# sanic-jwt Documentation

***Release 1.2.1***

**Adam Hopkins**

**Dec 04, 2018**

# Contents:

Sanic JWT adds authentication protection and endpoints to Sanic.

It is both **easy** to get up and running, and **extensible** for the developer. It can act to *protect endpoints* and also *provide authentication scoping*, all wrapped into a nice JWT.

Pick your favorite user management system, run *a single class to initialize*, and you are all set.

Open source code on GitHub

---

# CHAPTER 1

## Getting Started

In order to add **Sanic JWT**, all you need to do is initialize an instance of `sanic_jwt.Initialize` with your `Sanic()` instance, and an *authentication function*.

```
from sanic_jwt import Initialize

async def authenticate(request):
    return dict(user_id='some_id')

app = Sanic()
Initialize(app, authenticate=authenticate)
```

What is an authentication function? We'll get into it later, but for now all you need to know is that it is a function **you** control that takes a `request` and decides if there is a valid user or not. This gives you the flexibility to roll with whatever user management system you want.

After initialization, you now will have a couple endpoints at your disposal:

- `/auth`
- `/auth/verify`

To obtain a token, just send a **POST** call to the `/auth` endpoint:

```
curl -X POST http://localhost:8000/auth
```

You should get back a bit of JSON like this:

```
{
    "access_token": "<JWT>"
}
```

Want to check to make sure it is valid?

```
curl -X GET -H "Authorization: Bearer <JWT>" http://localhost:8000/auth/verify
```

Your response should be this:

```
{
    "valid": true
}
```

You now have a working authentication system. Woohoo!

CHAPTER 2

# Installation

Install from pypi using:

```
pip install sanic-jwt
```

Not much else to say. Depending upon whether or not you change the encryption algorithm, you may be prompted to also install cryptography. But, we only do that if you know you will need it.

```
pip install cryptography
```

# CHAPTER 3

## Simple Usage

Let's take a look at a real simple example on how to use Sanic JWT to see the core concepts. Suppose we have a very simple user management system that stores `User` objects in a `list`. You can also access a user through a `dict` index on the `user_id` and the `username`.

```
class User:

    def __init__(self, id, username, password):
        self.user_id = id
        self.username = username
        self.password = password

    def __repr__(self):
        return "User(id='{}')".format(self.user_id)

    def to_dict(self):
        return {"user_id": self.user_id, "username": self.username}


users = [User(1, "user1", "abcxyz"), User(2, "user2", "abcxyz")]
```

We want to be able to pass in a **username** and a **password** to authenticate our user, and then receive back an **access token** that can be used later on to access protected (aka private) data.

To get **Sanic JWT** started, we know that we need to *initialize* with the `authenticate` method. The job of this method is to take the `request` and determine if there is a valid user to be authenticated. Since the developer decides upon the user management system, it is our job to figure out what this method should do.

Very simple. Since we want to pass a **username** and a **password** to authenticate our user, we just need to check that the credentials are correct. If yes, we return the user. If no, we raise an *exception*.

```
from sanic_jwt import exceptions

async def authenticate(request, *args, **kwargs):
    username = request.json.get("username", None)
```

(continues on next page)

(continued from previous page)

```
    password = request.json.get("password", None)

    if not username or not password:
        raise exceptions.AuthenticationFailed("Missing username or password.")

    user = username_table.get(username, None)
    if user is None:
        raise exceptions.AuthenticationFailed("User not found.")

    if password != user.password:
        raise exceptions.AuthenticationFailed("Password is incorrect.")

    return user
```

**Warning:** In a real production setting it is advised to **not** tell the user why their authentication failed. Simply raising `exceptions.AuthenticationFailed` should be enough. Here, for example purposes, we added some helper messages just to make it clear where we are failing.

Our whole application now looks like this:

```
from sanic import Sanic
from sanic_jwt import exceptions
from sanic_jwt import initialize


class User:

    def __init__(self, id, username, password):
        self.user_id = id
        self.username = username
        self.password = password

    def __repr__(self):
        return "User(id='{}')".format(self.user_id)

    def to_dict(self):
        return {"user_id": self.user_id, "username": self.username}


users = [User(1, "user1", "abcxyz"), User(2, "user2", "abcxyz")]

username_table = {u.username: u for u in users}
userid_table = {u.user_id: u for u in users}


async def authenticate(request, *args, **kwargs):
    username = request.json.get("username", None)
    password = request.json.get("password", None)

    if not username or not password:
        raise exceptions.AuthenticationFailed("Missing username or password.")

    user = username_table.get(username, None)
    if user is None:
```

(continues on next page)

(continued from previous page)

```
        raise exceptions.AuthenticationFailed("User not found.")

    if password != user.password:
        raise exceptions.AuthenticationFailed("Password is incorrect.")

    return user


app = Sanic()
initialize(app, authenticate=authenticate)


if __name__ == "__main__":
    app.run(host="127.0.0.1", port=8888)
```

Let's try and get an access token now:

```
curl -iv -H "Content-Type: application/json" -d '{"username": "user1", "password":
↪"wrongpassword"}' http://localhost:8888/auth
```

Here is our response:

```
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 8888 (#0)
> POST /auth HTTP/1.1
> Host: localhost:8888
> User-Agent: curl/7.55.1
> Accept: */*
> Content-Type: application/json
> Content-Length: 50
>
* upload completely sent off: 50 out of 50 bytes
< HTTP/1.1 401 Unauthorized
HTTP/1.1 401 Unauthorized
< Connection: keep-alive
Connection: keep-alive
< Keep-Alive: 60
Keep-Alive: 60
< Content-Length: 22
Content-Length: 22
< Content-Type: text/plain; charset=utf-8
Content-Type: text/plain; charset=utf-8

<
* Connection #0 to host localhost left intact
{"reasons":"Password is incorrect.","exception":"AuthenticationFailed"}
```

Oops! Looks like we entered the wrong password. Let's try again:

```
curl -iv -H "Content-Type: application/json" -d '{"username": "user1", "password":
↪"abcxyz"}' http://localhost:8888/auth
```

Response:

```
*   Trying 127.0.0.1...
* TCP_NODELAY set
```

(continues on next page)

(continued from previous page)

```
* Connected to localhost (127.0.0.1) port 8888 (#0)
> POST /auth HTTP/1.1
> Host: localhost:8888
> User-Agent: curl/7.55.1
> Accept: */*
> Content-Type: application/json
> Content-Length: 43
>
* upload completely sent off: 43 out of 43 bytes
< HTTP/1.1 200 OK
HTTP/1.1 200 OK
< Connection: keep-alive
Connection: keep-alive
< Keep-Alive: 60
Keep-Alive: 60
< Content-Length: 140
Content-Length: 140
< Content-Type: application/json
Content-Type: application/json

<
* Connection #0 to host localhost left intact
{"access_token":"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
↪eyJ1c2VyX2lkIjoxLCJleHAiOjE1MTY2NTExNDB9.vmfQbfx0H8vIR6wILlLqS82bJILdwecfWlFRQuHb3Ck
↪"}
```

That looks better. We can head over to jwt.io and enter the `access_token` to see what the token consists of.

**Header**

```
{
  "typ": "JWT",
  "alg": "HS256"
}
```

**Payload**

```
{
  "user_id": 1,
  "exp": 1516651140
}
```

Now, we can confirm that this token works.

```
curl -iv -H "Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
↪eyJ1c2VyX2lkIjoxLCJleHAiOjE1MTY2NTExNDB9.vmfQbfx0H8vIR6wILlLqS82bJILdwecfWlFRQuHb3Ck
↪" http://localhost:8888/auth/verify
```

Response:

```
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 8888 (#0)
> GET /auth/verify HTTP/1.1
> Host: localhost:8888
> User-Agent: curl/7.55.1
> Accept: */*
```

(continues on next page)

(continued from previous page)

```
> Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
↪eyJ1c2VyX2lkIjoxLCJleHAiOjE1MTY2NTExNDB9.vmfQbfx0H8vIR6wILlLqS82bJILdwecfWlFRQuHb3Ck
>
< HTTP/1.1 200 OK
HTTP/1.1 200 OK
< Connection: keep-alive
Connection: keep-alive
< Keep-Alive: 60
Keep-Alive: 60
< Content-Length: 14
Content-Length: 14
< Content-Type: application/json
Content-Type: application/json

<
* Connection #0 to host localhost left intact
{"valid":true}
```

Excellent. Now that we can generate and verify tokens, we can get to work.

