BottleShip Documentation

Release 0.2.4

Oscar Martinez

February 19, 2016

Contents

1 BottleShip **3**
- 1.1 Introduction . . . 3
- 1.2 Features . . . 3
- 1.3 Getting Started . . . 3
- 1.4 Routes . . . 4
- 1.5 Security . . . 5
- 1.6 License . . . 7

2 Installation **9**

3 Usage **11**
- 3.1 Getting Started . . . 11
- 3.2 Routes . . . 12
- 3.3 Security . . . 13

4 Contributing **15**
- 4.1 Types of Contributions . . . 15
- 4.2 Get Started! . . . 16
- 4.3 Pull Request Guidelines . . . 16
- 4.4 Tips . . . 17

5 Credits **19**
- 5.1 Project Lead . . . 19
- 5.2 Contributors . . . 19

6 History **21**
- 6.1 0.1.0 (2016-01-17) . . . 21
- 6.2 0.2.0 (2016-01-17) . . . 21
- 6.3 0.2.1 (2016-01-19) . . . 21
- 6.4 0.2.2 (2016-02-14) . . . 21
- 6.5 0.2.3 (2016-02-15) . . . 21
- 6.6 0.2.4 (2016-02-17) . . . 21

7 Indices and tables **23**

Contents:

CHAPTER 1

BottleShip

Authentication for the Bottle web framework made simple.

- Free software: MIT license
- Documentation: https://bottleship.readthedocs.org.

1.1 Introduction

BottleShip is a very simple library for authentication using the Bottle web framework. It supports the standard workflow of registration, login, and authentication required by simple applications that need to maintain a state for individual users.

1.2 Features

- Very simple and easy to use
- Works both on Python 2.x and 3.x
- Very few dependencies

1.3 Getting Started

This documentation assumes that you already have a working Bottle application or that you are somewhat familiar with the Bottle web framework. If you need to reference documentation for Bottle, here is the link.

The easiest way to install BottleShip is using pip:

```
$ pip install bottleship
```

With BottleShip installed, this is what it takes to use authentication to lock certain routes so they can only be used by users who are logged in:

```
# Instantiate class and register "register" and "login" routes
bs = BottleShip()
bs.route('/register', method=('GET', 'POST'), callback=bs.register)
bs.route('/login', method=('GET', 'POST'), callback=bs.login)

# This API endpoint can only be reached by users who have logged in
```

```
@bs.require_auth('/testapi', method=('GET', 'POST'))
def testapi(bottleship_user_record):
    return "Hello, %s!" % bottleship_user_record.get('Username')
```

New users can register by visiting the `/register` endpoint and sending their `username` and `password` as part of their request. For example, a new user can be registered with the following request:

```
>>> curl http://127.0.0.1:8080/register?Username=john&Password=1234
... HTTP/1.0 200 OK
... Content-Length: 155
... Content-Type: text/html; charset=UTF-8
... Date: Sun, 17 Jan 2016 23:36:02 GMT
... Server: WSGIServer/0.1 Python/2.7.10
...
... {"Username": "john", "SecurityLevel": "plaintext", "Password": "1723328
... 704", "RemoteIpAddr": "127.0.0.1", "__id__": "040220e5-1cce-4cdd-af9d-2
... ad2885263aa"}
```

Similarly, to log in, a user can make the following request:

```
>>> curl http://127.0.0.1:8080/login?Username=john&Password=1234
... HTTP/1.0 200 OK
... Content-Length: 247
... Content-Type: text/html; charset=UTF-8
... Date: Sun, 17 Jan 2016 23:36:03 GMT
... Set-Cookie: Token=5f04ee43-83bb-46c0-96aa-65a2c585a796; Path=/
... Server: WSGIServer/0.1 Python/2.7.10
...
... {"Username": "john", "SecurityLevel": "plaintext", "LastLogin": "145307
... 3842.72", "Token": "5f04ee43-83bb-46c0-96aa-65a2c585a796", "__id__": "0
... 40220e5-1cce-4cdd-af9d-2ad2885263aa", "Key": null, "Password": "1723328
... 704", "RemoteIpAddr": "127.0.0.1"}
```

Both requests will return a JSON object that represents a record with all the information that the BottleShip server has about the user. A login request's returned JSON also has a field named `Token` that contains the user session token. In addition to that, the returned request will also store the session token as part of the cookies in the request headers.

