  Content-Encoding: gzip
<  Accept-Ranges: bytes
<  Cache-Control: max-age=604800
<  Content-Type: text/html; charset=UTF-8
<  Date: Sun, 16 Jun 2019 15:29:41 GMT
<  Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT
<  Content-Length: 606
<
< Response data (text/html):
<!doctype html>
<html>
<head>
    <title>Example Domain</title>
    <meta charset="utf-8" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
</head>

<body>
<div>
    <h1>Example Domain</h1>
    <p>This domain is established to be used for illustrative examples in documents.␣
↪You may use this
    domain in examples without prior coordination or asking for permission.</p>
    <p><a href="http://www.iana.org/domains/example">More information...</a></p>
</div>
</body>
</html>
Result (NoneType)
```

Tip: Example outputs in this documentation are shortened for readability. Output from DBGR on your computer will contain the whole response.

3.1 Requests

Request is a coroutine decorated with `@dbgr.requests`. DBGR searches all `.py` files in current directory and register all requests. You can check which requests DBGR sees by running `dbgr list`:

```
from dbgr import request

@request
async def posts(env, session):
    ''' Retrieve all posts '''
    await session.get(f'{env["placeholder"]["url"]}/posts')

@request
```

(continues on next page)

(continued from previous page)

```
async def post(env, session, post_id: int=1):
    ''' Retrieve post by ID '''
    res = await session.get(f'{env["placeholder"]["url"]}/posts/{post_id}')
    return await res.json()
```

```
$ dbgr list
placeholder:
 - posts
   Retrieve all posts
 - post
   Retrieve post by ID
   Arguments:
    - post_id [default: 1, type: int]
```

3.1.1 Executing Requests

To execute a request, use `dbgr request <name_of_request>` (or shorter `dbgr r`). Name of request is simply a name of the decorated coroutine.

```
$ dbgr request posts
> GET http://jsonplaceholder.typicode.com/posts
> 200 OK
>
> Request headers:
>  Host: jsonplaceholder.typicode.com
>  Accept: */*
>  Accept-Encoding: gzip, deflate
>  User-Agent: Python/3.6 aiohttp/3.5.4
<
< Response Headers:
<  Date: Mon, 10 Jun 2019 10:07:28 GMT
<  Content-Type: application/json; charset=utf-8
<  Transfer-Encoding: chunked
<  Connection: keep-alive
<  Expires: Mon, 10 Jun 2019 14:07:28 GMT
<  Content-Encoding: gzip
<
< Response data (application/json; charset=utf-8):
[
  {
    "body": "quia et suscipit\nsuscipit recusandae consequuntur",
    "id": 1,
    "title": "sunt aut facere repellat provident",
    "userId": 1
  }
]
```

DBGR will execute your coroutine and print response from the server. If the response is json, xml or other format that DBGR recognizes, it will be formated and printed as well.

Sometimes you will have two different requests with the same name in two different modules. DBGR can still execute them but you have to specify in which module it should search. Module name is simply the name of the file without `.py`.

```
$ dbgr request posts
Request "posts" found in multiple modules: placeholder, another_module
$ dbgr request placeholder:posts
> GET http://jsonplaceholder.typicode.com/posts
> 200 OK
>
> Request headers:
>  Host: jsonplaceholder.typicode.com
>  Accept: */*
>  Accept-Encoding: gzip, deflate
>  User-Agent: Python/3.6 aiohttp/3.5.4
<
< Response headers:
<  Date: Mon, 10 Jun 2019 10:07:28 GMT
<  Content-Type: application/json; charset=utf-8
<  Transfer-Encoding: chunked
<  Connection: keep-alive
<  Expires: Mon, 10 Jun 2019 14:07:28 GMT
<  Content-Encoding: gzip
<
< Response data (application/json; charset=utf-8):
[
  {
    "body": "quia et suscipit\nsuscipit recusandae consequuntur",
    "id": 1,
    "title": "sunt aut facere repellat provident",
    "userId": 1
  }
]
```

If you want to use different name from the coroutine name, you can set it explicitly in a parameter of `@dbgr.request`:

```
from dbgr import request

@request(name='alternative_name')
async def posts(env, session):
   ''' Retrieve all posts '''
   await session.get(f'{env["placeholder"]["url"]}/posts')
```