Best of luck creating an authentication scheme that works for you. If you have any questions about how to implement Sanic JWT (or to make it better), please create an issue or get in touch.

CHAPTER 4

# What is a JWT?

JSON Web Tokens ("JWT") are an easy way to pass authentication information to a web-based backend system. They are easy to work with, but admittedly they can be confusing for someone who has never used them. In short, here are some key concepts you should know.

This is meant to give someone a high level overview of JWTs and a practical working knowledge of what is needed to get up and running with them in an application. For more information, I suggest you read jwt.io.

## 4.1 What does it look like?

JWTs are string of a bunch of characters:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.
↪TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

Upon closer inspection, it consists of three parts separated by a `period`:

```
- eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
- eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9
- TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

In order, these parts are the **Header**, the **Payload**, and the **Signature**.

They can be decoded using base64.

```
The Header
{
    "alg": "HS256",
    "typ": "JWT"
}
```

(continues on next page)

(continued from previous page)

```
The Payload
{
    "sub": "1234567890",
    "name": "John Doe",
    "admin": true
}

The Signature
<some mess of stuff>
```

## 4.2 The Parts of a JWT

First, the header contains information about how the token's is encoded, and what it is.

Third, the signature is used to verify that it came from a trusted source. It is encrypted using a secret only known to the application. The secret is **NOT** passed along inside the token, and should **NOT** be shared.

Both of these sections, you will not need to concern yourself with to get started.

The Second section is the **payload**. This bit of JSON contains key/value pairs of information. Each one of these is called (in JWT terminology) a **claim**.

In the above example, there are three claims: `sub`, `name`, and `admin`.

## 4.3 The Payload

As an application developer, this is where you can send information from your server that authenticated a user (with a valid user name and password, for example) to a client application that needs to know what to display for the user. You can provide it with almost any bit of information you want that fits in JSON of course. And, because of the signature, you can be comfortable that the information inside the payload has not been compromised.

> **Warning:** The payload is **readable** to anyone that gets a hold of it. **DO NOT** pass sensitive information in it.

While there are no real restrictions on what claims are inside of a JWT, there are some industry standards. Below is a list that Sanic JWT has integrated for you to easily use.

`exp` - (short for *expires*) This claim is a timestamp that dictates when the access token will no longer be available. Because JWT access tokens cannot be invalidated after they are issued, they are typically given a short life span.

`nbf` - (short for *not before*) This claim is a timestamp that allows the token to be created and issued, but not yet enabled for usage until after a certain time.

`iat` - (short for *issued at*) This claim is a timestamp that provides the creation time of the JWT.

`iss` - (short for *issuer*) This claim is typically a URI or other identifier to say who created and issued the token.

`aud` - (short for *audience*) This claim identifies what service the JWT is intended to be used with. Typically it is a URI or other identifier that says the name of the client server that is supposed to be validating the token.

In addition to these claims, there is another claim that generally is important for Sanic JWT: `user_id`. This is meant to be some unique indentification of the user that requested the token. Other than that, you are free to add whatever information you would like. See *Payloads* for information on how to modify the payload in Sanic JWT.

CHAPTER 5

# Initialization

Sanic JWT operates under the hood by creating a Blueprint, and attaching a few routes to your application. This is accomplished with the `Initialize` class.

```
from sanic_jwt import Initialize
from sanic import Sanic

async def authenticate(request):
    return dict(user_id='some_id')

app = Sanic()
Initialize(app, authenticate=authenticate)
```

## 5.1 Concept

Sanic JWT is a user authentication system that does not require the developer to settle on any single user management system. This part is left up to the developer. Therefore, you (as the developer) are left with the responsibility of telling Sanic JWT how to tie into your user management system.

## 5.2 The `Initialize` class

This is the gateway worker into Sanic JWT. When initialized, it allows you to pass run time configurations to it, and gives you a window into customizing how the module will work for you. There are **five main parts** to it when initializing:

1. the **instance** of your Sanic app or a Sanic blueprint

**REQUIRED**

2. handler methods
   one of which (the `authenticate` handler) is **REQUIRED**
3. any runtime configurations you want to make
4. custom view classes for adding your own authentication endpoints
5. component overrides

### 5.2.1 Instance

Most web applications need authentication. With Sanic JWT, all you do is create your Sanic app, and then tell Sanic JWT.

```
from sanic_jwt import Initialize
from sanic import Sanic

app = Sanic()
Initialize(app, authenticate=lambda: True)
```

You can now go ahead and *protect* any route (whether on a blueprint or not).

```
from sanic_jwt import protected
from sanic.response import json

...

@app.route("/")
@protected()
async def test(request):
    return json({ "protected": True })
```

What if we ONLY want the authentication on some subset of our web application? Say, a Blueprint. Not a problem. Just initialize on the blueprint instance and continue as normal.

```
from sanic_jwt import Initialize
from sanic import Sanic, Blueprint

app = Sanic()
bp = Blueprint('my_blueprint')
Initialize(app, authenticate=lambda: True)
app.blueprint(bp)
```

**Warning:** If you are initializing on a blueprint, be careful of the ordering of `app.blueprint()` and `Initialize`. Putting them in the wrong order will cause the authentication endpoints to not properly attach.

---

**Note:** If you decide to initialize more than one instance of Sanic JWT (on multiple blueprints, for example), than an access token generated by one will be acceptable on **ALL** your instances unless they have different a `secret`. You can learn more about how to set that in *Configuration*.

---

Under the hood, Sanic JWT creates its own `Blueprint` for holding all of the *Endpoints and Responses*. If you decide to use your own blueprint (and by all means, feel free to do so!), just know that Sanic JWT will not create its

own. When this happens, Sanic JWT instead will attach to the blueprint that you passed to it.

This is a very powerful tool that allows you to really gain some granularity in your applications' authentication systems.

```
async def authenticate(request, *args, **kwargs):
    return get_my_user()

app = Sanic()
bp1 = Blueprint('my_blueprint_1')
bp2 = Blueprint('my_blueprint_2')

Initialize(app, authenticate=authenticate)
Initialize(bp1, authenticate=authenticate, access_token_name='mytoken')
Initialize(bp2, authenticate=authenticate, access_token_name='yourtoken')
```

In the above example, I now have three independent instances of Sanic JWT running side by side. Each is isolated to its own environment, and can have its own set of *Configuration*.

### 5.2.2 Handlers

There is a group of methods that Sanic JWT uses to hook into your application code. This is how it is able to live alongside your application and seemlessly plug in.

Each handler can be either a **method** or an **awaitable**. You decide.

```
# This works
async def authenticate(request, *args, **kwargs):
    ...

# And so does this
def authenticate(request, *args, **kwargs):
    ...
```

#### 1. `authenticate` - Required

**Purpose**: Just like Django's `authenticate` method, this is responsible for taking a given `request` and deciding whether or not there is a valid user to be authenticated. If yes, it **MUST** return:

- a `dict` with a `user_id` key, **or**
- an instance with an id and `to_dict` property.

By default, it looks for the id on the `user_id` property of a user instance. However, you can *change that to another property*.

If your user should **not** be authenticated, then you should *raise an exception*, preferably `AuthenticationFailed`. Please do not just `return None`. If you do, you will likely get a `500` error.

**Example**:

```
async def authenticate(request, *args, **kwargs):
    username = request.json.get('username', None)
    password = request.json.get('password', None)

    if not username or not password:
        raise exceptions.AuthenticationFailed("Missing username or password.")
```

(continues on next page)

(continued from previous page)

```
    user = await User.get(username=username)
    if user is None:
        raise exceptions.AuthenticationFailed("User not found.")

    if password != user.password:
        raise exceptions.AuthenticationFailed("Password is incorrect.")

    return user

Initialize(app, authenticate)
```

### 2. `store_refresh_token` *

**Purpose**: It is a handler to persist a refresh token to disk. See refresh tokens for more information. Sanic JWT create the refresh token, but you get to decide how it is stored.

**Example**:

```
async def store_refresh_token(user_id, refresh_token, *args, **kwargs):
    key = 'refresh_token_{user_id}'.format(user_id=user_id)
    await aredis.set(key, refresh_token)

