Flask-HTTPAuth Documentation

Miguel Grinberg

Apr 27, 2023

Contents

1 Basic authentication examples 3

2 Digest authentication example 5

3 Token Authentication Example 7

4 Using Multiple Authentication Schemes 9

5 User Roles 11

6 Deployment Considerations 13

7 Deprecated Basic Authentication Options 15

8 API Documentation 17

Python Module Index 23

Index 25

Flask-HTTPAuth is a Flask extension that simplifies the use of HTTP authentication with Flask routes.

CHAPTER 1

Basic authentication examples

The following example application uses HTTP Basic authentication to protect route `'/'`:

```
from flask import Flask
from flask_httpauth import HTTPBasicAuth
from werkzeug.security import generate_password_hash, check_password_hash

app = Flask(__name__)
auth = HTTPBasicAuth()

users = {
    "john": generate_password_hash("hello"),
    "susan": generate_password_hash("bye")
}

@auth.verify_password
def verify_password(username, password):
    if username in users and \
            check_password_hash(users.get(username), password):
        return username

@app.route('/')
@auth.login_required
def index():
    return "Hello, {}!".format(auth.current_user())

if __name__ == '__main__':
    app.run()
```

The function decorated with the `verify_password` decorator receives the username and password sent by the client. If the credentials belong to a user, then the function should return the user object. If the credentials are invalid the function can return `None` or `False`. The user object can then be queried from the `current_user()` method of the authentication instance.

CHAPTER 2

Digest authentication example

The following example uses HTTP Digest authentication:

```
from flask import Flask
from flask_httpauth import HTTPDigestAuth

app = Flask(__name__)
app.config['SECRET_KEY'] = 'secret key here'
auth = HTTPDigestAuth()

users = {
    "john": "hello",
    "susan": "bye"
}

@auth.get_password
def get_pw(username):
    if username in users:
        return users.get(username)
    return None

@app.route('/')
@auth.login_required
def index():
    return "Hello, {}!".format(auth.username())

if __name__ == '__main__':
    app.run()
```

CHAPTER 3

Token Authentication Example

The following example application uses a custom HTTP authentication scheme to protect route `'/'` with a token:

```
from flask import Flask
from flask_httpauth import HTTPTokenAuth

app = Flask(__name__)
auth = HTTPTokenAuth(scheme='Bearer')

tokens = {
    "secret-token-1": "john",
    "secret-token-2": "susan"
}

@auth.verify_token
def verify_token(token):
    if token in tokens:
        return tokens[token]

@app.route('/')
@auth.login_required
def index():
    return "Hello, {}!".format(auth.current_user())

if __name__ == '__main__':
    app.run()
```

The `HTTPTokenAuth` is a generic authentication handler that can be used with non-standard authentication schemes, with the scheme name given as an argument in the constructor. In the above example, the `WWW-Authenticate` header provided by the server will use `Bearer` as scheme:

```
WWW-Authenticate: Bearer realm="Authentication Required"
```

The `verify_token` callback receives the authentication credentials provided by the client on the `Authorization` header. This can be a simple token, or can contain multiple arguments, which the function will

have to parse and extract from the string. As with the `verify_password`, the function should return the user object if the token is valid.

In the examples directory you can find a complete example that uses JWS tokens. JWS tokens are similar to JWT tokens. However using JWT tokens would require an external dependency.

CHAPTER 4

Using Multiple Authentication Schemes

Applications sometimes need to support a combination of authentication methods. For example, a web application could be authenticated by sending client id and secret over basic authentication, while third party API clients use a JWS or JWT bearer token. The *MultiAuth* class allows you to protect a route with more than one authentication object. To grant access to the endpoint, one of the authentication methods must validate.

In the examples directory you can find a complete example that uses basic and token authentication.

CHAPTER 5

User Roles

Flask-HTTPAuth includes a simple role-based authentication system that can optionally be added to provide an additional layer of granularity in filtering accesses to routes. To enable role support, write a function that returns the list of roles for a given user and decorate it with the `get_user_roles` decorator:

```
@auth.get_user_roles
def get_user_roles(user):
    return user.get_roles()
```

To restrict access to a route to users having a given role, add the `role` argument to the `login_required` decorator:

```
@app.route('/admin')
@auth.login_required(role='admin')
def admins_only():
    return "Hello {}, you are an admin!".format(auth.current_user())
```

The `role` argument can take a list of roles, in which case users who have any of the given roles will be granted access:

```
@app.route('/admin')
@auth.login_required(role=['admin', 'moderator'])
def admins_only():
    return "Hello {}, you are an admin or a moderator!".format(auth.current_user())
```

In the most advanced usage, users can be filtered by having multiple roles:

```
@app.route('/admin')
@auth.login_required(role=['user', ['moderator', 'contributor']])
def admins_only():
    return "Hello {}, you are a user or a moderator/contributor!".format(auth.current_
↪user())
```

CHAPTER 6

Deployment Considerations

Be aware that some web servers do not pass the `Authorization` headers to the WSGI application by default. For example, if you use Apache with mod_wsgi, you have to set option `WSGIPassAuthorization On` as documented here.