If the login was successful, the user can now make the following request:

```
>>> curl http://127.0.0.1:8080/testapi3?Token=5f04ee43-83bb-46c0-96aa-65a2c
    585a796
... HTTP/1.0 200 OK
... Content-Length: 12
... Content-Type: text/html; charset=UTF-8
... Date: Sun, 17 Jan 2016 23:36:04 GMT
... Server: WSGIServer/0.1 Python/2.7.10
...
... Hello, john!
```

If everything worked, the user will receive `Hello, john!`.

1.4 Routes

BottleShip's `require_auth` method has nearly identical signature compared to Bottle's `route`. The main difference is that, instead of a single `callback` parameter, it has two:

- `callback_success`. Optional; roughly equivalent to Bottle's `callback`. When not set, defaults to sending a request back to the user with the status code of `200` and body `OK`.

- `callback_failure`. Optional; when not set, defaults to sending a request back to the user with status code of `403` and body containing more details about the failure (but no stack trace).

Like Bottle's `route` method, `require_auth` can be used both as a regular function that takes callable objects parameters for `callback_success` and `callback_failure`, or as a decorator to wrap the function `callback_success`.

For applications intended for web browsers that can rely on cookies for session tokens, this function is essentially a drop-in replacement for Bottle's `route`. For example, the following snippet:

```
app = Bottle()
@app.route('/hello/<name>')
def hello(name):
    return 'Hello, %s!' % name
```

Becomes this:

```
app = BottleShip()
@app.require_auth('/hello/<name>')
def hello(name):
    return 'Hello, %s!' % name
```

For convenience, and to avoid interfacing with the underlying data about the users at more than one layer in the application, routes can receive a copy of the record representing a user by adding a parameter named `bottleship_user_record` to the function's signature. The information will be represented as a `dict` and contains:

- `Username`
- `Password`, if any (hashed)
- `__id__`, used internally by the database engine
- `RemoteIpAddr`
- Any other information added by the client during registration or login as part of the request

Then, the previous example can be simplified further and changed to:

```
app = BottleShip()
@app.require_auth('/hello')
def hello(bottleship_user_record):
    return 'Hello, %s!' % bottleship_user_record.get('Username')
```

1.5 Security

Needless to say, you should not be transmitting passwords over a plain connection like it is done in the example above. If you cannot achieve a cryptographically secure connection between user and server, your only hope is to implement a public key scheme to allow for secure transmission of user password and token. Such scheme is not implemented in BottleShip, but it has a few mitigations in place that yield a marginal increase in security.

When registration takes place, all information provided by the user is recorded. Most of it is provided by the user himself so it could be easily forged, but the IP address is slightly more difficult to fake. Using the user IP address, along with some form of whitelisting (or blacklisting), allows for a relative improvement in the application security. To achieve this, one must provide the whitelist upon instantiation like:

```
valid_users = {"RemoteIpAddr": "127.0.0.1"}
bs = BottleShip(whitelist_cond=valid_users)
```

Then, when the user registers, BottleShip will make sure that only requests from the provided IP addresses have permission to reach the endpoint.

