flask-jwt-extended Documentation

Release 4.6.0

vimalloc rlam3

Mar 01, 2024

CONTENTS

CHAPTER ONE

INSTALLATION

The easiest way to start working with this extension with pip:

```
$ pip install flask-jwt-extended
```

If you want to use asymmetric (public/private) key signing algorithms, include the `asymmetric_crypto` extra requirements.

```
$ pip install flask-jwt-extended[asymmetric_crypto]
```

Note that if you are using ZSH (probably other shells as well), you will need to escape the brackets

```
$ pip install flask-jwt-extended\[asymmetric_crypto\]
```

If you prefer to install from source, you can clone this repo and run

```
$ python setup.py install
```

CHAPTER

TWO

BASIC USAGE

In its simplest form, there is not much to using this extension. You use *create_access_token()* to make JSON Web Tokens, *jwt_required()* to protect routes, and *get_jwt_identity()* to get the identity of a JWT in a protected route.

```python
from flask import Flask
from flask import jsonify
from flask import request

from flask_jwt_extended import create_access_token
from flask_jwt_extended import get_jwt_identity
from flask_jwt_extended import jwt_required
from flask_jwt_extended import JWTManager

app = Flask(__name__)

# Setup the Flask-JWT-Extended extension
app.config["JWT_SECRET_KEY"] = "super-secret"  # Change this!
jwt = JWTManager(app)


# Create a route to authenticate your users and return JWTs. The
# create_access_token() function is used to actually generate the JWT.
@app.route("/login", methods=["POST"])
def login():
    username = request.json.get("username", None)
    password = request.json.get("password", None)
    if username != "test" or password != "test":
        return jsonify({"msg": "Bad username or password"}), 401

    access_token = create_access_token(identity=username)
    return jsonify(access_token=access_token)


# Protect a route with jwt_required, which will kick out requests
# without a valid JWT present.
@app.route("/protected", methods=["GET"])
@jwt_required()
def protected():
    # Access the identity of the current user with get_jwt_identity
    current_user = get_jwt_identity()
```

(continues on next page)

(continued from previous page)

```
    return jsonify(logged_in_as=current_user), 200


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

To access a jwt_required protected view you need to send in the JWT with each request. By default, this is done with an authorization header that looks like:

```
Authorization: Bearer <access_token>
```

We can see this in action using HTTPie.

```
$ http GET :5000/protected

HTTP/1.0 401 UNAUTHORIZED
Content-Length: 39
Content-Type: application/json
Date: Sun, 24 Jan 2021 18:09:17 GMT
Server: Werkzeug/1.0.1 Python/3.8.6

{
    "msg": "Missing Authorization Header"
}


$ http POST :5000/login username=test password=test

HTTP/1.0 200 OK
Content-Length: 288
Content-Type: application/json
Date: Sun, 24 Jan 2021 18:10:39 GMT
Server: Werkzeug/1.0.1 Python/3.8.6

{
    "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
↪eyJmcmVzaCI6ZmFsc2UsImlhdCI6MTYxMTUxMTgzOSwianRpIjoiMmI0NzliNTQtYTI0OS00ZDNjLWE4NjItZGVkZGIzODljNmVlI
↪UpTueBRwNLK8e-06-oo5Y_9eWbaN5T3IHwKsy6Jauaw"
}


$ export JWT="eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
↪eyJmcmVzaCI6ZmFsc2UsImlhdCI6MTYxMTUxMTgzOSwianRpIjoiMmI0NzliNTQtYTI0OS00ZDNjLWE4NjItZGVkZGIzODljNmVlI
↪UpTueBRwNLK8e-06-oo5Y_9eWbaN5T3IHwKsy6Jauaw"


$ http GET :5000/protected Authorization:"Bearer $JWT"

HTTP/1.0 200 OK
Content-Length: 24
Content-Type: application/json
Date: Sun, 24 Jan 2021 18:12:02 GMT
```

(continues on next page)

(continued from previous page)

```
Server: Werkzeug/1.0.1 Python/3.8.6

{
    "logged_in_as": "test"
}
```

Important

Remember to change the JWT secret key in your application, and ensure that it is secure. The JWTs are signed with this key, and if someone gets their hands on it they will be able to create arbitrary tokens that are accepted by your web flask application.

CHAPTER

THREE

AUTOMATIC USER LOADING

In most web applications it is important to have access to the user who is accessing a protected route. We provide a couple callback functions that make this seamless while working with JWTs.

The first is `user_identity_loader()`, which will convert any `User` object used to create a JWT into a JSON serializable format.

On the flip side, you can use `user_lookup_loader()` to automatically load your `User` object when a JWT is present in the request. The loaded user is available in your protected routes via `current_user`.

Lets see an example of this while utilizing SQLAlchemy to store our users:

```python
from hmac import compare_digest

from flask import Flask
from flask import jsonify
from flask import request
from flask_sqlalchemy import SQLAlchemy

from flask_jwt_extended import create_access_token
from flask_jwt_extended import current_user
from flask_jwt_extended import jwt_required
from flask_jwt_extended import JWTManager


app = Flask(__name__)


app.config["JWT_SECRET_KEY"] = "super-secret"  # Change this!
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite://"
app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False


jwt = JWTManager(app)
db = SQLAlchemy(app)


class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.Text, nullable=False, unique=True)
    full_name = db.Column(db.Text, nullable=False)

    # NOTE: In a real application make sure to properly hash and salt passwords
```

(continues on next page)

(continued from previous page)

```
    def check_password(self, password):
        return compare_digest(password, "password")


# Register a callback function that takes whatever object is passed in as the
# identity when creating JWTs and converts it to a JSON serializable format.
@jwt.user_identity_loader
def user_identity_lookup(user):
    return user.id


# Register a callback function that loads a user from your database whenever
# a protected route is accessed. This should return any python object on a
# successful lookup, or None if the lookup failed for any reason (for example
# if the user has been deleted from the database).
@jwt.user_lookup_loader
def user_lookup_callback(_jwt_header, jwt_data):
    identity = jwt_data["sub"]
    return User.query.filter_by(id=identity).one_or_none()


@app.route("/login", methods=["POST"])
def login():
    username = request.json.get("username", None)
    password = request.json.get("password", None)

    user = User.query.filter_by(username=username).one_or_none()
    if not user or not user.check_password(password):
        return jsonify("Wrong username or password"), 401

    # Notice that we are passing in the actual sqlalchemy user object here
    access_token = create_access_token(identity=user)
    return jsonify(access_token=access_token)


@app.route("/who_am_i", methods=["GET"])
@jwt_required()
def protected():
    # We can now access our sqlalchemy User object via `current_user`.
    return jsonify(
        id=current_user.id,
        full_name=current_user.full_name,
        username=current_user.username,
    )


if __name__ == "__main__":
    db.create_all()
    db.session.add(User(full_name="Bruce Wayne", username="batman"))
    db.session.add(User(full_name="Ann Takamaki", username="panther"))
    db.session.add(User(full_name="Jester Lavore", username="little_sapphire"))
    db.session.commit()
```

(continues on next page)

(continued from previous page)

```
    app.run()
```

We can see this in action using HTTPie.

```
$ http POST :5000/login username=panther password=password

HTTP/1.0 200 OK
Content-Length: 281
Content-Type: application/json
Date: Sun, 24 Jan 2021 17:23:31 GMT
Server: Werkzeug/1.0.1 Python/3.8.6

{
    "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
↪eyJmcmVzaCI6ZmFsc2UsImlhdCI6MTYxMTUwOTAxMSwianRpIjoiNGFmN2ViNTAtMjk3Yy00ZmY4LWJmOTYtMTZlMDE5MWEzYzMwI
↪2UhZo-xo19NXaqKLwcMz0NBLAcxxEUeK4Ziqk1T_9h0"
}


$ export JWT="eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
↪eyJmcmVzaCI6ZmFsc2UsImlhdCI6MTYxMTUwOTAxMSwianRpIjoiNGFmN2ViNTAtMjk3Yy00ZmY4LWJmOTYtMTZlMDE5MWEzYzMwI
↪2UhZo-xo19NXaqKLwcMz0NBLAcxxEUeK4Ziqk1T_9h0"


$ http GET :5000/who_am_i Authorization:"Bearer $JWT"

HTTP/1.0 200 OK
Content-Length: 57
Content-Type: application/json
Date: Sun, 24 Jan 2021 17:31:34 GMT
Server: Werkzeug/1.0.1 Python/3.8.6

{
    "id": 2,
    "full_name": "Ann Takamaki",
    "username": "panther"
}
```

CHAPTER FOUR

STORING ADDITIONAL DATA IN JWTS

You may want to store additional information in the access token which you could later access in the protected views. This can be done using the `additional_claims` argument with the *`create_access_token()`* or *`create_refresh_token()`* functions. The claims can be accessed in a protected route via the *`get_jwt()`* function.

It is important to remember that JWTs are not encrypted and the contents of a JWT can be trivially decoded by anyone who has access to it. As such, you should never put any sensitive information in a JWT.