CHAPTER 7

Deprecated Basic Authentication Options

Before the `verify_password` described above existed there were other simpler mechanisms for implementing basic authentication. While these are deprecated they are still maintained. However, the `verify_password` callback should be preferred as it provides greater security and flexibility.

The `get_password` callback needs to return the password associated with the username given as argument. Flask-HTTPAuth will allow access only if `get_password(username) == password`. Example:

```
@auth.get_password
def get_password(username):
    return get_password_for_username(username)
```

Using this callback alone is in general not a good idea because it requires passwords to be available in plaintext in the server. In the more likely scenario that the passwords are stored hashed in a user database, then an additional callback is needed to define how to hash a password:

```
@auth.hash_password
def hash_pw(password):
    return hash_password(password)
```

In this example, you have to replace `hash_password()` with the specific hashing function used in your application. When the `hash_password` callback is provided, access will be granted when `get_password(username) == hash_password(password)`.

If the hashing algorithm requires the username to be known then the callback can take two arguments instead of one:

```
@auth.hash_password
def hash_pw(username, password):
    salt = get_salt(username)
    return hash_password(password, salt)
```

CHAPTER 8

API Documentation

class flask_httpauth.**HTTPBasicAuth**

This class handles HTTP Basic authentication for Flask routes.

__init__(*scheme=None*, *realm=None*)

Create a basic authentication object.

If the optional `scheme` argument is provided, it will be used instead of the standard "Basic" scheme in the `WWW-Authenticate` response. A fairly common practice is to use a custom scheme to prevent browsers from prompting the user to login.

The `realm` argument can be used to provide an application defined realm with the `WWW-Authenticate` header.

verify_password(*verify_password_callback*)

If defined, this callback function will be called by the framework to verify that the username and password combination provided by the client are valid. The callback function takes two arguments, the username and the password. It must return the user object if credentials are valid, or `True` if a user object is not available. In case of failed authentication, it should return `None` or `False`. Example usage:

```
@auth.verify_password
def verify_password(username, password):
    user = User.query.filter_by(username).first()
    if user and passlib.hash.sha256_crypt.verify(password, user.password_
↪hash):
        return user
```

If this callback is defined, it is also invoked when the request does not have the `Authorization` header with user credentials, and in this case both the `username` and `password` arguments are set to empty strings. The application can opt to return `True` in this case and that will allow anonymous users access to the route. The callback function can indicate that the user is anonymous by writing a state variable to `flask.g` or by checking if `auth.current_user()` is `None`.

Note that when a `verify_password` callback is provided the `get_password` and `hash_password` callbacks are not used.

get_user_roles(*roles_callback*)
If defined, this callback function will be called by the framework to obtain the roles assigned to a given user. The callback function takes a single argument, the user for which roles are requested. The user object passed to this function will be the one returned by the "verify" callback. If the verify callback returned `True` instead of a user object, then the `Authorization` object provided by Flask will be passed to this function. The function should return the role or list of roles that belong to the user. Example:

```
@auth.get_user_roles
def get_user_roles(user):
    return user.get_roles()
```

get_password(*password_callback*)
Deprecated This callback function will be called by the framework to obtain the password for a given user. Example:

```
@auth.get_password
def get_password(username):
    return db.get_user_password(username)
```

hash_password(*hash_password_callback*)
Deprecated If defined, this callback function will be called by the framework to apply a custom hashing algorithm to the password provided by the client. If this callback isn't provided the password will be checked unchanged. The callback can take one or two arguments. The one argument version receives the password to hash, while the two argument version receives the username and the password in that order. Example single argument callback:

```
@auth.hash_password
def hash_password(password):
    return md5(password).hexdigest()
```