Initialize(
    app,
    authenticate=lambda: True,
    store_refresh_token=store_refresh_token)
```

**Warning:** * This parameter is *not* required. However, if you decide to enable refresh tokens (by setting `refresh_token_enabled=True` in your configurations) then the application will raise a `RefreshTokenNotImplemented` exception if you forget to implement this.

### 3. `retrieve_refresh_token` *

**Purpose**: It is a handler to retrieve refresh token from disk. See refresh tokens for more information. Sanic JWT created the refresh token. You stored it. Now Sanic JWT wants it back, it is your job to retrieve it.

**Example**:

```
async def retrieve_refresh_token(user_id, *args, **kwargs):
    key = 'refresh_token_{user_id}'.format(user_id=user_id)
    return await aredis.get(key)

Initialize(
    app,
    authenticate=lambda: True,
    retrieve_refresh_token=retrieve_refresh_token)
```

**Warning:** * This parameter is *not* required. However, if you decide to enable refresh tokens (by setting `refresh_token_enabled=True` in your configurations) then the application will raise a `RefreshTokenNotImplemented` exception if you forget to implement this.

#### 4. `retrieve_user`

**Purpose**: It is a handler to retrieve a user object from your application. It is used to return the user object in the `/auth/me` endpoint, and also the `@inject_user` decorator *that you will learn about later*.

It should return:

- a `dict`, **or**
- an instance of some object with a `to_dict` method, **or**
- `None`

As we said before, you are deciding on the user management system. Sanic JWT is acting as the gatekeeper. But, inherently tied in are a number of use cases where it would be convenient to get your user object. This is how you do it.

**Example**:

```
class User:
    ...

    def to_dict(self):
        properties = ['user_id', 'username', 'email', 'verified']
        return {prop: getattr(self, prop, None) for prop in properties}

async def retrieve_user(request, payload, *args, **kwargs):
    if payload:
        user_id = payload.get('user_id', None)
        user = await User.get(user_id=user_id)
        return user
    else:
        return None

Initialize(
    app,
    authenticate=lambda: True,
    retrieve_user=retrieve_user)
```

You should now have an endpoint at `/auth/me` that will return a serialized form of your currently authenticated user.

```
{
    "me": {
        "user_id": "4",
        "username": "joe",
        "email": "joe@joemail.com",
        "verified": true
    }
}
```

#### 5. `add_scopes_to_payload` *

**Purpose**: It is a handler to add scopes to an access token. See *Scopes* for more information.

Scoping is a long discussion by itself. In short, it is a highly powerful tool to help with providing permissioning to your application. It is your job to add these scopes (if you want them) to the JWT. Then, you can specifiy which scopes are required on specific endpoints.

For now, all you need to do is return a `list` of one or more `strings`.

**Example**:

```
async def add_scopes_to_payload(user):
    return await user.get_scopes()

Initialize(
    app,
    authenticate=lambda: True,
    add_scopes_to_payload=add_scopes_to_payload)
```

### 6. `override_scope_validator` *

**Purpose**: It is a handler to override the default scope validation. See *Scopes* for more information.

This could be useful if you decide to bake some additional logic into your scopes. At its most simplified level, Sanic JWT looks at scopes and compares `fruit:apples` to `fruit:apples`. What if *sometimes* `fruit:oranges` should be accepted? You have the ability to code that override and make your own decision.

---

**Note:** Above, we said "Each of them can be either a method or an awaitable. You decide." What we forgot to mention was that `override_scope_validator` needs to be a regular `callable` and not an `awaitable`.

No async programming here. Sorry for the confusion.

---

**Example**:

```
def my_scope_override(is_valid,
    required,
    user_scopes,
    require_all_actions,
    *args,
    **kwargs):
    return False

Initialize(
    app,
    authenticate=lambda: True,
    override_scope_validator=my_scope_override)
```

### 7. `destructure_scopes`

**Purpose**: It is a handler that allows you to manipulate and handle the scopes before they are validated.

Sometimes, you may find the need to manipulate the scopes before they are validated against the protected resource. In this case, feel free to make changes:

**Example**:

```
async def my_destructure_scopes(scopes, *args, **kwargs):
    return scopes.replace("|", ":")

@app.route("/protected/nonstandardscopes")
@scoped("foo|bar")
def scoped_sync_route(request):
    return json({"nonstandardscopes": True})

Initialize(
    app,
    authenticate=lambda: True,
    destructure_scopes=my_destructure_scopes)
```

#### 8. `extend_payload`

**Purpose**: It is a handler to allow the developer to modify the payload by adding additional claims to it before it is bundled up and packaged inside a JWT.

One of the most powerful concepts of the JWT is that you are able to pass data (aka claims) inside its payload for use by a client application, and reuse when that JWT is being returned for verification. It is simply a method that takes the existing `payload` and returns it (with your brilliant modifications, of course)

**Example**:

```
async def my_extender(payload, user):
    username = user.to_dict().get("username")
    payload.update({"username": username})
    return payload

Initialize(
    app,
    authenticate=lambda: True,
    extend_payload=my_extender)
```

---

### 5.2.3 Runtime Configuration

There are several ways to *configure the settings* for Sanic JWT. One of the easiest is to simply pass the configurations as keyword objects on Initialize.

```
Initialize(
    app,
    access_token_name='mytoken',
    cookie_access_token_name='mytoken',
    cookie_set=True,
    user_id='id',
    claim_iat=True,
    cookie_domain='example.com',)
```

---

### 5.2.4 Additional Views

Sometimes you may need to add some endpoints to the authentication system. When this need arises, create a class based view, and map it as a tuple with the path and handler.

As an example, perhaps you would like to create a "passwordless" login. You could create a form that sends a POST with a user's email address to a `MagicLoginHandler`. That handler sends out an email with a link to your `/auth` endpoint that makes sure the link came from the email.

```
from sanic_jwt import BaseEndpoint

class MagicLoginHandler(BaseEndpoint):
    async def options(self, request):
        return response.text('', status=204)

    async def post(self, request):
        helper = MyCustomUserAuthHelper(app, request)
        token = helper.get_make_me_a_magic_token()
        helper.send_magic_token_to_user_email()

        # Persist the token
        key = f'magic-token-{token}'
        await app.redis.set(key, helper.user.uuid)

        response = {
            'magic-token': token
        }
        return json(response)

def check_magic_token(request):
    token = request.json.get('magic_token', '')
    key = f'magic-token-{token}'

    retrieval = await request.app.redis.get(key)
    if retrieval is None:
        raise Exception('Token expired or invalid')
    retrieval = str(retrieval)

    user = User.get(uuid=retrieval)

    return user

Initialize(
    app,
    authenticate=check_magic_token,
    class_views=[
        # The path will be relative to the url prefix (which defaults to /auth)
        ('/magic-login', MagicLoginHandler)
    ])
```

**Note:** Your class based views will probably also need to handle preflight requests, so do not forget to add an options response.

```
async def options(self, request):
    return response.text('', status=204)
```

### 5.2.5 Component Overrides

There are **three** components that are used under the hood that you can subclass and control:

- `Authentication` - for more advanced usage, see source code, or ask a question
- `Configuration` - see *Configuration* for more information
- `Responses` - see *Endpoints and Responses* for more information

Simply import, modify, and attach.

```
from sanic_jwt import Authentication, Configuration, Responses, Initialize

class MyAuthentication(Authentication):
    pass

class MyConfiguration(Configuration):
    pass

class MyResponses(Responses):
    pass

Initialize(
    app,
    authentication_class=MyAuthentication,
    configuration_class=MyConfiguration,
    responses_class=MyResponses,)
```

## 5.3 The `initialize` method

The old method for initializing Sanic JWT was to do so with the `initialize` method. It still works, and is in fact now just a wrapper for the `Initialize` class. However, it is recommended that you use the class because it is more explicit that you are declaring a new instance. And, even though there are no plans (as of June 2018) to depracate this, some day it likely will be.

CHAPTER 6

# Endpoints and Responses

Sanic JWT sets itself up to run as a Sanic Blueprint at the `/auth` path.

```
# Default
http://localhost:8000/auth
```

This is can be changed via the `url_prefix` setting. *See settings for more*.