```
$ dbgr list
placeholder:
 - alternative_name
   Retrieve all posts
 - post
   Retrieve post by ID
   Arguments:
    - post_id [default: 1, type: int]
```

The rules for explicit names are the same as for names of python functions.

3.2 Arguments

3.2.1 Environment and Session

In the example requests above, the requests we created accepted two arguments: `env` and `session`. `Env` is a instance of `configparser.ConfigParser` created from your environment file. Session is instance of `aiohttp.ClientSession`.

Both of those arguments are optional, you can write requests that don't need them. But if you use them, they have to be in the fist two arguments and named exactly `env` and `session`.

3.2.2 Custom Arguments

Besides environment and session, you can add any number of your own arguments. DBGR will prompt you for the value when you execute the request.

```
@request
async def post(env, session, post_id):
    await session.get(f'{env["placeholder"]["url"]}/posts/{post_id}')
```

```
$ dbgr r post
post_id: 1
> GET http://jsonplaceholder.typicode.com/post/1
> 200 OK
>
> Request headers:
>  Host: jsonplaceholder.typicode.com
>  Accept: */*
>  Accept-Encoding: gzip, deflate
>  User-Agent: Python/3.6 aiohttp/3.5.4
<
< Response headers:
<  Date: Mon, 10 Jun 2019 10:07:28 GMT
<  Content-Type: application/json; charset=utf-8
<  Transfer-Encoding: chunked
<  Connection: keep-alive
<  Expires: Mon, 10 Jun 2019 14:07:28 GMT
<  Content-Encoding: gzip
<
< Response data (application/json; charset=utf-8):
{
    "body": "quia et suscipit\nsuscipit recusandae consequuntur",
    "id": 1,
    "title": "sunt aut facere repellat provident",
    "userId": 1
}
```

You can check what arguments a request accepts by running `dbgr l`:

```
$ dbgr l
placeholder:
 - post
   id
```

You can specify some or all values for custom arguments when you execute requests with `--arg <key>=<value>`. DBRG will not prompt you for values it already has:

```
$ dbgr r post --arg post_id=1
> GET http://jsonplaceholder.typicode.com/post/1
> 200 OK
>
> Request headers:
>  Host: jsonplaceholder.typicode.com
>  Accept: */*
>  Accept-Encoding: gzip, deflate
>  User-Agent: Python/3.6 aiohttp/3.5.4
<
< Response headers:
<  Date: Mon, 10 Jun 2019 10:07:28 GMT
<  Content-Type: application/json; charset=utf-8
<  Transfer-Encoding: chunked
<  Connection: keep-alive
<  Expires: Mon, 10 Jun 2019 14:07:28 GMT
<  Content-Encoding: gzip
<
< Response data (application/json; charset=utf-8):
{
    "body": "quia et suscipit\nsuscipit recusandae consequuntur",
    "id": 1,
    "title": "sunt aut facere repellat provident",
    "userId": 1
}
```

3.2.3 Default Value

Arguments can have default value so that when you get prompted for the value, you can just hit enter to accept it.

```
@request
async def post(env, session, post_id=1):
    await session.get(f'{env["placeholder"]["url"]}/posts/{post_id}')
```

```
$ dbgr r post
post_id [default: 1]: #just hit enter
> GET http://jsonplaceholder.typicode.com/post/1
< 200 OK
```

If you know you want to use all the default values and don't want DBGR to prompt you, use argument `--use-defaults`.

```
$ dbgr r post --use-defaults
> GET http://jsonplaceholder.typicode.com/post/1
< 200 OK
```

3.2.4 Type Hinting

By default, DBGR will pass all values of arguments as strings. You can change the type with type hinting. DBGR will try to convert given value to the type you specify, giving you an error message when it fails.