Example two argument callback:

```
@auth.hash_password
def hash_pw(username, password):
    salt = get_salt(username)
    return hash(password, salt)
```

error_handler(*error_callback*)
If defined, this callback function will be called by the framework when it is necessary to send an authentication error back to the client. The function can take one argument, the status code of the error, which can be 401 (incorrect credentials) or 403 (correct, but insufficient credentials). To preserve compatiiblity with older releases of this package, the function can also be defined without arguments. The return value from this function must by any accepted response type in Flask routes. If this callback isn't provided a default error response is generated. Example:

```
@auth.error_handler
def auth_error(status):
    return "Access Denied", status
```

login_required(*view_function_callback*)
This callback function will be called when authentication is successful. This will typically be a Flask view function. Example:

```
@app.route('/private')
@auth.login_required
def private_page():
    return "Only for authorized people!"
```

An optional `role` argument can be given to further restrict access by roles. Example:

```
@app.route('/private')
@auth.login_required(role='admin')
def private_page():
    return "Only for admins!"
```

An optional `optional` argument can be set to `True` to allow the route to execute also when authentication is not included with the request, in which case `auth.current_user()` will be set to `None`. Example:

```
@app.route('/private')
@auth.login_required(optional=True)
def private_page():
    user = auth.current_user()
    return "Hello {}!".format(user.name if user is not None else 'anonymous')
```

current_user()
The user object returned by the `verify_password` callback on successful authentication. If no user is returned by the callback, this is set to the username passed by the client. Example:

```
@app.route('/')
@auth.login_required
def index():
    user = auth.current_user()
    return "Hello, {}!".format(user.name)
```

username()
Deprecated A view function that is protected with this class can access the logged username through this method. Example:

```
@app.route('/')
@auth.login_required
def index():
    return "Hello, {}!".format(auth.username())
```

class flask_httpauth.**HTTPDigestAuth**
This class handles HTTP Digest authentication for Flask routes. The `SECRET_KEY` configuration must be set in the Flask application to enable the session to work. Flask by default stores user sessions in the client as secure cookies, so the client must be able to handle cookies.

__init__(*self*, *scheme=None*, *realm=None*, *use_ha1_pw=False*, *qop='auth'*, *algorithm='MD5'*)
Create a digest authentication object.

If the optional `scheme` argument is provided, it will be used instead of the "Digest" scheme in the `WWW-Authenticate` response. A fairly common practice is to use a custom scheme to prevent browsers from prompting the user to login.

The `realm` argument can be used to provide an application defined realm with the `WWW-Authenticate` header.

If `use_ha1_pw` is False, then the `get_password` callback needs to return the plain text password for the given user. If `use_ha1_pw` is True, the `get_password` callback needs to return the HA1 value for the given user. The advantage of setting `use_ha1_pw` to `True` is that it allows the application to store the HA1 hash of the password in the user database.

The `qop` option configures a list of accepted quality of protection extensions. This argument can be given as a comma-separated string, a list of strings, or `None` to disable. The default is `auth`. The `auth-int` option is currently not implemented.

The `algorithm` option configures the hash generation algorithm to use. The default is `MD5`. The two algorithms that are implemented are `MD5` and `MD5-Sess`.

generate_ha1(*username*, *password*)
Generate the HA1 hash that can be stored in the user database when `use_ha1_pw` is set to True in the constructor.

generate_nonce(*nonce_making_callback*)
If defined, this callback function will be called by the framework to generate a nonce. If this is defined, `verify_nonce` should also be defined.

This can be used to use a state storage mechanism other than the session.

verify_nonce(*nonce_verify_callback*)
If defined, this callback function will be called by the framework to verify that a nonce is valid. It will be called with a single argument: the nonce to be verified.

This can be used to use a state storage mechanism other than the session.

generate_opaque(*opaque_making_callback*)
If defined, this callback function will be called by the framework to generate an opaque value. If this is defined, `verify_opaque` should also be defined.

This can be used to use a state storage mechanism other than the session.

verify_opaque(*opaque_verify_callback*)
If defined, this callback function will be called by the framework to verify that an opaque value is valid. It will be called with a single argument: the opaque value to be verified.

This can be used to use a state storage mechanism other than the session.

get_password(*password_callback*)
See basic authentication for documentation and examples.

get_user_roles(*roles_callback*)
See basic authentication for documentation and examples.

error_handler(*error_callback*)
See basic authentication for documentation and examples.

login_required(*view_function_callback*)
See basic authentication for documentation and examples.

current_user()
See basic authentication for documentation and examples.

username()
See basic authentication for documentation and examples.

class flask_httpauth.**HTTPTokenAuth**
This class handles HTTP authentication with custom schemes for Flask routes.