Another mitigation regarding the user IP address is the verification of addresses not changing between registration and login. This is achieved by appending `+ipaddr` to the desired security level upon registration. For example, a new user can be registered with the following request:

```
>>> curl http://127.0.0.1:8080/register?Username=john&Password=1234&Securit
    yLevel=plaintext%2Bipaddr
... HTTP/1.0 200 OK
... Content-Length: 162
... Content-Type: text/html; charset=UTF-8
... Date: Sun, 17 Jan 2016 23:36:05 GMT
... Server: WSGIServer/0.1 Python/2.7.10
...
... {"Username": "john", "SecurityLevel": "plaintext+ipaddr", "Password": "
... 1723328704", "RemoteIpAddr": "127.0.0.1", "__id__": "1b5ca834-f4fb-4f6a
... -96f3-5a427ca43270"}
```

Note that the + sign is URL encoded so `plaintext` becomes `plaintext+ipaddr`, which is encoded into `plaintext%2Bipaddr`. IP address verification is the only security feature that will persist between registration and login. Other than that, the security level during login can be whatever the client chooses regardless of the security level during registration.

A more sophisticated security mitigation is implementing HMAC signing for the information exchanged between client and server during registration and login. This requires an additional step to perform the key exchange prior to registration and/or login. The key exchange will provide the user with a single-use token that can be utilized by the client to send the server information signed with the secret key provided during the exchange.

```
>>> curl http://127.0.0.1:8080/swapkeys/hmac/5f04ee43-83bb-46c0-96aa-65a2c5
    85a796
... HTTP/1.0 200 OK
... Content-Length: 114
... Content-Type: text/html; charset=UTF-8
... Date: Sun, 17 Jan 2016 23:36:06 GMT
... Server: WSGIServer/0.1 Python/2.7.10
...
... !1ICg4mv4H8NGUyV5aveJU1fJ/wnFr0cOks+KMIvZuIo=?eyJUb2tlbiI6ICI0OGYyNWM4O
... S1mZDg2LTRhMzctOGYyNi00NmYxNmE0YzVlYWIifQ==
```

Note that the token is encoded in base64 and later signed with the user-provided key. Decoding the above string produces `{"Token": "48f25c89-fd86-4a37-8f26-46f16a4c5eab"}`.

Which can then be hashed and the signature verified using the user-provided secret key. In the next step, the client can send all the user information encoded and signed along with the single-use token so the server knows which key to verify the data with:

```
>>> curl http://127.0.0.1:8080/register?Token=48f25c89-fd86-4a37-8f26-46f16
    a4c5eab&Data=!6uz1tJzSZX%2F0EhVqj4ZpTMiiNmONVPY601ZHCHLXu9M%3D%3FeyJVc2
    VybmFtZSI6ImpvaG4iLCJQYXNzd29yZCI6IjEyMzQifQ%3D%3D
... HTTP/1.0 200 OK
... Content-Length: 202
... Content-Type: text/html; charset=UTF-8
... Date: Sun, 17 Jan 2016 23:36:07 GMT
... Server: WSGIServer/0.1 Python/2.7.10
...
... {"Username": "john", "SecurityLevel": "plaintext", "__id__": "3be4ed1c-
... d30d-4786-bfc7-97728120e7b2", "Key": "5f04ee43-83bb-46c0-96aa-65a2c585a
... 796", "Password": "1723328704", "RemoteIpAddr": "127.0.0.1"}
```

The data returned by the server is in plaintext because a security level was not specified in the request. If the client wants the user information encoded, he must explicitly specify a security level that enforces signature verification.

The only other method in the authentication workflow other than registration that supports encoding is login. The function signature is identical and the token is also of single-use. After login, any further references of `token` in the APIs assume that it is the session token. It is worth noting that, because the token and user key are expected to last as long as the session does, it is pointless to encode, hash, or otherwise obscure the token or user key. Since the same string, encrypted or otherwise, will be sent in each request by the client, it makes no difference to an attacker to sniff the plaintext version or the encrypted version of the token; he can just present the server with the same string and it will be accepted as valid. For similar reasons, the password is being sent in plaintext form to the server and it is only hashed internally.