```
Initialize(app, url_prefix='/api/authentication')
```

All Sanic JWT endpoints will now be available at:

```
# Custom
http://localhost:8000/api/authentication
```

## 6.1 Default Endpoints

By default, there are four endpoints that ship with Sanic JWT. You can change the path that they attach to by following configuration pattern below:

```
Initialize(
    app,
    path_to_authenticate='/my_authenticate',
    path_to_retrieve_user='/my_retrieve_user',
    path_to_verify='/my_verify',
    path_to_refresh='/my_refresh',
)
```

### 6.1.1 Authenticate

**Default Path**: `/auth`
**Acceptable Methods**: `POST`
**Purpose**: Generates an access token if the `authenticate` *method* is **truthy**.
**Example**:

Request

```
curl -X POST -H "Content-Type: application/json" -d '{"username": "<USERNAME>",
↪"password": "<PASSWORD>"}' http://localhost:8000/auth
```

Response

```
200 Response
{
    "access_token": "<JWT>"
}
```

### 6.1.2 Verification

**Default Path**: `/auth/verify`
**Acceptable Methods**: `GET`
**Purpose**: Check whether or not a given access token is valid.
**Example**:

Request

```
curl -X GET -H "Authorization: Bearer <JWT>" http://localhost:8000/auth/verify
```

Response

```
200 Response
{
    "valid": true
}

## or

400 Response
{
    "valid": false,
    "reason": "Signature has expired"
}
```

### 6.1.3 Current User Details

**Default Path**: `/auth/me`

**Acceptable Methods**: `GET`
**Purpose**: Retrieve information about the currently authenticated user.
**Example**:

Request

```
curl -X GET -H "Authorization: Bearer <JWT>" http://localhost:8000/auth/me
```

Response

```
200 Response
{
    "me": {
        user_id": 123456
    }
}
```

**Note:** Because this package does not know about you user management layer, you need to have a user object that either is a `dict` or a python object instance with a `to_dict()` method. The output of these methods will be used to generate the `/me` response.

### 6.1.4 Refresh Token

**Default Path**: `/auth/refresh`
**Acceptable Methods**: `POST`
**Purpose**: Ask for a new access token given an existing refresh token
**Example**:

Request

```
curl -X POST -H "Content-Type: application/json" -H "Authorization: Bearer <JWT>" -d '
↪{"refresh_token": "<REFRESH TOKEN>"}' http://localhost:8000/auth/refresh
```

Response

```
{
    "access_token": "<JWT>"
}
```

**Note:** Do not forget to supply an existing `access_token`. Even if it is expired, you **must** send the token along so that the application can get the `user_id` from the token's payload and cross reference it with the `refresh_token`. Think of it as an additional level of security. To understand why, checkout Issue #52.

## 6.2 Modify Responses

The responses for each of the default endpoints is extendable by subclassing the `Responses` class, and hooking into the appropriate method. Just make sure you return a `dict`.

Your custom `Responses` should be hooked up to Sanic JWT using the `responses_class` keyword argument on the `Initialize` instance.

```
from sanic_jwt import Responses

class MyResponses(Responses):
    @staticmethod
    def extend_authenticate(request,
                            user=None,
                            access_token=None,
                            refresh_token=None):
        return {}

    @staticmethod
    def extend_retrieve_user(request, user=None, payload=None):
        return {}

    @staticmethod
    def extend_verify(request, user=None, payload=None):
        return {}

    @staticmethod
    def extend_refresh(request,
                       user=None,
                       access_token=None,
                       refresh_token=None,
                       purported_token=None,
                       payload=None):
        return {}

Initialize(app, responses_class=MyResponses)
```

## 6.3 Custom Endpoints

Sometimes you may find the need to add another endpoint to your authentication system. You can do this by hooking it up at *initialization*.

```
from sanic_jwt import BaseEndpoint

class MyEndpoint(BaseEndpoint):
    ...

my_views = (
    ('/my-view', MyEndpoint),
)

Initialize(app, class_views=my_views)
```

**Example**:

What if we wanted a `/register` endpoint? It could easily be added like this:

```
from sanic_jwt import BaseEndpoint

class Register(BaseEndpoint):
    async def post(self, request, *args, **kwargs):
        username = request.json.get('username', None)
        email = request.json.get('email', None)

        helper = MyCustomUserAuthHelper()
        user = helper.register_new_user(username, email)

        access_token, output = await self.responses.get_access_token_output(
            request,
            user,
            self.config,
            self.instance)

        refresh_token = await self.instance.auth.get_refresh_token(request, user)
        output.update({
            self.config.refresh_token_name(): refresh_token
        })

        response = self.responses.get_token_reponse(
            request,
            access_token,
            output,
            refresh_token=refresh_token,
            config=self.config)


        return response

my_views = (
    ('/register', Register),
)

# Please note that this Initialize instance is incomplete.
# It is missing handlers for: authenticate, store_refresh_token, and
# retrieve_refresh_token.
# It is meant as illustrative purposes on how you might approach this.
# See https://github.com/ahopkins/sanic-jwt/issues/111 for more information.
Initialize(app, class_views=my_views, ...)
```

You hook up your custom endpoints at *initialization* by providing `Initialize` with a `class_views` argument naming your endpoint and its path.

```
my_endpoints = (
    ('/path/to/endpoint', MyCustomClassBasedView)
)
```

**Note:** It must be a class based view. While it is certainly possible to subclass Sanic's `sanic.views.HTTPMethodView`, it is recommended that you subclass `sanic_jwt.BaseEndpoint` instead so you have access to:

- `self.instance` (the current Sanic JWT),

- `self.config` (all current configurations), and
- `self.responses` (the current response class instance).

## 6.4 Exception Handling

You can customize how Sanic JWT handles responses on an exception by subclassing the `Responses` class, and overriding `exception_response`.

```
from sanic_jwt import Responses

class MyResponses(Responses):
@staticmethod
def exception_response(request, exception):
    exception_message = str(exception)
    return json({
        'error': True,
        'message': f'You encountered an exception: {exception_message}'
    }, status=exception.status_code)

Initialize(app, response_class=MyResponses)
```

## 6.5 Microservices

One of the benefits of a lightweight framework like Sanic is that it makes building microservice architectures simple, and flexible. If you are building a microservice application, likely you do not want all of your services to have the `/auth` endpoints!

Instead, you probably only want to authenticate against a single service, and use the token generated there among all your services. This can be easily accomplished with the `auth_mode=True` *Configuration*. Set it to `True` on your authentication service, and `False` everywhere else. All the decorators will still work as expected.

```
# Authentication service
Initialize(app, authenticate=lambda: True)

# Every other service
Initialize(app, auth_mode=False)
```

Now, the `/auth` endpoints are only on your authentication service, but the access token can be used on ANY of your other services.

**Note:** This works **only** if each of the services has the same `secret`.

CHAPTER 7

# Payloads

*As discussed*, JWTs have a payload that is essentially a key/value store of information.

With Sanic JWT, there are three main uses of the payload:

- passing claims (See *What is a JWT?* for more information)
- passing scope (See *Scopes* for more information)
- passing arbitrary information to the client

## 7.1 Built in Claims

Sanic JWT ships with the capability to add, and later verify, **five** standard claims: `exp`, `nbf`, `iat`, `iss`, and `aud`.

### 7.1.1 Expires - `exp`

**Purpose**: This claim is a timestamp that dictates when the access token will no longer be available. Because JWT access tokens cannot be invalidated after they are issued, they are typically given a short life span.
**Enabled by default**: Yes.

**Warning:** It is possible to disable token expiration. Do **NOT** do this unless you know what you are doing and why you are doing it.