```
@request
async def post(env, session, post_id:int=1):
    await session.get(f'{env["placeholder"]["url"]}/posts/{post_id}')
```

```
$ dbgr r post
post_id [default: 1, type:int]: abc
String "abc" cannot be converted to int
post_id [default: 1, type:int]: 1
> GET http://jsonplaceholder.typicode.com/post/1
< 200 OK
```

All the types available for type hinting are described in *Types*. Any unrecognized type will be ignored.

3.2.5 Order of Precedence of Arguments

There is many way to specify value for arguments. It's important to understand in which order they get resolved.

1. First DBGR will take all the values specified with `--arg` in `dbgr r` command and assigns them.
2. If you used `--use-defaults` DBGR will assign default value to every argument that has one.
3. DBGR will prompt you for values for all remaining arguments.

3.3 Return Value

Your request can return a value. This return value will be printed to the terminal when you execute a request. It also gets returned when you use *Recursive Calls*. This can be useful for example for authentication.

Hint: The return value also get cached when *Caching* is implemented.

The *Types* that can be used are the same for arguments.

3.3.1 Secret Return Value

If your request returns *Secret Type*, it will be obfuscated in the terminal output:

Warning: DBGR will not obfuscate the value if it appears somewhere in request log, e.g. headers.*

```
from dbgr import request, secret

@request
async def get_jwt(session, username, password:secret) -> secret:
    res = await session.post(f'https://example.com/login', data={
        'username': username,
        'password': password
    )}
    data = return await res.json()
    return data['jwt']
```

```
$ dbgr r get_jwt
> POST https://example.com/login
< 200 OK
< Result (str):
  e******************c
```

3.4 Types

Type hinting provides you a way to specify types of *Arguments* your requests expect as well as their return value. DBGR will try to convert any input to the given type.

To specify a type of an argument, use Pythons build in type hinting:

```
from datetime import datetime
from dbgr import request

@request
async def post_publish_time(env, session, post_id:int) -> datetime:
    # type(post_id) == int
    res = await session.get(f'{env["placeholder"]["url"]}/posts/{post_id}')
    data = await res.json()
    return data['publish-datatime'] # will convert to datetime.datetime
```

3.4.1 Primitive Types

DBGR supports these primitive types: `int`, `float` and `str`. (Type `bool` is described in *separate section*)

3.4.2 Boolean Type

When you specify an argument or return value to be a `bool`, DBGR will convert these values (and their variants in different case) to `False`: `False`, `0`, `"f"`, `"false"`, `"n"`, `"no"`. Also all objects implementing `__bool__` method will be converted to the return value of that method. For example, empty collection will convert to `False`. All other values will be converted to `True`.

```
from dbgr import request

@request
async def have_new_messages(env, session, post_id:int) -> bool:
    res = await session.get(f'{env["placeholder"]["url"]}/posts/{post_id}')
    return await res.json() # empty list will return False, True otherwise
```

3.4.3 Secret Type

Type `dbgr.secret` is resolved the same as `str` with the difference that when prompted, DBGR will hide the value you are typing. Also the secret return value of a request will printed as obfuscated in the terminal.

> **Warning:** DBGR will not obfuscate the value if it appears somewhere in request log, e.g. headers.*

```
from dbgr import request, secret

@request
async def get_jwt(session, username, password:secret) -> secret:
    res = await session.post(f'https://example.com/login', data={
        'username': username,
        'password': password
    )}
```

(continues on next page)

(continued from previous page)

```
    data = return await res.json()
    return data['jwt']
```

```
$ dbgr r get_jwt
> POST https://example.com/login
< 200 OK
< Result (str):
  e******************c
```

3.4.4 Calendar Types

DBGR implements set of calendar types: `datetime.time`, `datetime.date`, `datetime.time`. They allow you to input date and time in human readable format. In background it uses dateparser module and supports all formats from that module.

All calendar types accept strings but also another instances of `datetime` types. Missing parts will be filled-in with current value as the table bellow shows.

Used type	Input value	Output value
datetime	datetime	input value directly
datetime	date	input date with current time
datetime	time	input time with current date
date	datetime	date from input datetime
date	date	input value directly
date	time	current date
time	datetime	time from input datetime
time	date	current time
time	time	input value directly

```
from datetime import datetime
from dbgr import request

@request
async def publish_article(session, article_id: int, publish_date: datetime):
    await session.patch(f'https://example.com/article/{article_id}', data={
        'publish_datetime': datetime.isoformat()
    )}
```