1.6 License

CHAPTER 2

Installation

At the command line:

```
$ pip install bottleship
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv bottleship
$ pip install bottleship
```

CHAPTER 3

Usage

3.1 Getting Started

This documentation assumes that you already have a working Bottle application or that you are somewhat familiar with the Bottle web framework. If you need to reference documentation for Bottle, **'here is the link'_**.

The easiest way to install BottleShip is using pip:

```
$ pip install bottleship
```

With BottleShip installed, this is what it takes to use authentication to lock certain routes so they can only be used by users who are logged in:

```
# Instantiate class and register "register" and "login" routes
bs = BottleShip()
bs.route('/register', method=('GET', 'POST'), callback=bs.register)
bs.route('/login', method=('GET', 'POST'), callback=bs.login)

# This API endpoint can only be reached by users who have logged in
@bs.require_auth('/testapi', method=('GET', 'POST'))
def testapi(bottleship_user_record):
    return "Hello, %s!" % bottleship_user_record.get('Username')
```

New users can register by visiting the `/register` endpoint and sending their `username` and `password` as part of their request. For example, a new user can be registered with the following request:

```
>>> curl http://127.0.0.1:8080/register?Username=john&Password=1234
... HTTP/1.0 200 OK
... Content-Length: 155
... Content-Type: text/html; charset=UTF-8
... Date: Sun, 17 Jan 2016 23:36:02 GMT
... Server: WSGIServer/0.1 Python/2.7.10
...
... {"Username": "john", "SecurityLevel": "plaintext", "Password": "1723328
... 704", "RemoteIpAddr": "127.0.0.1", "__id__": "040220e5-1cce-4cdd-af9d-2
... ad2885263aa"}
```

Similarly, to log in, a user can make the following request:

```
>>> curl http://127.0.0.1:8080/login?Username=john&Password=1234
... HTTP/1.0 200 OK
... Content-Length: 247
... Content-Type: text/html; charset=UTF-8
... Date: Sun, 17 Jan 2016 23:36:03 GMT
```

```
... Set-Cookie: Token=5f04ee43-83bb-46c0-96aa-65a2c585a796; Path=/
... Server: WSGIServer/0.1 Python/2.7.10
...
... {"Username": "john", "SecurityLevel": "plaintext", "LastLogin": "145307
... 3842.72", "Token": "5f04ee43-83bb-46c0-96aa-65a2c585a796", "__id__": "0
... 40220e5-1cce-4cdd-af9d-2ad2885263aa", "Key": null, "Password": "1723328
... 704", "RemoteIpAddr": "127.0.0.1"}
```

Both requests will return a JSON object that represents a record with all the information that the BottleShip server has about the user. A login request's returned JSON also has a field named `Token` that contains the user session token. In addition to that, the returned request will also store the session token as part of the cookies in the request headers.

If the login was successful, the user can now make the following request:

```
>>> curl http://127.0.0.1:8080/testapi3?Token=5f04ee43-83bb-46c0-96aa-65a2c
    585a796
... HTTP/1.0 200 OK
... Content-Length: 12
... Content-Type: text/html; charset=UTF-8
... Date: Sun, 17 Jan 2016 23:36:04 GMT
... Server: WSGIServer/0.1 Python/2.7.10
...
... Hello, john!
```

If everything worked, the user will receive `Hello, john!`.

3.2 Routes

BottleShip's `require_auth` method has nearly identical signature compared to Bottle's `route`. The main difference is that, instead of a single `callback` parameter, it has two:

- `callback_success`. Optional; roughly equivalent to Bottle's `callback`. When not set, defaults to sending a request back to the user with the status code of `200` and body `OK`.
- `callback_failure`. Optional; when not set, defaults to sending a request back to the user with status code of `403` and body containing more details about the failure (but no stack trace).

Like Bottle's `route` method, `require_auth` can be used both as a regular function that takes callable objects parameters for `callback_success` and `callback_failure`, or as a decorator to wrap the function `callback_success`.