```
Initialize(app, verify_exp=False)
```

Okay, great. You know what you are doing. It is still revommended that you **NOT** do this. Are you sure you know what you are doing?

### 7.1.2 Audience - `aud`

**Purpose**: This claim identifies what service the JWT is intended to be used with. Typically it is a URI or other identifier that says the name of the client server that is supposed to be validating the token.
**Enabled by default**: No.
**How to use**: Set `claim_aud` to a `str`
**Example**:

```
Initialize(app, claim_aud='my_client_domain.com')
```

### 7.1.3 Issued at - `iat`

**Purpose**: This claim is a timestamp that provides the creation time of the JWT.
**Enabled by default**: No.
**How to use**: Set `claim_iat` to `True`
**Example**:

```
Initialize(app, claim_iat=True)
```

### 7.1.4 Issuer - `iss`

**Purpose**: This claim is typically a URI or other identifier to say who created and issued the token.
**Enabled by default**: No.
**How to use**: Set `claim_iss` to a `str`
**Example**:

```
Initialize(app, claim_iss='my_server_domain.com')
```

### 7.1.5 Not before - `NBF`

**Purpose**: This claim is a timestamp that allows the token to be created and issued, but not yet enabled for usage until after a certain time.
**Enabled by default**: No.
**How to use**: Set `claim_nbf` to `True`, and `claim_nbf_delta` to an offset in seconds
**Example**:

```
Initialize(app, claim_nbf=True, claim_nbf_delta=(60 * 3))
```

## 7.2 Custom Claims

Sometimes you may find a need to add claims to a JWT beyond what is built into Sanic JWT.

To do so, simply subclass `Claim` and register them at *initialization* by providing the custom claim class in a list to `custom_claims`.

```
from sanic_jwt import Claim, Initialize

MyCustomClaim(Claim):
    key = 'foo'

    def setup(self, payload, user):
        return 'bar'

    def verify(self, value):
        return value == 'bar'

Initialize(..., custom_claims=[MyCustomClaim])
```

There are three attributes that a `Claim` must have: `key`, `setup`, and `verify`.

`key`: The name of the claim and the key that will be inserted into the payload.
`setup`: A method to be run at the time the payload is created. It should return the value of the claim.
`verify`: A method to be run when a token is being verified. It should return a `boolean` whether or not the claim has been met.

---

## 7.3 Extra Verifications

Besides registering custom claims, sometimes you may find the need to do additional verifications on a payload. For example, perhaps you want to run checks that span more than one claim on the payload.

To accomplish this, you can register a list of methods (that each return a `boolean`) at *initialization* by providing the list to `extra_verifications`.

```
def check_number_of_claims(payload):
    return len(payload.keys()) == 5

extra_verifications = [check_number_of_claims]
Initialize(
    ...,
    extra_verifications=extra_verifications
)
```

---

## 7.4 Payload Handlers

*As discussed*, there are a few handlers on the `Initialize` instance that can be used to modify the payload.

### 7.4.1 Adding Scopes

**Argument**: `add_scopes_to_payload`
**Purpose**: If you are using the `@scoped` *decorator*, then you will need a way to inject the `payload` with the user's scopes. It should return either a single scope, or a list of scopes. *Read about scopes* for more information.
**Return**: `str` or a `list` of `str`
**Example**:

```
async def my_scope_extender(user, *args, **kwargs):
    return user.scopes

Initialize(app, add_scopes_to_payload=my_scope_extender)
```

**Note:** The return of the `authenticate` method will be injected into this handler as `user` for your convenience.

### 7.4.2 Extending the payload

**Argument**: `extend_payload`
**Purpose**: To add an arbitrary set of claims or information to the payload.
**Return**: `dict`
**Example**:

```
def my_foo_bar_payload_extender(payload, *args, **kwargs):
    payload.update({
        'foo': 'bar'
    })

    return payload

Initialize(app, extend_payload=my_foo_bar_payload_extender)
```

## 7.5 Token signing

JWTs need to be digitally signed to allow for cryptographically verifying that an access token was generated by your application.

```
secret = 'XXXXXXXXXXXXXXXXXXXXXXXX'

Initialize(
    app,
    secret=mysecret)
```

There are several hashing algorithms that can be used to accomplish this. Check out the *Configuration* page to see which algorithms are supported, and read this for more information.

If you decide to use an RSA or an EC algorithm, then you **must** provide Sanic JWT with both a public key and a private key to handle the encoding and decoding of the tokens.

```
from pathlib import Path

public_ec_key = Path('/path') / 'to' / 'my-ec-public-key.pem'
private_ec_key = Path('/path') / 'to' / 'my-ec-private-key.pem'

Initialize(
    app,
    public_key=public_ec_key,
    private_key=private_ec_key,
    algorithm='ES256')
```

CHAPTER 8

# Protecting Routes

The purpose of this package, beyond the creation of JWTs, is to protect routes so that only users with a valid access token can reach certain resources. Endpoints in your application can be protected using the `@protected` decorator.

## 8.1 The `@protected` decorator

**Purpose**: To protect an endpoint from being accessed without a valid access token.
**Example**:

```
from sanic_jwt.decorators import protected


@app.route("/")
async def open_route(request):
    return json({"protected": False})


@app.route("/protected")
@protected()
async def protected_route(request):
    return json({"protected": True})
```

Now, anyone can access the `/` route. But, only users that pass a valid access token can reach `/protected`.

If you have initialized Sanic JWT on a `Blueprint`, then you will need to pass the instance of that blueprint into the `@protected` decorator.

```
bp = Blueprint('Users')
Initialize(bp, app=app)

@bp.get('/users/<id>')
@protected(bp)
async def users(request, id):
    ...
```

Alternatively (and probably preferably), you can also access the decorator from the `Initialize` instance. This makes it easier if you forget to pass the `Blueprint`.

```
bp = Blueprint('Users')
sanicjwt = Initialize(bp, app=app)

@bp.get('/users/<id>')
@sanicjwt.protected()
async def users(request, id):
    ...
```

### 8.1.1 Class based views

Using the standard Sanic methodology, you can protect class based views with the same decorator.

```
class PublicView(HTTPMethodView):
def get(self, request):
    return json({"protected": False})


class ProtectedView(HTTPMethodView):
    decorators = [protected()]

    async def get(self, request):
        return json({"protected": True})

app.add_route(PublicView.as_view(), '/')
app.add_route(ProtectedView.as_view(), '/protected')
```

---

## 8.2 Passing the Token

There are two general methodologies for passing a token: cookie based, and header based. By default, Sanic JWT will expect you to send tokens thru HTTP headers.

```
curl -X GET -H "Authorization: Bearer <JWT>" http://localhost:8000/auth/me
```

### 8.2.1 Header Tokens

Header tokens are passed by adding an `Authorization` header that consists of two parts:

1. the word `Bearer`
2. the JWT access token

If you would like, you can modify this behavior by changing the *configuration* for `authorization_header` and `authorization_header_prefix`.

```
Initialize(
    app,
    authorization_header='somecustomheader',
    authorization_header_prefix='MeFirst',)
```

```
curl -X GET -H "somecustomheader: MeFirst <JWT>" http://localhost:8000/auth/me
```

### 8.2.2 Cookie Tokens

If you would like to use tokens in cookies instead of headers, you need to first set `cookie_set=True`

```
Initialize(app, cookie_set=True)
```

Now, Sanic JWT will reject any request that does not have a valid access token in its cookie. As the developer, you can control how the cookie is generated with the following settings:

`cookie_domain` - changes domain associated with a cooke (defaults to '')
`cookie_httponly` - whether to set an httponly flag on the cookie (defaults to `True`)
`cookie_access_token_name` - the name where the cookie is stored for access token
`cookie_refresh_token_name` - the name where the cookie is stored for refresh token

```
Initialize(
    app,
    cookie_set=True,
    cookie_domain='mydomain.com',
    cookie_httponly=False,
    cookie_access_token_name='some-token',)
```

**Warning:** If you are using cookies to pass JWTs, then it is recommended that you do **not** disable `cookie_httponly`. Doing so means that any javascript running on the client can access the token. Bad news.

### 8.2.3 Query String Tokens

Sometimes, both header based authentication and cookie based authentication will not be enough. A third option is available to look for tokens inside query string arguments:

http://localhost?access_token=<JWT>

This can be enabled with `query_string_set=True`. One potential use for this would be authentication of a websocket endpoint where sending headers and cookies may be more challenging due to Javascript limitations.

**Warning:** In most scenarios, it is advisable to **not** use query strings for authentication. One of the biggest reasons is that the tokens may be easily leaked if a URL is copied and pasted, or because the token may end up in server logs. However, the option is available if you need it and you feel comfortable that you can mitigate any risks.

### 8.2.4 Both Header and Cookie

If you enable `cookie_set`, you will get a `MissingAuthorizationCookie` exception if the cookie is not present. However, sometimes you may want to fall back and look for a header token if the cookie is not there.

Is such cases, change `cookie_strict` to `False`.