```python
from flask import Flask
from flask import jsonify
from flask import request

from flask_jwt_extended import create_access_token
from flask_jwt_extended import get_jwt
from flask_jwt_extended import jwt_required
from flask_jwt_extended import JWTManager

app = Flask(__name__)

app.config["JWT_SECRET_KEY"] = "super-secret"  # Change this!
jwt = JWTManager(app)


@app.route("/login", methods=["POST"])
def login():
    username = request.json.get("username", None)
    password = request.json.get("password", None)
    if username != "test" or password != "test":
        return jsonify({"msg": "Bad username or password"}), 401

    # You can use the additional_claims argument to either add
    # custom claims or override default claims in the JWT.
    additional_claims = {"aud": "some_audience", "foo": "bar"}
    access_token = create_access_token(username, additional_claims=additional_claims)
    return jsonify(access_token=access_token)


# In a protected view, get the claims you added to the jwt with the
# get_jwt() method
@app.route("/protected", methods=["GET"])
@jwt_required()
def protected():
```

(continues on next page)

(continued from previous page)

```
    claims = get_jwt()
    return jsonify(foo=claims["foo"])


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

Alternately you can use the *additional_claims_loader()* decorator to register a callback function that will be called whenever a new JWT is created, and return a dictionary of claims to add to that token. In the case that both *additional_claims_loader()* and the `additional_claims` argument are used, both results are merged together, with ties going to the data supplied by the `additional_claims` argument.

```
# Using the additional_claims_loader, we can specify a method that will be
# called when creating JWTs. The decorated method must take the identity
# we are creating a token for and return a dictionary of additional
# claims to add to the JWT.
@jwt.additional_claims_loader
def add_claims_to_access_token(identity):
    return {
        "aud": "some_audience",
        "foo": "bar",
        "upcase_name": identity.upper(),
    }
```

CHAPTER FIVE

PARTIALLY PROTECTING ROUTES

There may be cases where you want to use the same route regardless of if a JWT is present in the request or not. In these situations, you can use *jwt_required()* with the `optional=True` argument. This will allow the endpoint to be accessed regardless of if a JWT is sent in with the request.

If no JWT is present, *get_jwt()* and *get_jwt_header()*, will return an empty dictionary. *get_jwt_identity()*, *current_user*, and *get_current_user()* will return None.

If a JWT that is expired or not verifiable is in the request, an error will be still returned like normal.

```python
from flask import Flask
from flask import jsonify
from flask import request

from flask_jwt_extended import create_access_token
from flask_jwt_extended import get_jwt_identity
from flask_jwt_extended import jwt_required
from flask_jwt_extended import JWTManager

app = Flask(__name__)

# Setup the Flask-JWT-Extended extension
app.config["JWT_SECRET_KEY"] = "super-secret"  # Change this!
jwt = JWTManager(app)


@app.route("/login", methods=["POST"])
def login():
    username = request.json.get("username", None)
    password = request.json.get("password", None)
    if username != "test" or password != "test":
        return jsonify({"msg": "Bad username or password"}), 401

    access_token = create_access_token(identity=username)
    return jsonify(access_token=access_token)


@app.route("/optionally_protected", methods=["GET"])
@jwt_required(optional=True)
def optionally_protected():
    current_identity = get_jwt_identity()
    if current_identity:
```

(continues on next page)

(continued from previous page)

```
        return jsonify(logged_in_as=current_identity)
    else:
        return jsonify(logged_in_as="anonymous user")


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

CHAPTER SIX

JWT LOCATIONS

JWTs can be sent in with a request in many different ways. You can control which ways you want to accept JWTs in your Flask application via the `JWT_TOKEN_LOCATION` *configuration option*. You can also override that global configuration on a per route basis via the `locations` argument in *`jwt_required()`*.

```python
from flask import Flask
from flask import jsonify

from flask_jwt_extended import create_access_token
from flask_jwt_extended import jwt_required
from flask_jwt_extended import JWTManager
from flask_jwt_extended import set_access_cookies
from flask_jwt_extended import unset_jwt_cookies

app = Flask(__name__)

# Here you can globally configure all the ways you want to allow JWTs to
# be sent to your web application. By default, this will be only headers.
app.config["JWT_TOKEN_LOCATION"] = ["headers", "cookies", "json", "query_string"]

# If true this will only allow the cookies that contain your JWTs to be sent
# over https. In production, this should always be set to True
app.config["JWT_COOKIE_SECURE"] = False

# Change this in your code!
app.config["JWT_SECRET_KEY"] = "super-secret"

jwt = JWTManager(app)


@app.route("/login_without_cookies", methods=["POST"])
def login_without_cookies():
    access_token = create_access_token(identity="example_user")
    return jsonify(access_token=access_token)


@app.route("/login_with_cookies", methods=["POST"])
def login_with_cookies():
    response = jsonify({"msg": "login successful"})
    access_token = create_access_token(identity="example_user")
    set_access_cookies(response, access_token)
```

(continues on next page)

(continued from previous page)

```
    return response


@app.route("/logout_with_cookies", methods=["POST"])
def logout_with_cookies():
    response = jsonify({"msg": "logout successful"})
    unset_jwt_cookies(response)
    return response


@app.route("/protected", methods=["GET", "POST"])
@jwt_required()
def protected():
    return jsonify(foo="bar")


@app.route("/only_headers")
@jwt_required(locations=["headers"])
def only_headers():
    return jsonify(foo="baz")


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

Lets take a look at how you could utilize all of these locations using some javascript in a web browser.

6.1 Headers

Working JWTs via headers is a pretty simple process. All you need to do is store the token when you login, and add the token as a header each time you make a request to a protected route. Logging out is as simple as deleting the token.

```
async function login() {
  const response = await fetch('/login_without_cookies', {method: 'post'});
  const result = await response.json();
  localStorage.setItem('jwt', result.access_token);
}

function logout() {
  localStorage.removeItem('jwt');
}

async function makeRequestWithJWT() {
  const options = {
    method: 'post',
    headers: {
      Authorization: `Bearer ${localStorage.getItem('jwt')}`,
    }
  };
  const response = await fetch('/protected', options);
  const result = await response.json();
```

(continues on next page)

(continued from previous page)

```
  return result;
}
```

6.2 Cookies

Cookies are a fantastic way of handling JWTs if you are using a web browser. They offer some nice benefits compared to the headers approach:

- They can be configured to send only over HTTPS. This prevents a JWT from accidentally being sent, and possibly compromised, over an unsecure connection.
- They are stored in an http-only cookie, which prevents XSS attacks from being able to steal the underlying JWT.
- You Flask application can implicitly refresh JWTs that are close to expiring, which simplifies the logic of keeping active users logged in. More on this in the next section!

Of course, when using cookies you also need to do some additional work to prevent Cross Site Request Forgery (CSRF) attacks. In this extension we handle this via something called double submit verification.

The basic idea behind double submit verification is that a JWT coming from a cookie will only be considered valid if a special double submit token is also present in the request, and that double submit token must not be something that is automatically sent by a web browser (ie it cannot be another cookie).

By default, we accomplish this by setting two cookies when someone logging in. The first cookie contains the JWT, and encoded in that JWT is the double submit token. This cookie is set as http-only, so that it cannot be access via javascript (this is what prevents XSS attacks from being able to steal the JWT). The second cookie we set contains only the same double submit token, but this time in a cookie that is readable by javascript. Whenever a request is made, it needs to include an `X-CSRF-TOKEN` header, with the value of the double submit token. If the value in this header does not match the value stored in the JWT, the request is kicked out as invalid.

Because the double submit token needs to be present as a header (which wont be automatically sent on a request), and some malicious javascript running on a different domain will not be able to read the cookie containing the double submit token on your website, we have successfully thwarted any CSRF attacks.

This does mean that whenever you are making a request, you need to manually include the `X-CSRF-TOKEN` header, otherwise your requests will be kicked out as invalid too. Lets look at how to do that:

```
async function login() {
  await fetch('/login_with_cookies', {method: 'post'});
}

async function logout() {
  await fetch('/logout_with_cookies', {method: 'post'});
}

function getCookie(name) {
  const value = `; ${document.cookie}`;
  const parts = value.split(`; ${name}=`);
  if (parts.length === 2) return parts.pop().split(';').shift();
}

async function makeRequestWithJWT() {
  const options = {
    method: 'post',
```

(continues on next page)

(continued from previous page)

```
    credentials: 'same-origin',
    headers: {
      'X-CSRF-TOKEN': getCookie('csrf_access_token'),
    },
  };
  const response = await fetch('/protected', options);
  const result = await response.json();
  return result;
}
```

Note that there are additional CSRF options, such as looking for the double submit token in a form, changing cookie paths, etc, that can be used to tailor things to the needs of your application. See *Cross Site Request Forgery Options:* for details.

6.3 Query String

You can also send in the JWT as part of the query string. However, It is very important to note that in most cases we recommend *NOT* doing this. It can lead to some non-obvious security issues, such as saving the JWT in a browsers history or the JWT being logged in your backend server, which could both potentially lead to a compromised token. However, this feature might provide some limited usefulness, such as sending password reset links, and therefore we support it in this extension.

```
async function login() {
  const response = await fetch('/login_without_cookies', {method: 'post'});
  const result = await response.json();
  localStorage.setItem('jwt', result.access_token);
}

function logout() {
  localStorage.removeItem('jwt');
}

async function makeRequestWithJWT() {
  const jwt = localStorage.getItem('jwt')
  const response = await fetch(`/protected?jwt=${jwt}`, {method: 'post'});
  const result = await response.json();
  return result;
}
```

6.4 JSON Body

This looks very similar to the Headers approach, except that we send the JWT in as part of the JSON Body instead of a header. Be aware that HEAD or GET requests cannot have a JSON body as part of the request, so this only works for actions like POST/PUT/PATCH/DELETE/etc.