For applications intended for web browsers that can rely on cookies for session tokens, this function is essentially a drop-in replacement for Bottle's `route`. For example, the following snippet:

```
app = Bottle()
@app.route('/hello/<name>')
def hello(name):
    return 'Hello, %s!' % name
```

Becomes this:

```
app = BottleShip()
@app.require_auth('/hello/<name>')
def hello(name):
    return 'Hello, %s!' % name
```

For convenience, and to avoid interfacing with the underlying data about the users at more than one layer in the application, routes can receive a copy of the record representing a user by adding a parameter named

`bottleship_user_record` to the function's signature. The information will be represented as a `dict` and contains:

- `Username`
- `Password`, if any (hashed)
- `__id__`, used internally by the database engine
- `RemoteIpAddr`
- Any other information added by the client during registration or login as part of the request

Then, the previous example can be simplified further and changed to:

```
app = BottleShip()
@app.require_auth('/hello')
def hello(bottleship_user_record):
    return 'Hello, %s!' % bottleship_user_record.get('Username')
```

3.3 Security

Needless to say, you should not be transmitting passwords over a plain connection like it is done in the example above. If you cannot achieve a cryptographically secure connection between user and server, your only hope is to implement a public key scheme to allow for secure transmission of user password and token. Such scheme is not implemented in BottleShip, but it has a few mitigations in place that yield a marginal increase in security.

When registration takes place, all information provided by the user is recorded. Most of it is provided by the user himself so it could be easily forged, but the IP address is slightly more difficult to fake. Using the user IP address, along with some form of whitelisting (or blacklisting), allows for a relative improvement in the application security. To achieve this, one must provide the whitelist upon instantiation like:

```
valid_users = {"RemoteIpAddr": "127.0.0.1"}
bs = BottleShip(whitelist_cond=valid_users)
```

Then, when the user registers, BottleShip will make sure that only requests from the provided IP addresses have permission to reach the endpoint.

Another mitigation regarding the user IP address is the verification of addresses not changing between registration and login. This is achieved by appending `+ipaddr` to the desired security level upon registration. For example, a new user can be registered with the following request:

```
>>> curl http://127.0.0.1:8080/register?Username=john&Password=1234&Securit
    yLevel=plaintext%2Bipaddr
... HTTP/1.0 200 OK
... Content-Length: 162
... Content-Type: text/html; charset=UTF-8
... Date: Sun, 17 Jan 2016 23:36:05 GMT
... Server: WSGIServer/0.1 Python/2.7.10
...
... {"Username": "john", "SecurityLevel": "plaintext+ipaddr", "Password": "
... 1723328704", "RemoteIpAddr": "127.0.0.1", "__id__": "1b5ca834-f4fb-4f6a
... -96f3-5a427ca43270"}
```

Note that the + sign is URL encoded so `plaintext` becomes `plaintext+ipaddr`, which is encoded into `plaintext%2Bipaddr`. IP address verification is the only security feature that will persist between registration and login. Other than that, the security level during login can be whatever the client chooses regardless of the security level during registration.

A more sophisticated security mitigation is implementing HMAC signing for the information exchanged between client and server during registration and login. This requires an additional step to perform the key exchange prior to registration and/or login. The key exchange will provide the user with a single-use token that can be utilized by the client to send the server information signed with the secret key provided during the exchange.

```
>>> curl http://127.0.0.1:8080/swapkeys/hmac/5f04ee43-83bb-46c0-96aa-65a2c5
    85a796
... HTTP/1.0 200 OK
... Content-Length: 114
... Content-Type: text/html; charset=UTF-8
... Date: Sun, 17 Jan 2016 23:36:06 GMT
... Server: WSGIServer/0.1 Python/2.7.10
...
... !1ICg4mv4H8NGUyV5aveJU1fJ/wnFr0cOks+KMIvZuIo=?eyJUb2tlbiI6ICI0OGYyNWM4O
... S1mZDg2LTRhMzctOGYyNi00NmYxNmE0YzVlYWIifQ==
```

Note that the token is encoded in base64 and later signed with the user-provided key. Decoding the above string produces `{"Token": "48f25c89-fd86-4a37-8f26-46f16a4c5eab"}`.