```
Initialize(
    app,
    cookie_set=True,
    cookie_strict=False,)
```

This will now tell Sanic JWT to look for the cookie first. If it is not present, before throwing an exception, it will fallback and look for an `Authorization` header.

### 8.2.5 Per view declaration

Perhaps you realize that you would like to make the declaration on a single view? Most of your views will operate using a cookie, but one particular endpoint (for whatever reason) will best be served to accept headers. Not a problem. You can simply pass in the configuration parameters right into the decorator!

```
Initialize(
    app,
    cookie_set=True,
    cookie_strict=False,)

@app.route("/protected_by_header")
@protected(cookie_set=False)
async def protected_by_header_route(request):
    ...
```

Learn more about *configuration overrides*.

---

**Note:** This paradigm works for all configurations. Feel free to experiment and change config settings at the lowest level you might need them.

---

## 8.3 Advanced Decorators

Want to have a greater level of control? Instead of just importing the decorators from the `sanic_jwt.decorators` module, you can also use the decorator directly off your initialized Sanic JWT instance!

```
sanicjwt = Initialize(app)

@app.route("/protected")
@sanicjwt.protected()
async def protected_route(request):
    ...
```

This also works for blueprints (and has the added advantage that you no longer need to pass the *bp* instance in.

```
bp = Blueprint('Users')
Initialize(bp, app=app)

@bp.get('/users/<id>')
@bp.protected()
async def users(request, id):
    ...
```

**Note:** This concept of having instance based decorators also works for the `scoped` decorator: `bp.scoped('foobar')`; and the `@inject_user` decorator.

### 8.3.1 `@inject_user` decorator

You've gone thru the hard work and added a `retrieve_user` method. You might as well be able to reap the benefits by leveraging that method to inject your user data into your endpoints.

```
@app.route("/protected/user")
@inject_user()
@protected()
async def my_protected_user(request, user):
    return json({"user_id": user.user_id})
```

CHAPTER 9

# Scopes

In addition to protecting routes to authenticated users, they can be scoped to require one or more scopes by applying the `@scoped()` decorator. This means that only users with a particular scope can access a particular endpoint.

**Note:** If you are using the `@scoped` decorator, you do **NOT** also need the `@protected` decorator. It is assumed that if you are scoping the endpoint, that it is also meant to be protected.

## 9.1 Requirements for a scoped

A **scope** is a string that consists of two parts:

- *namespace*
- *actions*

For example, it might look like this: `user:read`.

**namespace** - A scope can have either one namespace, or no namespaces
**action** - A scope can have either no actions, or many actions

## 9.2 Example Scopes

```
Example #1
scope:       user
namespace:   user
action:      --

Example #2
scope:       user:read
namespace:   user
action:      read

Example #3
scope:       user:read:write
namespace:   user
action:      [read, write]

Example #4
scope:       :read
namespace:   --
action:      read

Example #5
scope:       :read:write
namespace:   --
action:      [read, write]
```

## 9.3 How are scopes accepted?

In defining a scoped route, you define one or more scopes that will be acceptable.

A scope is accepted if the payload contains a scope that is **equal to or higher** than what is required.

For sake of clarity in the below explanation, `required_scope` means the scope that is required for access, and `user_scope` is the scope that the access token has in its payload.

A scope is acceptable . . .

- If the `required_scope` namespace and the `user_scope` namespace are equal

```
# True
required_scope = 'user'
user_scope = 'user'
```

- **If the `required_scope` has actions, then the `user_scope` must be:**
  - top level (no defined actions), or
  - also has the same actions

```
# True
required_scope = 'user:read'
user_scope = 'user'

# True
required_scope = 'user:read'
user_scope = 'user:read'
```

(continues on next page)

(continued from previous page)

```
# True
required_scope = 'user:read'
user_scope = 'user:read:write'

# True
required_scope = ':read'
user_scope = ':read'

# False
required_scope = 'user:write'
user_scope = 'user:read'
```

## 9.4 Examples

Here is a list of example scopes and whether they pass or not:

| required scope | user scope(s) | outcome |
|---|---|---|
| 'user' | ['something'] | **False** |
| 'user' | ['user'] | **True** |
| 'user:read' | ['user'] | **True** |
| 'user:read' | ['user:read'] | **True** |
| 'user:read' | ['user:write'] | **False** |
| 'user:read' | ['user:read:write'] | **True** |
| 'user' | ['user:read'] | **False** |
| 'user:read:write' | ['user:read'] | **False** |
| 'user:read:write' | ['user:read:write'] | **True** |
| 'user:read:write' | ['user:write:read'] | **True** |
| 'user' | ['something', 'else'] | **False** |
| 'user' | ['something', 'else', 'user'] | **True** |
| 'user:read' | ['something:else', 'user:read'] | **True** |
| 'user:read' | ['user:read', 'something:else'] | **True** |
| ':read' | [':read'] | **True** |
| ':read' | ['admin'] | **True** |

## 9.5 The @scoped decorator

### 9.5.1 Basics

In order to protect a route from being accessed by tokens without the appropriate scope(s), pass in one or more scopes:

```
@app.route("/protected/scoped/1")
@scoped('user')
async def protected_route1(request):
    return json({"protected": True, "scoped": True})
```

In the above example, only an access token with a payload containing a scope for `user` will be accepted (such as the payload below).

```
{
    "user_id": 1,
    "scopes: ["user"]
}
```

You can also define multiple scopes:

```
@scoped(['user', 'admin'])
```

In the above example with a `['user', 'admin']` scope, a payload **MUST** contain both `user` and `admin`.

But, what if we only want to require one of the scopes, and not both `user` AND `admin`? Easy:

```
@scoped(['user', 'admin'], False)
```

Now, having a scope of either `user` OR `admin` will be acceptable.

If you have initialized Sanic JWT on a `Blueprint`, then you will need to pass the instance of that blueprint into the `@scoped` decorator.

```
bp = Blueprint('Users')
Initialize(bp)

@bp.get('/users/<id>')
@scoped(['user', 'admin'], initialized_on=bp)
async def users(request, id):
    ...
```

### 9.5.2 Parameters

The `@scoped()` decorator takes three parameters:

- `scopes`
- `requires_all` - default `True`
- `require_all_actions` - default `True`

#### `scopes` - Required

Either a single `string`, or a `list` of strings that are the defined scopes for the route. Or, a `callable` or `awaitable` that returns the same.

```
@scoped('user')
...
```

Or

```
@scoped(['user', 'admin'])
...
```

Or

```
def get_some_scopes(request, *args, **kwargs):
    return ['user', 'admin']
```

(continues on next page)

(continued from previous page)

```
@scoped(get_some_scopes)
...
```

Or

```
async def get_some_scopes(request, *args, **kwargs):
    return await something_that_returns_scopes()

@scoped(get_some_scopes)
...
```

#### `require_all` - Optional

A `boolean` that determines whether all of the **defined scopes**, or just one must be satisfied. Defaults to `True`.

```
@scoped(['user', 'admin'])
...
# A payload MUST have both 'user' and 'admin' scopes


@scoped(['user', 'admin'], require_all=False)
...
# A payload can have either 'user' or 'admin' scope
```

#### `require_all_actions` - Optional

A `boolean` that determines whether all of the **actions** on a defined scope, or just one must be satisfied. Defaults to `True`.