__init__(*scheme='Bearer'*, *realm=None*, *header=None*)
Create a token authentication object.

The `scheme` argument can be use to specify the scheme to be used in the `WWW-Authenticate` response. The `Authorization` header sent by the client must include this scheme followed by the token. Example:

```
Authorization: Bearer this-is-my-token
```

The `realm` argument can be used to provide an application defined realm with the `WWW-Authenticate` header.

The `header` argument can be used to specify a custom header instead of `Authorization` from where to obtain the token. If a custom header is used, the `scheme` should not be included. Example:

```
X-API-Key: this-is-my-token
```

verify_token(*verify_token_callback*)
This callback function will be called by the framework to verify that the credentials sent by the client with the `Authorization` header are valid. The callback function takes one argument, the token provided by the client. The function must return the user object if the token is valid, or `True` if a user object is not available. In case of a failed authentication, the function should return `None` or `False`. Example usage:

```
@auth.verify_token
def verify_token(token):
    return User.query.filter_by(token=token).first()
```

Note that a `verify_token` callback is required when using this class.

get_user_roles(*roles_callback*)
See basic authentication for documentation and examples.

error_handler(*error_callback*)
See basic authentication for documentation and examples.

login_required(*view_function_callback*)
See basic authentication for documentation and examples.

current_user()
See basic authentication for documentation and examples.

class flask_httpauth.**HTTPMultiAuth**
This class handles HTTP authentication with custom schemes for Flask routes.

__init__(*auth_object*, ...)
Create a multiple authentication object.

The arguments are one or more instances of `HTTPBasicAuth`, `HTTPDigestAuth` or `HTTPTokenAuth`. A route protected with this authentication method will try all the given authentication objects until one succeeds.

login_required(*view_function_callback*)
See basic authentication for documentation and examples.

current_user()
See basic authentication for documentation and examples.

Python Module Index

f

flask_httpauth, 17

Index

Symbols

__init__() (*flask_httpauth.HTTPBasicAuth method*), 17
__init__() (*flask_httpauth.HTTPDigestAuth method*), 19
__init__() (*flask_httpauth.HTTPMultiAuth method*), 21
__init__() (*flask_httpauth.HTTPTokenAuth method*), 20

C

current_user() (*flask_httpauth.HTTPBasicAuth method*), 19
current_user() (*flask_httpauth.HTTPDigestAuth method*), 20
current_user() (*flask_httpauth.HTTPMultiAuth method*), 21
current_user() (*flask_httpauth.HTTPTokenAuth method*), 21

E

error_handler() (*flask_httpauth.HTTPBasicAuth method*), 18
error_handler() (*flask_httpauth.HTTPDigestAuth method*), 20
error_handler() (*flask_httpauth.HTTPTokenAuth method*), 21

F

flask_httpauth (*module*), 17

G

generate_ha1() (*flask_httpauth.HTTPDigestAuth method*), 20
generate_nonce() (*flask_httpauth.HTTPDigestAuth method*), 20
generate_opaque() (*flask_httpauth.HTTPDigestAuth method*), 20
get_password() (*flask_httpauth.HTTPBasicAuth method*), 18
get_password() (*flask_httpauth.HTTPDigestAuth method*), 20
get_user_roles() (*flask_httpauth.HTTPBasicAuth method*), 17
get_user_roles() (*flask_httpauth.HTTPDigestAuth method*), 20
get_user_roles() (*flask_httpauth.HTTPTokenAuth method*), 21

H

hash_password() (*flask_httpauth.HTTPBasicAuth method*), 18
HTTPBasicAuth (*class in flask_httpauth*), 17
HTTPDigestAuth (*class in flask_httpauth*), 19
HTTPMultiAuth (*class in flask_httpauth*), 21
HTTPTokenAuth (*class in flask_httpauth*), 20

L

login_required() (*flask_httpauth.HTTPBasicAuth method*), 18
login_required() (*flask_httpauth.HTTPDigestAuth method*), 20
login_required() (*flask_httpauth.HTTPMultiAuth method*), 21
login_required() (*flask_httpauth.HTTPTokenAuth method*), 21

U

username() (*flask_httpauth.HTTPBasicAuth method*), 19
username() (*flask_httpauth.HTTPDigestAuth method*), 20

V

verify_nonce() (*flask_httpauth.HTTPDigestAuth method*), 20
verify_opaque() (*flask_httpauth.HTTPDigestAuth method*), 20

verify_password() (*flask_httpauth.HTTPBasicAuth method*), 17
verify_token() (*flask_httpauth.HTTPTokenAuth method*), 21