Sending JWTs in a JSON body is probably not very useful most of the time, but we include the option for it regardless.

```
async function login() {
  const response = await fetch('/login_without_cookies', {method: 'post'});
```

(continues on next page)

(continued from previous page)

```
  const result = await response.json();
  localStorage.setItem('jwt', result.access_token);
}

function logout() {
  localStorage.removeItem('jwt');
}

// Note that if we change the method to `get` this will blow up with a
// "TypeError: Window.fetch: HEAD or GET Request cannot have a body"
async function makeRequestWithJWT() {
  const options = {
    method: 'post',
    body: JSON.stringify({access_token: localStorage.getItem('jwt')}),
    headers: {
      'Content-Type': 'application/json',
    },
  };
  const response = await fetch('/protected', options);
  const result = await response.json();
  return result;
}
```

CHAPTER

SEVEN

REFRESHING TOKENS

In most web applications, it would not be ideal if a user was logged out in the middle of doing something because their JWT expired. Unfortunately we can't just change the expires time on a JWT on each request, as once a JWT is created it cannot be modified. Lets take a look at some options for solving this problem by refreshing JWTs.

7.1 Implicit Refreshing With Cookies

One huge benefit to storing your JWTs in cookies (when your frontend is a website) is that the frontend does not have to handle any logic when it comes to refreshing a token. It can all happen implicitly with the cookies your Flask application sets.

The basic idea here is that at the end of every request, we will check if there is a JWT that is close to expiring. If we find a JWT that is nearly expired, we will replace the current cookie containing the JWT with a new JWT that has a longer time until it expires.

This is our recommended approach when your frontend is a website.

```python
from datetime import datetime
from datetime import timedelta
from datetime import timezone

from flask import Flask
from flask import jsonify

from flask_jwt_extended import create_access_token
from flask_jwt_extended import get_jwt
from flask_jwt_extended import get_jwt_identity
from flask_jwt_extended import jwt_required
from flask_jwt_extended import JWTManager
from flask_jwt_extended import set_access_cookies
from flask_jwt_extended import unset_jwt_cookies

app = Flask(__name__)

# If true this will only allow the cookies that contain your JWTs to be sent
# over https. In production, this should always be set to True
app.config["JWT_COOKIE_SECURE"] = False
app.config["JWT_TOKEN_LOCATION"] = ["cookies"]
app.config["JWT_SECRET_KEY"] = "super-secret"  # Change this in your code!
app.config["JWT_ACCESS_TOKEN_EXPIRES"] = timedelta(hours=1)
```

(continues on next page)

(continued from previous page)

```python
jwt = JWTManager(app)


# Using an `after_request` callback, we refresh any token that is within 30
# minutes of expiring. Change the timedeltas to match the needs of your application.
@app.after_request
def refresh_expiring_jwts(response):
    try:
        exp_timestamp = get_jwt()["exp"]
        now = datetime.now(timezone.utc)
        target_timestamp = datetime.timestamp(now + timedelta(minutes=30))
        if target_timestamp > exp_timestamp:
            access_token = create_access_token(identity=get_jwt_identity())
            set_access_cookies(response, access_token)
        return response
    except (RuntimeError, KeyError):
        # Case where there is not a valid JWT. Just return the original response
        return response


@app.route("/login", methods=["POST"])
def login():
    response = jsonify({"msg": "login successful"})
    access_token = create_access_token(identity="example_user")
    set_access_cookies(response, access_token)
    return response


@app.route("/logout", methods=["POST"])
def logout():
    response = jsonify({"msg": "logout successful"})
    unset_jwt_cookies(response)
    return response


@app.route("/protected")
@jwt_required()
def protected():
    return jsonify(foo="bar")


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

7.2 Explicit Refreshing With Refresh Tokens

Alternatively, this extension comes out of the box with refresh token support. A refresh token is a long lived JWT that can only be used to creating new access tokens.

You have a couple choices about how to utilize a refresh token. You could store the expires time of your access token on your frontend, and each time you make an API request first check if the current access token is near or already expired, and refresh it as needed. This approach is pretty simple and will work fine in most cases, but do be aware that if your frontend has a clock that is significantly off, you might run into issues.

An alternative approach involves making an API request with your access token and then checking the result to see if it worked. If the result of the request is an error message saying that your token is expired, use the refresh token to generate a new access token and redo the request with the new token. This approach will work regardless of the clock on your frontend, but it does require having some potentially more complicated logic.

Using refresh tokens is our recommended approach when your frontend is not a website (mobile, api only, etc).

```
from datetime import timedelta

from flask import Flask
from flask import jsonify

from flask_jwt_extended import create_access_token
from flask_jwt_extended import create_refresh_token
from flask_jwt_extended import get_jwt_identity
from flask_jwt_extended import jwt_required
from flask_jwt_extended import JWTManager

app = Flask(__name__)

app.config["JWT_SECRET_KEY"] = "super-secret"  # Change this!
app.config["JWT_ACCESS_TOKEN_EXPIRES"] = timedelta(hours=1)
app.config["JWT_REFRESH_TOKEN_EXPIRES"] = timedelta(days=30)
jwt = JWTManager(app)


@app.route("/login", methods=["POST"])
def login():
    access_token = create_access_token(identity="example_user")
    refresh_token = create_refresh_token(identity="example_user")
    return jsonify(access_token=access_token, refresh_token=refresh_token)


# We are using the `refresh=True` options in jwt_required to only allow
# refresh tokens to access this route.
@app.route("/refresh", methods=["POST"])
@jwt_required(refresh=True)
def refresh():
    identity = get_jwt_identity()
    access_token = create_access_token(identity=identity)
    return jsonify(access_token=access_token)


@app.route("/protected", methods=["GET"])
```

(continues on next page)

(continued from previous page)

```
@jwt_required()
def protected():
    return jsonify(foo="bar")


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

Making a request with a refresh token looks just like making a request with an access token. Here is an example using HTTPie.

```
$ http POST :5000/refresh Authorization:"Bearer $REFRESH_TOKEN"
```

Warning: Note that when an access token is invalidated (e.g. logging a user out), any corresponding refresh token(s) must be revoked too. See *Revoking Refresh Tokens* for details on how to handle this.

7.3 Token Freshness Pattern

The token freshness pattern is a very simple idea. Every time a user authenticates by providing a username and password, they receive a `fresh` access token that can access any route. But after some time, that token should no longer be considered `fresh`, and some critical or dangerous routes will be blocked until the user verifies their password again. All other routes will still work normally for the user even though their token is no longer `fresh`. As an example, we might not allow users to change their email address unless they have a `fresh` token, but we do allow them use the rest of our Flask application normally.

The token freshness pattern is built into this extension, and works seamlessly with both token refreshing strategies discussed above. Lets take a look at this with the explicit refresh example (it will look basically same in the implicit refresh example).

```
from datetime import timedelta

from flask import Flask
from flask import jsonify
from flask import request

from flask_jwt_extended import create_access_token
from flask_jwt_extended import create_refresh_token
from flask_jwt_extended import get_jwt_identity
from flask_jwt_extended import jwt_required
from flask_jwt_extended import JWTManager

app = Flask(__name__)

app.config["JWT_SECRET_KEY"] = "super-secret"  # Change this!
app.config["JWT_ACCESS_TOKEN_EXPIRES"] = timedelta(hours=1)
app.config["JWT_REFRESH_TOKEN_EXPIRES"] = timedelta(days=30)
jwt = JWTManager(app)
```

(continues on next page)

(continued from previous page)

```python
# We verify the users password here, so we are returning a fresh access token
@app.route("/login", methods=["POST"])
def login():
    username = request.json.get("username", None)
    password = request.json.get("password", None)
    if username != "test" or password != "test":
        return jsonify({"msg": "Bad username or password"}), 401

    access_token = create_access_token(identity="example_user", fresh=True)
    refresh_token = create_refresh_token(identity="example_user")
    return jsonify(access_token=access_token, refresh_token=refresh_token)


# If we are refreshing a token here we have not verified the users password in
# a while, so mark the newly created access token as not fresh
@app.route("/refresh", methods=["POST"])
@jwt_required(refresh=True)
def refresh():
    identity = get_jwt_identity()
    access_token = create_access_token(identity=identity, fresh=False)
    return jsonify(access_token=access_token)


# Only allow fresh JWTs to access this route with the `fresh=True` arguement.
@app.route("/protected", methods=["GET"])
@jwt_required(fresh=True)
def protected():
    return jsonify(foo="bar")


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

We also support marking a token as fresh for a given amount of time after it is created. You can do this by passing a `datetime.timedelta` to the `fresh` option when creating JWTs:

```python
create_access_token(identity, fresh=datetime.timedelta(minutes=15))
```

CHAPTER

EIGHT

JWT REVOKING / BLOCKLIST

JWT revoking is a mechanism for preventing an otherwise valid JWT from accessing your routes while still letting other valid JWTs in. To utilize JWT revoking in this extension, you must define a callback function via the *token_in_blocklist_loader()* decorator. This function is called whenever a valid JWT is used to access a protected route. The callback will receive the JWT header and JWT payload as arguments, and must return `True` if the JWT has been revoked.

In production, you will want to use some form of persistent storage (database, redis, etc) to store your JWTs. It would be bad if your application forgot that a JWT was revoked if it was restarted. We can provide some general recommendations on what type of storage engine to use, but ultimately the choice will depend on your specific application and tech stack.

8.1 Redis

If your only requirements are to check if a JWT has been revoked, our recommendation is to use redis. It is blazing fast, can be configured to persist data to disc, and can automatically clear out JWTs after they expire by utilizing the Time To Live (TTL) functionality when storing a JWT. Here is an example using redis:

```python
from datetime import timedelta

import redis
from flask import Flask
from flask import jsonify

from flask_jwt_extended import create_access_token
from flask_jwt_extended import get_jwt
from flask_jwt_extended import jwt_required
from flask_jwt_extended import JWTManager