```
@scoped(':read:write')
...
# A payload MUST have both the `:read` and `:write` actions in scope


@scoped(':read:write', require_all_actions=False)
...
# A payload can have either the `:read` or `:write` action in scope
```

## 9.6 Handler

See *Payloads* for how to add scopes to a payload using `add_scopes_to_payload`.

## 9.7 Sample Code

```
from sanic import Sanic
from sanic.response import json
from sanic_jwt import exceptions
from sanic_jwt import initialize
```

(continues on next page)

(continued from previous page)

```
from sanic_jwt.decorators import protected
from sanic_jwt.decorators import scoped


class User(object):
    def __init__(self, id, username, password, scopes):
        self.user_id = id
        self.username = username
        self.password = password
        self.scopes = scopes

    def __str__(self):
        return "User(id='%s')" % self.id


users = [
    User(1, 'user1', 'abcxyz', ['user']),
    User(2, 'user2', 'abcxyz', ['user', 'admin']),
    User(3, 'user3', 'abcxyz', ['user:read']),
    User(4, 'user4', 'abcxyz', ['client1']),
]

username_table = {u.username: u for u in users}
userid_table = {u.user_id: u for u in users}


async def authenticate(request, *args, **kwargs):
    username = request.json.get('username', None)
    password = request.json.get('password', None)

    if not username or not password:
        raise exceptions.AuthenticationFailed("Missing username or password.")

    user = username_table.get(username, None)
    if user is None:
        raise exceptions.AuthenticationFailed("User not found.")

    if password != user.password:
        raise exceptions.AuthenticationFailed("Password is incorrect.")

    return user


async def my_scope_extender(user, *args, **kwargs):
    return user.scopes


app = Sanic()
Initialize(
    app,
    authenticate=authenticate,
    add_scopes_to_payload=my_scope_extender)


@app.route("/")
async def test(request):
    return json({"hello": "world"})
```

(continues on next page)

(continued from previous page)

```
@app.route("/protected")
@protected()
async def protected_route(request):
    return json({"protected": True, "scoped": False})


@app.route("/protected/scoped/1")
@protected()
@scoped('user')
async def protected_route1(request):
    return json({"protected": True, "scoped": True})


@app.route("/protected/scoped/2")
@protected()
@scoped('user:read')
async def protected_route2(request):
    return json({"protected": True, "scoped": True})


@app.route("/protected/scoped/3")
@protected()
@scoped(['user', 'admin'])
async def protected_route3(request):
    return json({"protected": True, "scoped": True})


@app.route("/protected/scoped/4")
@protected()
@scoped(['user', 'admin'], False)
async def protected_route4(request):
    return json({"protected": True, "scoped": True})


@app.route("/protected/scoped/5")
@scoped('user')
async def protected_route5(request):
    return json({"protected": True, "scoped": True})


@app.route("/protected/scoped/6/<id>")
@scoped(lambda *args, **kwargs: 'user')
async def protected_route6(request, id):
    return json({"protected": True, "scoped": True})


def client_id_scope(request, *args, **kwargs):
    return 'client' + kwargs.get('id')


@app.route("/protected/scoped/7/<id>")
@scoped(client_id_scope)
async def protected_route7(request, id):
    return json({"protected": True, "scoped": True})
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    app.run(host="127.0.0.1", port=8888)
```

CHAPTER 10

# Refresh Tokens

## 10.1 What is a refresh token?

Access tokens are disposable. Because they cannot be expired, they have a *short* lifespan. Every time an access token expires, the client needs to reauthenticate. Access tokens are generated, and sent to the client. They are not persisted. This means that they cannot be revoked.

Refresh tokens solve these two problems. It is a token that is stored by the server. At any time a client can send the refresh token to the server and ask for a new access token.

The server takes the refresh token, looks up in its data store to see if it is acceptable. If yes, then a new access token is generated and sent to the client.

In a best practices scenario, refresh tokens and access tokens work together to provide a user friendly, yet secure, authentication environment.

## 10.2 Configuration

Sanic JWT facilitates the creation and passing of refresh tokens. However, just like with authentication, the storage and retrieval of the tokens is left to the developer. Why? This allows the you to decide how to persist the token, and allows you to deactivate a token at any time.

There are three steps needed:

1. Enable refresh tokens via the settings configuration (`refresh_token_enabled`)
2. Initialize Sanic JWT with a method for storing refresh tokens (`store_refresh_token`)
3. Initialize Sanic JWT with a method for retrieving refresh tokens (`retrieve_refresh_token`)

## 10.3 Enable refresh tokens

Out of the box, Sanic JWT will **not** generate refresh tokens for you. If you want to make use of them, simply enable them as you would any other *Configuration*. The easiest is probably just to pass `refresh_token_enabled` into `Initialize`.

```
Initialize(
    app,
    authenticate=lambda: True,
    refresh_token_enabled=True,)
```

---

## 10.4 Handlers

As mentioned, there are two **required** handlers you must create if you would like to provide refresh tokens to users.

### 10.4.1 1. `store_refresh_token`

When running `Initialize`, pass it an attribute that can go to your data store and persist a refresh token. The method is passed `user_id` (which comes from the user object returned from the `authenticate` method), and `refresh_token`.

It can be **either** a callable or an awaitable. Here are two different examples that do the same thing: persist a `refresh_token` to Redis.

```
async def store_refresh_token(user_id, refresh_token, *args, **kwargs):
    key = f'refresh_token_{user_id}'
    await aredis.set(key, refresh_token)
```

```
def store_refresh_token(user_id, refresh_token, *args, **kwargs):
    key = f'refresh_token_{user_id}'
    redis.set(key, refresh_token)
```

Then you hook it up to the initialize script like this:

```
Initialize(
    app,
    authenticate=lambda: True,
    store_refresh_token=store_refresh_token)
```

### 10.4.2 2. `retrieve_refresh_token`

When running `Initialize`, pass it an attribute that can go to your data store and retrieve a refresh token. The method is passed `user_id` (which comes from the user object returned from the `authenticate` method), and the `request` object to determine if it contains what is needed to retrieve a token.

It can be **either** a callable or an awaitable. Here are two different examples that all do the same thing: retrieve a `refresh_token` from Redis.

```
async def retrieve_refresh_token(request, user_id, *args, **kwargs):
    key = f'refresh_token_{user_id}'
    return await aredis.get(key)
```

```
def retrieve_refresh_token(request, user_id, *args, **kwargs):
    key = f'refresh_token_{user_id}'
    return redis.get(key)
```

Then you hook it up to the initialize script like this:

```
Initialize(
    app,
    authenticate=lambda: True,
    retrieve_refresh_token=retrieve_refresh_token)
```

---

## 10.5 Using the refresh token

In order to get a new access token, you need to hit the refresh token endpoint. See *Endpoints and Responses* for more information.

## 10.6 Can I have an expirable refresh token?

This question has come up a couple times in the past. Allow us to explain why this is not a feature of Sanic JWT.

When enabled, Sanic JWT issues a refresh token that is a `utf-8` encoded string containing 24 characters. It is **not** a JWT. Therefore, it does not have a payload and is not subject to validation.

The core of deciding whether or not to accept a refresh token is left to the developer. That is the purpose of `store_refresh_token` and `retrieve_refresh_token`.

Therefore, if you would like to expire the token, then this is something for you to handle at the application layer.

For more information on this, see Issue #34 and Issue #66.

We agree. Having the control expire a token is wonderful. Having it be done automatically? Even better. But, this is something that seems better left to the individual developer to decide upon, rather than Sanic JWT making that choice for you. Our goal here is to enable the developer to build a more secure platform, not make decisions for them.

### 10.6.1 But, I really want one!

Okay, fine. If you really would like to issue a JWT, or any kind of token, you can. Sanic JWT gives you the control to override our default method of generating refresh tokens. Something like this would work:

```
import uuid

def generate_refresh_token(*args, **kwargs):
    return str(uuid.uuid4())

Initialize(
    ...
    generate_refresh_token=generate_refresh_token,
)
```

You as the developer have the control to issue whatever you would like. If you want that refresh token to be a JWT, go for it! You will need to generate it, and then validate it in the `retrieve_refresh_token` handler. I'll let the exercise be up to you, but feel free to post an issue and ask for help.

CHAPTER 11

# Exceptions

There is a standard set of exceptions that Sanic JWT uses to communicate. Here is a subset of exceptions that you may find helpful while creating your application.

- `AuthenticationFailed`
- `MissingAuthorizationHeader`
- `MissingAuthorizationCookie`
- `InvalidAuthorizationHeader`
- `MissingRegisteredClaim`
- `Unauthorized`

It is recommended that you use exceptions in your Sanic JWT implementation. If an exception occurs, then you can control what message to return to the client. See *Endpoints and Responses* for more information.

CHAPTER 12

# Configuration

## 12.1 How to add settings

There are several ways to configure Sanic JWT depeding upon your project's complexity and use case.

### 12.1.1 The Sanic way

Any way that Sanic offers to load configration will work. Simply convert the setting name to all caps, and add the `SANIC_JWT_` prefix.

```
app = Sanic()
app.config.SANIC_JWT_ACCESS_TOKEN_NAME = 'jwt'

Initialize(app)
```

If you choose this approach, Sanic JWT will only know about configurations set _BEFORE_ you call `Initialize`.

### 12.1.2 Inline at initialization

One of the easiest methods is to simply name the setting and value as a keyword argument on the `Initialize` object.