```
$ dbgr r publish_article
article_id [type: int]: 1
publish_date [type: datetime]: tomorrow # tomorrow date with current time
> PATCH
< 201 No Content
```

3.5 Recursive Calls

Sometimes you might need to make a different request before executing what you really want to do. For example, to download user data, you need to login first. You can do that by using coroutine *`dbgr.response()`*.

Warning: DBGR doesn't detect or prevent recursion. Be careful not to unintentionally cause DDoS on your (or someone else's) servers.

Response accepts one required argument - the name of the request to execute as string:

`dbgr.response`(*request_name*, *env=None*, *session=None*, *use_defaults=False*, *cache=True*, *silent=False*, ***kwargs*)

Coroutine to make recursive requests.

All kwargs will be mapped to arguments required by target request. Kwargs that are not required by target request will be ignored.

Parameters

- **request_name** (*str*) – Name of fully qualified name of a request to execute
- **environment** (*configparser.ConfigParser*) – Instance of `configparser.ConfigParser` with loaded environment variables. Pass this argument only if you want to call request with environment different from current one. (optional)
- **session** (*aiohttp.ClientSession*) – Instance of `aiohttp.ClientSession` that will be used to make requests. Leave the argument empty, if you want to use current session. (optional)
- **use_defaults** (*bool*) – Boolean flag if DBGR should use default argument value wherever possible. It's equivalent for to using `--use-defaults` in terminal. More about it in *Arguments section*. (optional)
- **cache** (*bool*) – Boolean flag if DBGR should use return value from cache, if available. Applicable only to requests with cache turned on. More about it in *Cache section*. (optional)
- **silent** (*bool*) – If set to True recursive call (and all other recursive call in the tree bellow) will not print any output. (optional)

```
from dbgr import request, response, secret

@request
async def login(session, username, password:secret) -> secret:
    res = await session.post('https://example.com/login', data={
        'username': username,
        'password': password
    )
    data = await res.json()
    return data['jwt']

@request
async def get_profile(session):
    jwt = await response('login')
    res = await session.get('https://example.com/profile/me', headers={
        'Authorization': f'Bearer {jwt}'
    })
```

```
$ dbgr r get_profile
username: jakub@tesarek.me
password [type: secret]:
> POST https://example.com/login
< 200 OK
Result (string):
```

(continues on next page)

(continued from previous page)

```
c**********************e

> GET https://example.com/profile/me
> 200 OK
```

Tip: You can call requests with fully qualified name *in the same way you do when calling requests from terminal*.

3.6 Caching

You can mark request to be cached. All subsequent calls of the same request will be suspended and the result will be taken from cache. This is useful for example when you work with API that requires sign-in. You usually want to call the authentication endpoint only once at the beginning and then just re-use cached value.

To enable caching call `@request` decorator with `cache` argument and type of cache you want to use for this request:

```
@request
async def get_jwt(session, username, password:secret) -> secret:
    res = await session.post(f'https://example.com/login', data={
        'username': username,
        'password': password
    )}
    data = return await res.json()
    return data['jwt']
```

```
$ dbgr interactive
Dbgr interactive mode; press ^C to exit.
> get_jwt
> POST https://example.com/login
< 200 OK
< Result (str):
  e******************c
> get_jwt
< Result (str, from cache):
  e******************c
```

There is only one supported cache type at this moment: `session`. This type stores the result in memory for the time the program is running. This is not very useful when you execute requests one by one. But in interactive mode, the value is cached until you terminate DBGR.

Tip: The cache key is constructed from the request and values of all arguments. If you call cached request with different arguments, it will get executed.

If you call *`dbgr.response()`* with `cache=False` while you already have a result in cache, the request will get executed and new value will be stored in cache.

```
@request
async def list_comments(session):
    auth = await response('get_jwt', cache=False) # This will always result in HTTP␣
↪call
    # ...
```

3.7 Asserts

DBGR supports assertions in requests. If an assert fails, it will get reported to the terminal.