Which can then be hashed and the signature verified using the user-provided secret key. In the next step, the client can send all the user information encoded and signed along with the single-use token so the server knows which key to verify the data with:

```
>>> curl http://127.0.0.1:8080/register?Token=48f25c89-fd86-4a37-8f26-46f16
    a4c5eab&Data=!6uz1tJzSZX%2F0EhVqj4ZpTMiiNmONVPY601ZHCHLXu9M%3D%3FeyJVc2
    VybmFtZSI6ImpvaG4iLCJQYXNzd29yZCI6IjEyMzQifQ%3D%3D
... HTTP/1.0 200 OK
... Content-Length: 202
... Content-Type: text/html; charset=UTF-8
... Date: Sun, 17 Jan 2016 23:36:07 GMT
... Server: WSGIServer/0.1 Python/2.7.10
...
... {"Username": "john", "SecurityLevel": "plaintext", "__id__": "3be4ed1c-
... d30d-4786-bfc7-97728120e7b2", "Key": "5f04ee43-83bb-46c0-96aa-65a2c585a
... 796", "Password": "1723328704", "RemoteIpAddr": "127.0.0.1"}
```

The data returned by the server is in plaintext because a security level was not specified in the request. If the client wants the user information encoded, he must explicitly specify a security level that enforces signature verification.

The only other method in the authentication workflow other than registration that supports encoding is login. The function signature is identical and the token is also of single-use. After login, any further references of `token` in the APIs assume that it is the session token. It is worth noting that, because the token and user key are expected to last as long as the session does, it is pointless to encode, hash, or otherwise obscure the token or user key. Since the same string, encrypted or otherwise, will be sent in each request by the client, it makes no difference to an attacker to sniff the plaintext version or the encrypted version of the token; he can just present the server with the same string and it will be accepted as valid. For similar reasons, the password is being sent in plaintext form to the server and it is only hashed internally.

CHAPTER 4

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

4.1 Types of Contributions

4.1.1 Report Bugs

Report bugs at https://github.com/omtinez/bottleship/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

4.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

4.1.4 Write Documentation

BottleShip could always use more documentation, whether as part of the official BottleShip docs, in docstrings, or even on the web in blog posts, articles, and such.

4.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/omtinez/bottleship/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

4.2 Get Started!

Ready to contribute? Here's how to set up *bottleship* for local development.

1. Fork the *bottleship* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/bottleship.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv bottleship
$ cd bottleship/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 bottleship tests
$ python setup.py test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, 3.3, 3.4 and 3.5, and for PyPy. Check https://travis-ci.org/omtinez/bottleship/pull_requests and make sure that the tests pass for all supported Python versions.

4.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_bottleship
```

CHAPTER 5

Credits

5.1 Project Lead

- Oscar Martinez <omtinez@gmail.com>

5.2 Contributors

None yet. Why not be the first?

CHAPTER 6

History

6.1 0.1.0 (2016-01-17)

- First release on PyPI.

6.2 0.2.0 (2016-01-17)

- Fixed Python 2 vs 3 compatibility.
- Updated documentation and setup TravisCI

6.3 0.2.1 (2016-01-19)

- Update interface with pddb
- Fix line endings

6.4 0.2.2 (2016-02-14)

- Added logout() function and corresponding documentation
- Updated example html file

6.5 0.2.3 (2016-02-15)

- Added logout() tests
- Added callback_success fallback for callback parameter in require_auth()

6.6 0.2.4 (2016-02-17)

- Dynamically import all public methods and classes from bottle

CHAPTER 7

Indices and tables

- genindex
- modindex
- search