ACCESS_EXPIRES = timedelta(hours=1)

app = Flask(__name__)
app.config["JWT_SECRET_KEY"] = "super-secret"  # Change this!
app.config["JWT_ACCESS_TOKEN_EXPIRES"] = ACCESS_EXPIRES
jwt = JWTManager(app)

# Setup our redis connection for storing the blocklisted tokens. You will probably
# want your redis instance configured to persist data to disk, so that a restart
# does not cause your application to forget that a JWT was revoked.
jwt_redis_blocklist = redis.StrictRedis(
    host="localhost", port=6379, db=0, decode_responses=True
```

(continues on next page)

(continued from previous page)

```
)


# Callback function to check if a JWT exists in the redis blocklist
@jwt.token_in_blocklist_loader
def check_if_token_is_revoked(jwt_header, jwt_payload: dict):
    jti = jwt_payload["jti"]
    token_in_redis = jwt_redis_blocklist.get(jti)
    return token_in_redis is not None


@app.route("/login", methods=["POST"])
def login():
    access_token = create_access_token(identity="example_user")
    return jsonify(access_token=access_token)


# Endpoint for revoking the current users access token. Save the JWTs unique
# identifier (jti) in redis. Also set a Time to Live (TTL)  when storing the JWT
# so that it will automatically be cleared out of redis after the token expires.
@app.route("/logout", methods=["DELETE"])
@jwt_required()
def logout():
    jti = get_jwt()["jti"]
    jwt_redis_blocklist.set(jti, "", ex=ACCESS_EXPIRES)
    return jsonify(msg="Access token revoked")


# A blocklisted access token will not be able to access this any more
@app.route("/protected", methods=["GET"])
@jwt_required()
def protected():
    return jsonify(hello="world")


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

Warning: Note that configuring redis to be disk-persistent is an absolutely necessity for production use. Otherwise, events like power outages or server crashes/reboots would cause all invalidated tokens to become valid again (assuming the secret key does not change). This is especially concering for long-lived refresh tokens, discussed below.

8.2 Database

If you need to keep track of information about revoked JWTs our recommendation is to utilize a database. This allows you to easily store and utilize metadata for revoked tokens, such as when it was revoked, who revoked it, can it be un-revoked, etc. Here is an example using SQLAlchemy:

```
from datetime import datetime
from datetime import timedelta
from datetime import timezone

from flask import Flask
from flask import jsonify
from flask_sqlalchemy import SQLAlchemy

from flask_jwt_extended import create_access_token
from flask_jwt_extended import get_jwt
from flask_jwt_extended import jwt_required
from flask_jwt_extended import JWTManager

app = Flask(__name__)

ACCESS_EXPIRES = timedelta(hours=1)
app.config["JWT_SECRET_KEY"] = "super-secret"  # Change this!
app.config["JWT_ACCESS_TOKEN_EXPIRES"] = ACCESS_EXPIRES
jwt = JWTManager(app)

# We are using an in memory database here as an example. Make sure to use a
# database with persistent storage in production!
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite://"
app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False
db = SQLAlchemy(app)


# This could be expanded to fit the needs of your application. For example,
# it could track who revoked a JWT, when a token expires, notes for why a
# JWT was revoked, an endpoint to un-revoked a JWT, etc.
# Making jti an index can significantly speed up the search when there are
# tens of thousands of records. Remember this query will happen for every
# (protected) request,
# If your database supports a UUID type, this can be used for the jti column
# as well
class TokenBlocklist(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    jti = db.Column(db.String(36), nullable=False, index=True)
    created_at = db.Column(db.DateTime, nullable=False)


# Callback function to check if a JWT exists in the database blocklist
@jwt.token_in_blocklist_loader
def check_if_token_revoked(jwt_header, jwt_payload: dict) -> bool:
    jti = jwt_payload["jti"]
    token = db.session.query(TokenBlocklist.id).filter_by(jti=jti).scalar()
```

(continues on next page)

(continued from previous page)

```
    return token is not None


@app.route("/login", methods=["POST"])
def login():
    access_token = create_access_token(identity="example_user")
    return jsonify(access_token=access_token)


# Endpoint for revoking the current users access token. Saved the unique
# identifier (jti) for the JWT into our database.
@app.route("/logout", methods=["DELETE"])
@jwt_required()
def modify_token():
    jti = get_jwt()["jti"]
    now = datetime.now(timezone.utc)
    db.session.add(TokenBlocklist(jti=jti, created_at=now))
    db.session.commit()
    return jsonify(msg="JWT revoked")


# A blocklisted access token will not be able to access this any more
@app.route("/protected", methods=["GET"])
@jwt_required()
def protected():
    return jsonify(hello="world")


if __name__ == "__main__":
    db.create_all()
    app.run()
```

8.3 Revoking Refresh Tokens

It is critical to note that a user's refresh token must also be revoked when logging out; otherwise, this refresh token could just be used to generate a new access token. Usually this falls to the responsibility of the frontend application, which must send two separate requests to the backend in order to revoke these tokens.

This can be implemented via two separate routes marked with `@jwt_required()` and `@jwt_required(refresh=True)` to revoke access and refresh tokens, respectively. However, it is more convenient to provide a single endpoint where the frontend can send a DELETE for each token. The following is an example:

```
@app.route("/logout", methods=["DELETE"])
@jwt_required(verify_type=False)
def logout():
    token = get_jwt()
    jti = token["jti"]
    ttype = token["type"]
    jwt_redis_blocklist.set(jti, "", ex=ACCESS_EXPIRES)
```

(continues on next page)

(continued from previous page)

```
    # Returns "Access token revoked" or "Refresh token revoked"
    return jsonify(msg=f"{ttype.capitalize()} token successfully revoked")
```

or, for the database format:

```
class TokenBlocklist(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    jti = db.Column(db.String(36), nullable=False, index=True)
    type = db.Column(db.String(16), nullable=False)
    user_id = db.Column(
        db.ForeignKey('person.id'),
        default=lambda: get_current_user().id,
        nullable=False,
    )
    created_at = db.Column(
        db.DateTime,
        server_default=func.now(),
        nullable=False,
    )

@app.route("/logout", methods=["DELETE"])
@jwt_required(verify_type=False)
def modify_token():
    token = get_jwt()
    jti = token["jti"]
    ttype = token["type"]
    now = datetime.now(timezone.utc)
    db.session.add(TokenBlocklist(jti=jti, type=ttype, created_at=now))
    db.session.commit()
    return jsonify(msg=f"{ttype.capitalize()} token successfully revoked")
```

Token type and user columns are not required and can be omitted. That being said, including these can help to audit that the frontend is performing its revoking job correctly and revoking both tokens.

Alternatively, there are a few ways to revoke both tokens at once:

1. Send the access token in the header (per usual), and send the refresh token in the DELETE request body. This saves a request but still needs frontend changes, so may not be worth implementing
2. Embed the refresh token's jti in the access token. The revoke route should be authenticated with the access token. Upon revoking the access token, extract the refresh jti from it and invalidate both. This has the advantage of requiring no extra work from the frontend.
3. Store every generated tokens jti in a database upon creation. Have a boolean column to represent whether it is valid or not, which the `token_in_blocklist_loader` should respond based upon. Upon revoking a token, mark that token row as invalid, as well as all other tokens from the same user generated at the same time. This would also allow for a "log out everywhere" option where all tokens for a user are invalidated at once, which is otherwise not easily possibile

The best option of course depends and needs to be chosen based upon the circumstances. If there if ever a time where an unknown, untracked token needs to be immediately invalidated, this can be accomplished by changing the secret key.

CHAPTER

NINE

CONFIGURATION OPTIONS

You can change many options for this extension works via Flask's Configuration Handling. For example:

```
app.config["OPTION_NAME"] = option_value
```

9.1 Overview:

- *General Options:*
    - *JWT_ACCESS_TOKEN_EXPIRES*
    - *JWT_ALGORITHM*
    - *JWT_DECODE_ALGORITHMS*
    - *JWT_DECODE_AUDIENCE*
    - *JWT_DECODE_ISSUER*
    - *JWT_DECODE_LEEWAY*
    - *JWT_ENCODE_AUDIENCE*
    - *JWT_ENCODE_ISSUER*
    - *JWT_ENCODE_NBF*
    - *JWT_ERROR_MESSAGE_KEY*
    - *JWT_IDENTITY_CLAIM*
    - *JWT_PRIVATE_KEY*
    - *JWT_PUBLIC_KEY*
    - *JWT_REFRESH_TOKEN_EXPIRES*
    - *JWT_SECRET_KEY*
    - *JWT_TOKEN_LOCATION*
- *Header Options:*
    - *JWT_HEADER_NAME*
    - *JWT_HEADER_TYPE*
- *Cookie Options:*
    - *JWT_ACCESS_COOKIE_NAME*

  - *JWT_ACCESS_COOKIE_PATH*
  - *JWT_COOKIE_CSRF_PROTECT*
  - *JWT_COOKIE_DOMAIN*
  - *JWT_COOKIE_SAMESITE*
  - *JWT_COOKIE_SECURE*
  - *JWT_REFRESH_COOKIE_NAME*
  - *JWT_REFRESH_COOKIE_PATH*
  - *JWT_SESSION_COOKIE*

- *Cross Site Request Forgery Options:*
  - *JWT_ACCESS_CSRF_COOKIE_NAME*
  - *JWT_ACCESS_CSRF_COOKIE_PATH*
  - *JWT_ACCESS_CSRF_FIELD_NAME*
  - *JWT_ACCESS_CSRF_HEADER_NAME*
  - *JWT_CSRF_CHECK_FORM*
  - *JWT_CSRF_IN_COOKIES*
  - *JWT_CSRF_METHODS*
  - *JWT_REFRESH_CSRF_COOKIE_NAME*
  - *JWT_REFRESH_CSRF_COOKIE_PATH*
  - *JWT_REFRESH_CSRF_FIELD_NAME*
  - *JWT_REFRESH_CSRF_HEADER_NAME*
- *Query String Options:*
  - *JWT_QUERY_STRING_NAME*
  - *JWT_QUERY_STRING_VALUE_PREFIX*
- *JSON Body Options:*
  - *JWT_JSON_KEY*
  - *JWT_REFRESH_JSON_KEY*

9.2 General Options:

`JWT_ACCESS_TOKEN_EXPIRES`

How long an access token should be valid before it expires. This can be a datetime.timedelta, dateutil.relativedelta, or a number of seconds (`Integer`).