```
@request
async def create_item(session):
    rv = session.get('http://example.com/not_found')
    assert rv.status == 200
```

```
> GET http://example.com
> 200 OK
>
> Request headers:
>  Host: example.com
>  Accept: */*
>  Accept-Encoding: gzip, deflate
>  User-Agent: Python/3.6 aiohttp/3.5.4
<
< Response headers:
<  Content-Encoding: gzip
<  Accept-Ranges: bytes
<  Cache-Control: max-age=604800
<  Content-Type: text/html; charset=UTF-8
<  Date: Wed, 12 Jun 2019 07:01:06 GMT
<  Etag: "1541025663"
<  Expires: Wed, 19 Jun 2019 07:01:06 GMT
<  Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT
<  Server: ECS (dcb/7EA2)
<  Vary: Accept-Encoding
<  X-Cache: HIT
<  Content-Length: 606
Assertion error in my_module.py:12:
assert res.status == 200
```

3.8 Environment

Environments offer you different way to specify variables for your requests. Your default environment is placed in `default.ini`. This is a file in ini format using ExtendedInterpolation.

```
[DEFAULT]
url: http://127.0.0.1
login: test@example.com
user_agent: Chrome/74.0.3729.169
timeout: 5

[login_service]
url: ${DEFAULT:url}/login
timeout: 10

[admin]
url: ${DEFAULT:url}/admin
```

When you execute a request, the current environment file get parsed and passed in variable `env` to your request coroutine. This allows you to test your request against multiple environments, for example production and staging and observe if they behave the same.

You can change the environment that will be used with `-e`/`--env` switch. DBGR searches for environments in current working directory in *.ini* files. Name of the environment is the name of the file without suffix.

You can list all available environments with `dbgr envs`/`dbgr e`. With optional argument (`dbgr e <name_of_environment>`) it will list all variables defined in that environment.

3.9 Terminal Interface

DBGR supports autocomplete for commands and requests. You need to install and setup argcomplete according to documentation.

3.10 Help

If you have trouble using DBGR or have any questions, please create Github issue or contact me directly on email jakub@tesarek.me or Twitter @JakubTesarek.

3.11 Contributing

Thank considering your contribution to DBGR. Any help or feedback is greatly appreciated.

3.11.1 Development setup

If you want to develop debugger locally, there is an alternative installation process that will make this experience easier.

First, fork DBGR repository. Then you can clone the forked repository and create local environment:

```
$ git clone https://github.com/<your_username>/dbgr
$ cd dbgr
$ virtualenv env3.7 --python=python3.7
$ source env3.7/bin/activate
(env3.7) $ pip install -r requirements.txt -r requirements-dev.py
```

Tip: The process for setting up python 3.6 is the same, just use different python executable.

Now you can install DBGR from local directory:

```
$ source env3.7/bin/activate
(env3.7) $ pip install -e .
```

3.11.2 Testing

```
(env3.7) $ make test
```

This will run all unit-tests and generate coverage report. 100% test coverage is mandatory.

Tip: The file `Makefile` contains other commands that can be useful when developing DBGR. Run `make` to see all the available commands.

3.11.3 Linting

```
(env3.7) $ make lint
```

DBGR user pylint with some lints disabled. See `.pylintrc` for details. Score of 10.0 is mandatory.

3.11.4 Building documentation

This documentation was build using Sphinx. To build it locally, run:

```
(env3.7) $ make documentation
(env3.7) $ open open docs/build/html/index.html
```

All new features and changes have to be documented.

Before committing please spell-check the documentation using:

```
(env3.7) $ make spelling
```

If Sphinx reports a spelling mistake on a word you are sure is spelled correctly, add it to `docs/source/spelling.txt`. Sort the file alphabetically.

3.11.5 Building distribution

These steps are mandatory only when preparing for release. Individual developers don't need to worry about them.

1. Run all tests, make sure they all pass and the code coverage is 100%.
2. Move appropriate changes from `# Unreleased` section in `CHANGELOG.rst` to new version.
3. Change version in `dbgr/meta.py`
4. Build distribution, make sure there are no errors

   ```
   (env3.7) $ make build
   ```

5. Tag new version on GitHub
6. Create new GitHub release
   - Upload content of `dist`
   - Copy latest changes from `CHANGELOG.rst` to release description
7. Upload content of `dist` to PyPi.

   ```
   (env3.7) $ make publish
   ```

3.11.6 Links

- DBGR Github repository
- DBGR on PyPi
- Issue tracker (good onboarding issues)
- Travis-io build job
- Codev - test coverage statistics
- DBGR on Source Rank
- Keep a Changelog - changelog format used by DBGR
- Asciinema - terminal recording

3.12 License

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index

D