If set to `False` tokens will never expire. **This is dangerous and should be avoided in most case**

This can be overridden on a per token basis by passing the `expires_delta` argument to *`flask_jwt_extended.create_access_token()`*

Default: `datetime.timedelta(minutes=15)`

`JWT_ALGORITHM`

Which algorithm to sign the JWT with. See PyJWT for the available algorithms.

Default: `"HS256"`

`JWT_DECODE_ALGORITHMS`

Which algorithms to use when decoding a JWT. See PyJWT for the available algorithms.

By default this will always be the same algorithm that is defined in `JWT_ALGORITHM`.

Default: `["HS256"]`

`JWT_DECODE_AUDIENCE`

The string or list of audiences (`aud`) expected in a JWT when decoding it.

Default: `None`

`JWT_DECODE_ISSUER`

The issuer (`iss`) you expect in a JWT when decoding it.

Default: `None`

`JWT_DECODE_LEEWAY`

The number of seconds a token will be considered valid before the Not Before Time (*nbf) and after the Expires Time (`exp*). This can be useful when dealing with clock drift between clients.

Default: `0`

`JWT_ENCODE_AUDIENCE`

The string or list of audiences (`aud`) for created JWTs.

Default: `None`

`JWT_ENCODE_ISSUER`

The issuer (`iss`) for created JWTs.

Default: `None`

`JWT_ENCODE_NBF`

The not before (`nbf`) claim which defines that a JWT MUST NOT be accepted for processing during decode.

Default: `True`

`JWT_ERROR_MESSAGE_KEY`

The key for error messages in a JSON response returned by this extension.

Default: `"msg"`

`JWT_IDENTITY_CLAIM`

The claim in a JWT that is used as the source of identity.

Default: `"sub"`

`JWT_PRIVATE_KEY`

The secret key used to encode JWTs when using an asymmetric signing algorithm (such as `RS*` or `ES*`). The key must be in PEM format.

Do not reveal the secret key when posting questions or committing code.

Default: `None`

JWT_PUBLIC_KEY

The secret key used to decode JWTs when using an asymmetric signing algorithm (such as `RS*` or `ES*`). The key must be in PEM format.

Default: `None`

JWT_REFRESH_TOKEN_EXPIRES

How long a refresh token should be valid before it expires. This can be a datetime.timedelta, dateutil.relativedelta, or a number of seconds (`Integer`).

If set to `False` tokens will never expire. **This is dangerous and should be avoided in most case**

This can be overridden on a per token basis by passing the `expires_delta` argument to *`flask_jwt_extended.create_refresh_token()`*

Default: `datetime.timedelta(days=30)`

JWT_SECRET_KEY

The secret key used to encode and decode JWTs when using a symmetric signing algorithm (such as `HS*`). It should be a long random string of bytes, although unicode is accepted too. For example, copy the output of this to your config.

```
$ python -c 'import os; print(os.urandom(16))'
b'_5#y2L"F4Q8z\n\xec]/'
```

If this value is not set, Flask's SECRET_KEY is used instead.

Do not reveal the secret key when posting questions or committing code.

Note: there is ever a need to invalidate all issued tokens (e.g. a security flaw was found, or the revoked token database was lost), this can be easily done by changing the JWT_SECRET_KEY (or Flask's SECRET_KEY, if JWT_SECRET_KEY is unset).

Default: `None`

JWT_TOKEN_LOCATION

Where to look for a JWT when processing a request. The available options are `"headers"`, `"cookies"`, `"query_string"`, and `"json"`.

You can pass in a list to check more then one location, for example `["headers", "cookies"]`. The order of the list sets the precedence of where JWTs will be looked for.

This can be overridden on a per-route basis by using the `locations` argument in *`flask_jwt_extended.jwt_required()`*.

Default: `"headers"`

9.3 Header Options:

These are only applicable if a route is configured to accept JWTs via headers.

JWT_HEADER_NAME

What header should contain the JWT in a request

Default: `"Authorization"`

JWT_HEADER_TYPE

What type of header the JWT is in. If this is an empty string, the header should contain nothing besides the JWT.

Default: `"Bearer"`

9.4 Cookie Options:

These are only applicable if a route is configured to accept JWTs via cookies.

`JWT_ACCESS_COOKIE_NAME`

The name of the cookie that will hold the access token.

Default: `"access_token_cookie"`

`JWT_ACCESS_COOKIE_PATH`

The path for the access cookies

Default: `"/"`

`JWT_COOKIE_CSRF_PROTECT`

Controls if Cross Site Request Forgery (CSRF) protection is enabled when using cookies.

This should always be True in production

Default: `True`

`JWT_COOKIE_DOMAIN`

Value to use for cross domain cookies. For example, if `JWT_COOKIE_DOMAIN` is `".example.com"`, the cookies will be set so they are readable by the domains www.example.com, foo.example.com etc. Otherwise, a cookie will only be readable by the domain that set it.

Default: `None`

`JWT_COOKIE_SAMESITE`

Controls how the cookies should be sent in a cross-site browsing context. Available options are `"None"`, `"Lax"`, or `"Strict"`.

To use `SameSite=None`, you must set this option to the string `"None"` as well as setting `JWT_COOKIE_SECURE` to `True`.

See the MDN docs for more information.

Default: `None`, which is treated as `"Lax"` by browsers.

`JWT_COOKIE_SECURE`

Controls if the `secure` flag should be placed on cookies created by this extension. If a cookie is marked as `secure` it will only be sent by the web browser over an HTTPS connection.

This should always be True in production.

Default: `False`

`JWT_REFRESH_COOKIE_NAME`

The name of the cookie that will hold the refresh token.

Note: We generally do not recommend using refresh tokens with cookies. See *Implicit Refreshing With Cookies*.

Default: `"refresh_token_cookie"`

`JWT_REFRESH_COOKIE_PATH`

The path for the refresh cookies

Note: We generally do not recommend using refresh tokens with cookies. See *Implicit Refreshing With Cookies*.

Default: `"/"`

`JWT_SESSION_COOKIE`

Controls if the cookies will be set as session cookies, which are deleted when the browser is closed.

Default: `True`

9.5 Cross Site Request Forgery Options:

These are only applicable if a route is configured to accept JWTs via cookies and `JWT_COOKIE_CSRF_PROTECT` is `True`.

`JWT_ACCESS_CSRF_COOKIE_NAME`

The name of the cookie that contains the CSRF double submit token. Only applicable if `JWT_CSRF_IN_COOKIES` is `True`

Default: `csrf_access_token`

`JWT_ACCESS_CSRF_COOKIE_PATH`

The path of the access CSRF double submit cookie.

Default: `"/"`

`JWT_ACCESS_CSRF_FIELD_NAME`

Name of the form field that should contain the CSRF double submit token for an access token. Only applicable if `JWT_CSRF_CHECK_FORM` is `True`

Default: `"csrf_token"`

`JWT_ACCESS_CSRF_HEADER_NAME`

The name of the header on an incoming request that should contain the CSRF double submit token.

Default: `"X-CSRF-TOKEN"`

`JWT_CSRF_CHECK_FORM`

Controls if form data should also be check for the CSRF double submit token.

Default: `False`

`JWT_CSRF_IN_COOKIES`

Controls if the CSRF double submit token will be stored in additional cookies. If setting this to `False`, you can use *`flask_jwt_extended.get_csrf_token()`* to get the csrf token from an encoded JWT, and return it to your frontend in whatever way suites your application.

Default: `True`

`JWT_CSRF_METHODS`

A list of HTTP methods that we should do CSRF checks on.

Default: `["POST", "PUT", "PATCH", "DELETE"]`

`JWT_REFRESH_CSRF_COOKIE_NAME`

The name of the cookie that contains the CSRF double submit token. Only applicable if `JWT_CSRF_IN_COOKIES` is `True`

Note: We generally do not recommend using refresh tokens with cookies. See *Implicit Refreshing With Cookies*.

Default: `csrf_refresh_token`

JWT_REFRESH_CSRF_COOKIE_PATH

The path of the refresh CSRF double submit cookie.

Note: We generally do not recommend using refresh tokens with cookies. See *Implicit Refreshing With Cookies*.

Default: `"/"`

JWT_REFRESH_CSRF_FIELD_NAME

Name of the form field that should contain the CSRF double submit token for a refresh token. Only applicable if `JWT_CSRF_CHECK_FORM` is `True`

Note: We generally do not recommend using refresh tokens with cookies. See *Implicit Refreshing With Cookies*.

Default: `"csrf_token"`

JWT_REFRESH_CSRF_HEADER_NAME

The name of the header on an incoming request that should contain the CSRF double submit token.

Note: We generally do not recommend using refresh tokens with cookies. See *Implicit Refreshing With Cookies*.

Default: `"X-CSRF-TOKEN"`

9.6 Query String Options:

These are only applicable if a route is configured to accept JWTs via query string.

JWT_QUERY_STRING_NAME

What query string parameter should contain the JWT.

Default: `"jwt"`

JWT_QUERY_STRING_VALUE_PREFIX

An optional prefix string that should show up before the JWT in a query string parameter.

For example, if this was `"Bearer "`, the query string should look like `"/endpoint?jwt=Bearer <JWT>"`

Default: `""`

9.7 JSON Body Options:

These are only applicable if a route is configured to accept JWTs via the JSON body.

JWT_JSON_KEY

What key should contain the access token in the JSON body of a request.

Default: `"access_token"`

JWT_REFRESH_JSON_KEY

What key should contain the refresh token in the JSON body of a request.

Default: `"access_token"`

CHAPTER TEN

CHANGING DEFAULT BEHAVIORS

This extension provides sensible default behaviors. For example, if an expired token attempts to access a protected endpoint, you will get a JSON response back like `{"msg": "Token has expired"}` and a 401 status code. However there may be various behaviors of this extension that you want to customize to your application's needs. We can do that with the various loader functions. Here is an example of how to do that.

```python
from flask import Flask
from flask import jsonify

from flask_jwt_extended import create_access_token
from flask_jwt_extended import jwt_required
from flask_jwt_extended import JWTManager

app = Flask(__name__)

app.config["JWT_SECRET_KEY"] = "super-secret"  # Change this!
jwt = JWTManager(app)


# Set a callback function to return a custom response whenever an expired
# token attempts to access a protected route. This particular callback function
# takes the jwt_header and jwt_payload as arguments, and must return a Flask
# response. Check the API documentation to see the required argument and return
# values for other callback functions.
@jwt.expired_token_loader
def my_expired_token_callback(jwt_header, jwt_payload):
    return jsonify(code="dave", err="I can't let you do that"), 401


@app.route("/login", methods=["POST"])
def login():
    access_token = create_access_token("example_user")
    return jsonify(access_token=access_token)


@app.route("/protected", methods=["GET"])
@jwt_required()
def protected():
    return jsonify(hello="world")


```

(continues on next page)

(continued from previous page)

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

There are all sorts of callbacks that can be defined to customize the behaviors of this extension. See the *Configuring Flask-JWT-Extended* API Documentation for a full list of callback functions that are available in this extension.

CHAPTER

ELEVEN

CUSTOM DECORATORS

You can create your own decorators that extend the functionality of the decorators provided by this extension. For example, you may want to create your own decorator that verifies a JWT is present as well as verifying that the current user is an administrator.

flask_jwt_extended.verify_jwt_in_request() can be used to build your own decorators. This is the same function used by *flask_jwt_extended.jwt_required()*.

Here is an example of how this might look.

```python
from functools import wraps

from flask import Flask
from flask import jsonify

from flask_jwt_extended import create_access_token
from flask_jwt_extended import get_jwt
from flask_jwt_extended import JWTManager
from flask_jwt_extended import verify_jwt_in_request

app = Flask(__name__)

app.config["JWT_SECRET_KEY"] = "super-secret"  # Change this!
jwt = JWTManager(app)


# Here is a custom decorator that verifies the JWT is present in the request,
# as well as insuring that the JWT has a claim indicating that this user is
# an administrator
def admin_required():
    def wrapper(fn):
        @wraps(fn)
        def decorator(*args, **kwargs):
            verify_jwt_in_request()
            claims = get_jwt()
            if claims["is_administrator"]:
                return fn(*args, **kwargs)
            else:
                return jsonify(msg="Admins only!"), 403

        return decorator
```

(continues on next page)

(continued from previous page)

```
    return wrapper


@app.route("/login", methods=["POST"])
def login():
    access_token = create_access_token(
        "admin_user", additional_claims={"is_administrator": True}
    )
    return jsonify(access_token=access_token)


@app.route("/protected", methods=["GET"])
@admin_required()
def protected():
    return jsonify(foo="bar")


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

CHAPTER

TWELVE

API DOCUMENTATION

This is the documentation for all of the API that is exported in this extension.

12.1 Configuring Flask-JWT-Extended

class `flask_jwt_extended.`**`JWTManager`**(*app: Flask | None = None*, *add_context_processor: bool = False*)

An object used to hold JWT settings and callback functions for the Flask-JWT-Extended extension.

Instances of *`JWTManager`* are *not* bound to specific apps, so you can create one in the main body of your code and then bind it to your app in a factory function.

`additional_claims_loader`(*callback: Callable*) → Callable

This decorator sets the callback function used to add additional claims when creating a JWT. The claims returned by this function will be merged with any claims passed in via the `additional_claims` argument to *`create_access_token()`* or *`create_refresh_token()`*.

The decorated function must take **one** argument.

The argument is the identity that was used when creating a JWT.

The decorated function must return a dictionary of claims to add to the JWT.

`additional_headers_loader`(*callback: Callable*) → Callable

This decorator sets the callback function used to add additional headers when creating a JWT. The headers returned by this function will be merged with any headers passed in via the `additional_headers` argument to *`create_access_token()`* or *`create_refresh_token()`*.

The decorated function must take **one** argument.

The argument is the identity that was used when creating a JWT.

The decorated function must return a dictionary of headers to add to the JWT.

`decode_key_loader`(*callback: Callable*) → Callable

This decorator sets the callback function for dynamically setting the JWT decode key based on the **UNVERIFIED** contents of the token. Think carefully before using this functionality, in most cases you probably don't need it.

The decorated function must take **two** arguments.

The first argument is a dictionary containing the header data of the unverified JWT.

The second argument is a dictionary containing the payload data of the unverified JWT.

The decorated function must return a *string* that is used to decode and verify the token.

`encode_key_loader`(*callback: Callable*) → Callable

This decorator sets the callback function for dynamically setting the JWT encode key based on the tokens identity. Think carefully before using this functionality, in most cases you probably don't need it.

The decorated function must take **one** argument.

The argument is the identity used to create this JWT.

The decorated function must return a *string* which is the secrete key used to encode the JWT.

`expired_token_loader`(*callback: Callable*) → Callable

This decorator sets the callback function for returning a custom response when an expired JWT is encountered.

The decorated function must take **two** arguments.

The first argument is a dictionary containing the header data of the JWT.

The second argument is a dictionary containing the payload data of the JWT.

The decorated function must return a Flask Response.

`init_app`(*app: Flask*, *add_context_processor: bool = False*) → None

Register this extension with the flask app.

Parameters

- `app` – The Flask Application object
- `add_context_processor` – Controls if *current_user* is should be added to flasks template context (and thus be available for use in Jinja templates). Defaults to `False`.

`invalid_token_loader`(*callback: Callable*) → Callable

This decorator sets the callback function for returning a custom response when an invalid JWT is encountered.

This decorator sets the callback function that will be used if an invalid JWT attempts to access a protected endpoint.

The decorated function must take **one** argument.

The argument is a string which contains the reason why a token is invalid.

The decorated function must return a Flask Response.

`needs_fresh_token_loader`(*callback: Callable*) → Callable

This decorator sets the callback function for returning a custom response when a valid and non-fresh token is used on an endpoint that is marked as `fresh=True`.

The decorated function must take **two** arguments.

The first argument is a dictionary containing the header data of the JWT.

The second argument is a dictionary containing the payload data of the JWT.

The decorated function must return a Flask Response.

`revoked_token_loader`(*callback: Callable*) → Callable

This decorator sets the callback function for returning a custom response when a revoked token is encountered.

The decorated function must take **two** arguments.

The first argument is a dictionary containing the header data of the JWT.

The second argument is a dictionary containing the payload data of the JWT.

The decorated function must return a Flask Response.

`token_in_blocklist_loader`(*callback: Callable*) → Callable

This decorator sets the callback function used to check if a JWT has been revoked.

The decorated function must take **two** arguments.

The first argument is a dictionary containing the header data of the JWT.

The second argument is a dictionary containing the payload data of the JWT.

The decorated function must be return `True` if the token has been revoked, `False` otherwise.

`token_verification_failed_loader`(*callback: Callable*) → Callable

This decorator sets the callback function used to return a custom response when the claims verification check fails.

The decorated function must take **two** arguments.

The first argument is a dictionary containing the header data of the JWT.

The second argument is a dictionary containing the payload data of the JWT.

The decorated function must return a Flask Response.

`token_verification_loader`(*callback: Callable*) → Callable

This decorator sets the callback function used for custom verification of a valid JWT.

The decorated function must take **two** arguments.

The first argument is a dictionary containing the header data of the JWT.

The second argument is a dictionary containing the payload data of the JWT.

The decorated function must return `True` if the token is valid, or `False` otherwise.

`unauthorized_loader`(*callback: Callable*) → Callable

This decorator sets the callback function used to return a custom response when no JWT is present.

The decorated function must take **one** argument.

The argument is a string that explains why the JWT could not be found.

The decorated function must return a Flask Response.

`user_identity_loader`(*callback: Callable*) → Callable

This decorator sets the callback function used to convert an identity to a JSON serializable format when creating JWTs. This is useful for using objects (such as SQLAlchemy instances) as the identity when creating your tokens.

The decorated function must take **one** argument.

The argument is the identity that was used when creating a JWT.

The decorated function must return JSON serializable data.

`user_lookup_error_loader`(*callback: Callable*) → Callable

This decorator sets the callback function used to return a custom response when loading a user via *`user_lookup_loader()`* fails.

The decorated function must take **two** arguments.

The first argument is a dictionary containing the header data of the JWT.

The second argument is a dictionary containing the payload data of the JWT.

The decorated function must return a Flask Response.

user_lookup_loader(*callback: Callable*) → Callable

This decorator sets the callback function used to convert a JWT into a python object that can be used in a protected endpoint. This is useful for automatically loading a SQLAlchemy instance based on the contents of the JWT.

The object returned from this function can be accessed via `current_user` or `get_current_user()`

The decorated function must take **two** arguments.

The first argument is a dictionary containing the header data of the JWT.

The second argument is a dictionary containing the payload data of the JWT.

The decorated function can return any python object, which can then be accessed in a protected endpoint. If an object cannot be loaded, for example if a user has been deleted from your database, `None` must be returned to indicate that an error occurred loading the user.

12.2 Verify Tokens in Request

flask_jwt_extended.**jwt_required**(*optional: bool = False*, *fresh: bool = False*, *refresh: bool = False*, *locations: str | Sequence | None = None*, *verify_type: bool = True*, *skip_revocation_check: bool = False*) → Any

A decorator to protect a Flask endpoint with JSON Web Tokens.

Any route decorated with this will require a valid JWT to be present in the request (unless optional=True, in which case no JWT is also valid) before the endpoint can be called.

Parameters

- **optional** – If `True`, allow the decorated endpoint to be accessed if no JWT is present in the request. Defaults to `False`.
- **fresh** – If `True`, require a JWT marked with `fresh` to be able to access this endpoint. Defaults to `False`.
- **refresh** – If `True`, requires a refresh JWT to access this endpoint. If `False`, requires an access JWT to access this endpoint. Defaults to `False`.
- **locations** – A location or list of locations to look for the JWT in this request, for example `'headers'` or `['headers', 'cookies']`. Defaults to `None` which indicates that JWTs will be looked for in the locations defined by the `JWT_TOKEN_LOCATION` configuration option.
- **verify_type** – If `True`, the token type (access or refresh) will be checked according to the `refresh` argument. If `False`, type will not be checked and both access and refresh tokens will be accepted.
- **skip_revocation_check** – If `True`, revocation status of the token will be *not* checked. If `False`, revocation status of the token will be checked.

flask_jwt_extended.**verify_jwt_in_request**(*optional: bool = False*, *fresh: bool = False*, *refresh: bool = False*, *locations: str | Sequence | None = None*, *verify_type: bool = True*, *skip_revocation_check: bool = False*) → Tuple[dict, dict] | None

Verify that a valid JWT is present in the request, unless `optional=True` in which case no JWT is also considered valid.

Parameters

- **optional** – If True, do not raise an error if no JWT is present in the request. Defaults to False.
- **fresh** – If True, require a JWT marked as fresh in order to be verified. Defaults to False.
- **refresh** – If True, requires a refresh JWT to access this endpoint. If False, requires an access JWT to access this endpoint. Defaults to False
- **locations** – A location or list of locations to look for the JWT in this request, for example 'headers' or ['headers', 'cookies']. Defaults to None which indicates that JWTs will be looked for in the locations defined by the JWT_TOKEN_LOCATION configuration option.
- **verify_type** – If True, the token type (access or refresh) will be checked according to the refresh argument. If False, type will not be checked and both access and refresh tokens will be accepted.
- **skip_revocation_check** – If True, revocation status of the token will be *not* checked. If False, revocation status of the token will be checked.

Returns

A tuple containing the jwt_header and the jwt_data if a valid JWT is present in the request. If optional=True and no JWT is in the request, None will be returned instead. Raise an exception if an invalid JWT is in the request.

12.3 Utilities

flask_jwt_extended.**create_access_token**(*identity: Any*, *fresh: bool | float | timedelta = False*, *expires_delta: Literal[False] | timedelta | None = None*, *additional_claims=None*, *additional_headers=None*)

Create a new access token.

Parameters

- **identity** – The identity of this token. It can be any data that is json serializable. You can use *user_identity_loader()* to define a callback function to convert any object passed in into a json serializable format.
- **fresh** – If this token should be marked as fresh, and can thus access endpoints protected with @jwt_required(fresh=True). Defaults to False.

  This value can also be a datetime.timedelta, which indicate how long this token will be considered fresh.
- **expires_delta** – A datetime.timedelta for how long this token should last before it expires. Set to False to disable expiration. If this is None, it will use the JWT_ACCESS_TOKEN_EXPIRES config value (see *Configuration Options*)
- **additional_claims** – Optional. A hash of claims to include in the access token. These claims are merged into the default claims (exp, iat, etc) and claims returned from the *additional_claims_loader()* callback. On conflict, these claims take precedence.
- **headers** – Optional. A hash of headers to include in the access token. These headers are merged into the default headers (alg, typ) and headers returned from the *additional_headers_loader()* callback. On conflict, these headers take precedence.

Returns

An encoded access token

`flask_jwt_extended.`**`create_refresh_token`**`(`*identity: Any*, *expires_delta: Literal[False] | timedelta | None = None*, *additional_claims=None*, *additional_headers=None*`)`

Create a new refresh token.

Parameters

- **`identity`** – The identity of this token. It can be any data that is json serializable. You can use *`user_identity_loader()`* to define a callback function to convert any object passed in into a json serializable format.
- **`expires_delta`** – A `datetime.timedelta` for how long this token should last before it expires. Set to False to disable expiration. If this is None, it will use the `JWT_REFRESH_TOKEN_EXPIRES` config value (see *Configuration Options*)
- **`additional_claims`** – Optional. A hash of claims to include in the refresh token. These claims are merged into the default claims (exp, iat, etc) and claims returned from the *`additional_claims_loader()`* callback. On conflict, these claims take precedence.
- **`headers`** – Optional. A hash of headers to include in the refresh token. These headers are merged into the default headers (alg, typ) and headers returned from the *`additional_headers_loader()`* callback. On conflict, these headers take precedence.

Returns

An encoded refresh token

`flask_jwt_extended.`**`current_user`**

A LocalProxy for accessing the current user. Roughly equilivant to *`get_current_user()`*

`flask_jwt_extended.`**`decode_token`**`(`*encoded_token: str*, *csrf_value: str | None = None*, *allow_expired: bool = False*`)` → dict

Returns the decoded token (python dict) from an encoded JWT. This does all the checks to ensure that the decoded token is valid before returning it.

This will not fire the user loader callbacks, save the token for access in protected endpoints, checked if a token is revoked, etc. This is puerly used to ensure that a JWT is valid.

Parameters

- **`encoded_token`** – The encoded JWT to decode.
- **`csrf_value`** – Expected CSRF double submit value (optional).
- **`allow_expired`** – If `True`, do not raise an error if the JWT is expired. Defaults to `False`

Returns

Dictionary containing the payload of the JWT decoded JWT.

`flask_jwt_extended.`**`get_csrf_token`**`(`*encoded_token: str*`)` → str

Returns the CSRF double submit token from an encoded JWT.

Parameters

`encoded_token` – The encoded JWT

Returns

The CSRF double submit token (string)

`flask_jwt_extended.`**`get_current_user`**`()` → Any

In a protected endpoint, this will return the user object for the JWT that is accessing the endpoint.

This is only usable if *`user_lookup_loader()`* is configured. If the user loader callback is not being used, this will raise an error.

If no JWT is present due to `jwt_required(optional=True)`, `None` is returned.

Returns
The current user object for the JWT in the current request

`flask_jwt_extended.`**`get_jti`**`(`*encoded_token: str*`)` → str | None

Returns the JTI (unique identifier) of an encoded JWT

Parameters
`encoded_token` – The encoded JWT to get the JTI from.

Returns
The JTI (unique identifier) of a JWT, if it is present.

`flask_jwt_extended.`**`get_jwt`**`()` → dict

In a protected endpoint, this will return the python dictionary which has the payload of the JWT that is accessing the endpoint. If no JWT is present due to `jwt_required(optional=True)`, an empty dictionary is returned.

Returns
The payload (claims) of the JWT in the current request

`flask_jwt_extended.`**`get_jwt_header`**`()` → dict

In a protected endpoint, this will return the python dictionary which has the header of the JWT that is accessing the endpoint. If no JWT is present due to `jwt_required(optional=True)`, an empty dictionary is returned.

Returns
The headers of the JWT in the current request

`flask_jwt_extended.`**`get_jwt_identity`**`()` → Any

In a protected endpoint, this will return the identity of the JWT that is accessing the endpoint. If no JWT is present due to `jwt_required(optional=True)`, `None` is returned.

Returns
The identity of the JWT in the current request

`flask_jwt_extended.`**`get_unverified_jwt_headers`**`(`*encoded_token: str*`)` → dict

Returns the Headers of an encoded JWT without verifying the signature of the JWT.

Parameters
`encoded_token` – The encoded JWT to get the Header from.

Returns
JWT header parameters as python dict()

`flask_jwt_extended.`**`set_access_cookies`**`(`*response: Response*, *encoded_access_token: str*, *max_age=None*, *domain=None*`)` → None

Modifiy a Flask Response to set a cookie containing the access JWT. Also sets the corresponding CSRF cookies if `JWT_CSRF_IN_COOKIES` is `True` (see *Configuration Options*)

Parameters

- **`response`** – A Flask Response object.
- **`encoded_access_token`** – The encoded access token to set in the cookies.
- **`max_age`** – The max age of the cookie. If this is None, it will use the `JWT_SESSION_COOKIE` option (see *Configuration Options*). Otherwise, it will use this as the cookies `max-age` and the JWT_SESSION_COOKIE option will be ignored. Values should be the number of seconds (as an integer).
- **`domain`** – The domain of the cookie. If this is None, it will use the `JWT_COOKIE_DOMAIN` option (see *Configuration Options*). Otherwise, it will use this as the cookies `domain` and the JWT_COOKIE_DOMAIN option will be ignored.

`flask_jwt_extended.`**`set_refresh_cookies`**(*response: Response*, *encoded_refresh_token: str*, *max_age: int | None = None*, *domain: str | None = None*) → None

Modifiy a Flask Response to set a cookie containing the refresh JWT. Also sets the corresponding CSRF cookies if `JWT_CSRF_IN_COOKIES` is `True` (see *Configuration Options*)

Parameters

- **`response`** – A Flask Response object.
- **`encoded_refresh_token`** – The encoded refresh token to set in the cookies.
- **`max_age`** – The max age of the cookie. If this is None, it will use the `JWT_SESSION_COOKIE` option (see *Configuration Options*). Otherwise, it will use this as the cookies `max-age` and the JWT_SESSION_COOKIE option will be ignored. Values should be the number of seconds (as an integer).
- **`domain`** – The domain of the cookie. If this is None, it will use the `JWT_COOKIE_DOMAIN` option (see *Configuration Options*). Otherwise, it will use this as the cookies `domain` and the JWT_COOKIE_DOMAIN option will be ignored.

`flask_jwt_extended.`**`unset_access_cookies`**(*response: Response*, *domain: str | None = None*) → None

Modifiy a Flask Response to delete the cookie containing an access JWT. Also deletes the corresponding CSRF cookie if applicable.

Parameters

- **`response`** – A Flask Response object
- **`domain`** – The domain of the cookie. If this is None, it will use the `JWT_COOKIE_DOMAIN` option (see *Configuration Options*). Otherwise, it will use this as the cookies `domain` and the JWT_COOKIE_DOMAIN option will be ignored.

`flask_jwt_extended.`**`unset_jwt_cookies`**(*response: Response*, *domain: str | None = None*) → None

Modifiy a Flask Response to delete the cookies containing access or refresh JWTs. Also deletes the corresponding CSRF cookies if applicable.

Parameters

`response` – A Flask Response object

`flask_jwt_extended.`**`unset_refresh_cookies`**(*response: Response*, *domain: str | None = None*) → None

Modifiy a Flask Response to delete the cookie containing a refresh JWT. Also deletes the corresponding CSRF cookie if applicable.

Parameters

- **`response`** – A Flask Response object
- **`domain`** – The domain of the cookie. If this is None, it will use the `JWT_COOKIE_DOMAIN` option (see *Configuration Options*). Otherwise, it will use this as the cookies `domain` and the JWT_COOKIE_DOMAIN option will be ignored.

CHAPTER

THIRTEEN

4.0.0 BREAKING CHANGES AND UPGRADE GUIDE

This release includes a lot of breaking changes that have been a long time coming, and will require some manual intervention to upgrade your application. Breaking changes are never fun, but I really believe they are for the best. As a result of all these changes, this extension should be simpler to use, provide more flexibility, and allow for easier additions to the API without introducing further breaking changes. Here is everything you will need to be aware of when upgrading to 4.0.0.

13.1 Encoded JWT Changes (IMPORTANT)

- The `JWT_USER_CLAIMS` configuration option has been removed. Now when creating JWTs with additional claims, those claims are put on the top level of the token, instead of inside the the nested `user_claims` dictionary. This has the very important benefit of allowing you to override reserved claims (such as `nbf`) which was not previously possible in this extension.

  IMPORTANT NOTE:

  This has the unfortunate side effect that any existing JWTs your application is using will not work correctly if they utilize additional claims. We **strongly** suggest changing your secret key to force all users to get the new format of JWTs. If that is not feasible for your application you could build a shim to handle both the old JWTs which store additional claims in the `user_claims` key, and the new format where additional claims are now stored at the top level, until all the JWTs have had a chance to cycle to the new format.

- The default `JWT_IDENTITY_CLAIM` option is now `sub` instead of `identity`.

13.2 General Changes

- Dropped support for everything before Python 3.6 (including Python 2).
- Requires PyJWT >= 2.0.0.
- Depreciation warnings in `3.25.2` have been removed and are now errors:
  - The `JWT_CSRF_HEADER_NAME` option has removed.
  - The `jwt.expired_token_loader` will error if the callback does not take an argument for the expired token header and expired token payload.
  - The `jwt.decode_key_loader` will error if the callback does not take an argument for the unverified_headers and the unverified_payload.
- Calling `get_jwt()`, `get_jwt_header()`, or `get_jwt_identity()` will raise a `RuntimeError` when called outside of a protected context (ie if you forgot `@jwt_required()` or `verify_jwt_in_request()`). Previously these calls would return `None`.

- Calling `get_jwt()` or `get_jwt_header()` will return an empty dictionary if called from an optionally protected endpoint. Previously this would return `None`.
- Calling `get_current_user()` or `current_user` will raise a `RuntimeError` if no `@jwt.user_lookup_loader` callback is defined.

13.3 Blacklist Changes

- All occurrences of `blacklist` have been renamed to `blocklist`
- The `JWT_BLACKLIST_ENABLED` option has been removed. If you do not want to check a JWT against your blocklist, do not register a callback function with `@jwt.token_in_blocklist_loader`.
- The `JWT_BLACKLIST_TOKEN_CHECKS` option has been removed. If you don't want to check a given token type against the blocklist, specifically ignore it in your callback function by checking the `jwt_payload["type"]` and short circuiting accordingly. `jwt_payload["type"]` will be either `"access"` or `"refresh"`.

13.4 Callback Function Changes

- Renamed `@jwt.claims_verification_loader` to `@jwt.token_verification_loader`
- Renamed `@jwt.claims_verification_failed_loader` to `@jwt.token_verification_failed_loader`
- Renamed `@jwt.user_claims_loader` to `@jwt.additional_claims_loader`
- Renamed `@jwt.user_in_blacklist_loader` to `@jwt.user_in_blocklist_loader`
- Renamed `@jwt.user_loader_callback_loader` to `@jwt.user_lookup_loader`
- Renamed `@jwt.user_loader_error_loader` to `@jwt.user_lookup_error_loader`
- The following callback functions have all been changed to take two arguments. Those arguments are the `jwt_headers` and `jwt_payload`.
  - `@jwt.needs_fresh_token_loader`
  - `@jwt.revoked_token_loader`
  - `@jwt.user_lookup_loader`
  - `@jwt.user_lookup_error_loader`
  - `@jwt.expired_token_loader`
  - `@jwt.token_in_blocklist_loader`
  - `@jwt.token_verification_loader`
  - `@jwt.token_verification_failed_loader`

```python
@jwt.revoked_token_loader
def revoked_token_response(jwt_header, jwt_payload):
    return jsonify(msg=f"I'm sorry {jwt_payload['sub']} I can't let you do that")
```

- The arguments for `@jwt.decode_key_loader` have been reversed to be consistent with the rest of the application. Previously the arguments were `(jwt_payload, jwt_headers)`. Now they are `(jwt_headers, jwt_payload)`.

13.5 API Changes

- **All view decorators have been moved to a single decorator:**
  - `@jwt_required` is now `@jwt_required()`
  - `@jwt_optional` is now `@jwt_required(optional=True)`
  - `@fresh_jwt_required` is now `@jwt_required(fresh=True)`
  - `@jwt_refresh_token_required` is now `@jwt_required(refresh=True)`
- **All additional `verify_jwt_in_request` functions have been moved to a single method:**
  - `verify_jwt_in_request_optional()` is now `verify_jwt_in_request(optional=True)`
  - `verify_jwt_refresh_token_in_request()` is now `verify_jwt_in_request(refresh=True)`
  - `verify_fresh_jwt_in_request()` is now `verify_jwt_in_request(fresh=True)`
- Renamed `get_raw_jwt()` to `get_jwt()`
- Renamed `get_raw_jwt_headers()` to `get_jwt_headers()`
- Removed `get_jwt_claims()`. Use `get_jwt()` instead.
- The `headers` argument in `create_access_token()` and `create_refresh_token()` has been renamed to `additional_headers`.
  - If you pass in the `additional_headers`, it will now be merged with the headers returned by the `@jwt.additional_headers_loader` callback, with ties going to the `additional_headers` argument.
- The `user_claims` argument in `create_access_token()` and `create_refresh_token()` has been renamed to `additional_claims`.
  - If you pass in the `additional_claims` option, it will now be merged with the claims returned by the `@jwt.additional_claims_loader` callback, with ties going to the `additional_claims` argument.
- The `JWT_VERIFY_AUDIENCE` option has been removed. If you do not want to verify the JWT audience (`aud`) claim, simply do not set the `JWT_DECODE_AUDIENCE` option.
- The `JWT_CLAIMS_IN_REFRESH_TOKEN` option has been removed. Additional claims will now always be put in the JWT regardless of if it is an access or refresh tokens. If you don't want additional claims in your refresh tokens, do not include any additional claims when creating the refresh token.
- Removed `UserLoadError` from `flask_jwt_extended.exceptions`. Use `UserLookupError` instead.

13.6 New Stuff

- Add `locations` argument to `@jwt_required()` and `verify_jwt_in_request`. This will allow you to override the `JWT_LOCATIONS` option on a per route basis.
- Revamped and cleaned up documentation. It should be clearer how to work with this extension both on the backend and frontend now.
- Lots of code cleanup behind the scenes.

PYTHON MODULE INDEX

INDEX