```
Initialize(
    app,
    access_token_name='jwt')
```

### 12.1.3 `Configuration` class

For a more fine grain control, you can subclass the `Configuration` class and provide the settings as attributes on the class.

```
from sanic_jwt import Configuration

class MyConfiguration(Configuration):
    access_token_name='jwt'

Initialize(
    app,
    configuration_class=MyConfiguration)
```

What if you need to calculate a setting? No problem. Each of the settings can be declared at initialization with the `set_<setting>()` method.

```
from sanic_jwt import Configuration

class MyConfiguration(Configuration):
    def set_access_token_name(self):
        return 'jwt'

Initialize(
    app,
    configuration_class=MyConfiguration)
```

But, it does not need to be a callable. This works too:

```
from sanic_jwt import Configuration

class MyConfiguration(Configuration):
    set_access_token_name = 'jwt'

Initialize(
    app,
    configuration_class=MyConfiguration)
```

Okay ... need to go even **further**? You can also have a setting evaluated on each request with the `get_<setting>()` method:

```
auth_header_key = "x-authorization-header"

class MyConfig(Configuration):

    def get_authorization_header(self, request):
        if auth_header_key in request.headers:
            return request.headers.get(auth_header_key)

        return "authorization"

Initialize(
    app,
    configuration_class=MyConfig
)
```

This brings up an important point. If you go with the getter method, then in order to not waste resources, it will be evaluated only **one** time per request. The output of your getter will be cached for the lifespan of that request only.

As you can see, the getter method is passed the `request` object as a parameter.

## 12.2 Settings

### 12.2.1 access_token_name

**Purpose**: The key to be used by the application to identify the access token.
**Default**: `'access_token'`

### 12.2.2 algorithm

**Purpose**: The hashing algorithm used to generate the tokens. Your available options are listed below.
**Default**: `'HS256'`

```
HS256 - HMAC using SHA-256 hash algorithm (default)
HS384 - HMAC using SHA-384 hash algorithm
HS512 - HMAC using SHA-512 hash algorithm
ES256 - ECDSA signature algorithm using SHA-256 hash algorithm
ES384 - ECDSA signature algorithm using SHA-384 hash algorithm
ES512 - ECDSA signature algorithm using SHA-512 hash algorithm
RS256 - RSASSA-PKCS1-v1_5 signature algorithm using SHA-256 hash algorithm
RS384 - RSASSA-PKCS1-v1_5 signature algorithm using SHA-384 hash algorithm
RS512 - RSASSA-PKCS1-v1_5 signature algorithm using SHA-512 hash algorithm
PS256 - RSASSA-PSS signature using SHA-256 and MGF1 padding with SHA-256
PS384 - RSASSA-PSS signature using SHA-384 and MGF1 padding with SHA-384
PS512 - RSASSA-PSS signature using SHA-512 and MGF1 padding with SHA-512
```

### 12.2.3 auth_mode

**Purpose**: Whether to enable the `/auth` endpoints or not. Helpful for microservice applications.
**Default**: `True`

### 12.2.4 authorization_header

**Purpose**: The HTTP request header used to identify the token.
**Default**: `'authorization'`

### 12.2.5 authorization_header_prefix

**Purpose**: The prefix for the JWT in the HTTP request header used to identify the token.
**Default**: `'Bearer'`

### 12.2.6 authorization_header_refresh_prefix

**Purpose**: *Reserved. Not in use.*
**Default**: `'Refresh'`

### 12.2.7 claim_aud

**Purpose**: The aud (audience) claim identifies the recipients that the JWT is intended for. Each principal intended to process the JWT MUST identify itself with a value in the audience claim. If the principal processing the claim does not identify itself with a value in the aud claim when this claim is present, then the JWT MUST be rejected. In the general case, the aud value is an array of case-sensitive strings, each commonly containing a string or URI value. In the special case when the JWT has one audience, the aud value MAY be a single case-sensitive string containing a string or URI value. Use of this claim is OPTIONAL. If you assign a str value, then the aud claim will be generated for all requests, and will be required to verify a token.
**Default**: `None`

### 12.2.8 claim_iat

**Purpose**: The iat (issued at) claim identifies the time at which the JWT was issued. This claim can be used to determine the age of the JWT. Its value will be a numeric timestamp. Use of this claim is *OPTIONAL*. If you assign a `True` value, then the iat claim will be generated for all requests.
**Default**: `False`

### 12.2.9 claim_iss

**Purpose**: The iss (issuer) claim identifies the principal that issued the JWT. The iss value is a case-sensitive string usually containing a string or URI value. Use of this claim is *OPTIONAL*. If you assign a str value, then the iss claim will be generated for all requests, and will be required to verify a token.
**Default**: `None`, requires a `str` value

### 12.2.10 claim_nbf

**Purpose**: The nbf (not before) claim identifies the time before which the JWT MUST NOT be accepted for processing. The processing of the nbf claim requires that the current date/time MUST be after or equal to the not-before date/time listed in the nbf claim. Implementers MAY provide for some small leeway, usually no more than a few minutes, to account for clock skew. Its value will be a numeric timestamp. Use of this claim is *OPTIONAL*. If you assign a `True` value, then the `nbf` claim will be generated for all requests, and will be required to verify a token. If `True`, the `nbf` claim will be set to the current time of the generation of the token. You can modify this with two additional settings: `nbf_delta` (the number of seconds to add to the timestamp) and `leeway` (the number of seconds of leeway you want to allow for).
**Default**: `False`

#### 12.2.11 claim_nbf_delta

**Purpose**: The offset in seconds between the moment of token generation and the moment when you would like the token to be valid in the future.
**Default**: `60 * 3`

#### 12.2.12 cookie_access_token_name

**Purpose**: The name of the cookie to be set for storing the access token if using cookie based authentication.
**Default**: `'access_token'`

#### 12.2.13 cookie_domain

**Purpose**: The domain to associate a cookie with.
**Default**: `''`

#### 12.2.14 cookie_httponly

**Purpose**: It enables HTTP only cookies. **HIGHLY recommended that you do not turn this off, unless you know what you are doing.**
**Default**: `True`

#### 12.2.15 cookie_refresh_token_name

**Purpose**: The name of the cookie to be set for storing the refresh token if using cookie based authentication.
**Default**: `'refresh_token'`

#### 12.2.16 cookie_set

**Purpose**: By default, the application will look for access tokens in the HTTP request headers. If you would instead prefer to send them through cookies, enable this to `True`.
**Default**: `False`

#### 12.2.17 cookie_strict

**Purpose**: If `cookie_set` is enabled, an exception will be raised if the cookie is not present. To allow for an authorization header to be used as a fallback, turn `cookie_strict` to `False`.
**Default**: `True`

### 12.2.18 cookie_token_name

Alias for `cookie_access_token_name`

### 12.2.19 debug

**Purpose**: Used for development and testing of the package. You will likely never need this.
**Default**: `False`

### 12.2.20 do_protection

**Purpose**: Whether or not protection should be inforced. This almost **always** should stay as `True`, unless you know what you are doing since it will effectively render the `@protected` decorator useless and all traffic will be passed thru.
**Default**: `True`

### 12.2.21 expiration_delta

**Purpose**: The length of time that the access token should be valid. *Since there is NO way to revoke an access token, it is recommended to keep this time period short, and to enable refresh tokens (which can be revoked) to retrieve new access tokens.*
**Default**: `60 * 5 * 6`, aka 30 minutes

### 12.2.22 generate_refresh_token

**Purpose**: A method to create and return a refresh token.
**Default**: `sanic_jwt.utils.generate_refresh_token`

### 12.2.23 leeway

**Purpose**: The number of seconds of leeway that the application will use to account for slight changes in system time configurations.
**Default**: `60 * 3`, aka 3 minutes

### 12.2.24 path_to_authenticate

**Purpose**: The path to the authentication endpoint.
**Default**: `'/'`

#### 12.2.25 path_to_refresh

**Purpose**: The path to the token refresh endpoint.
**Default**: `'/refresh'`

#### 12.2.26 path_to_retrieve_user

**Purpose**: The path to the view current user endpoint.
**Default**: `'/me'`

#### 12.2.27 path_to_verify

**Purpose**: The path to the token verification endpoint.
**Default**: `'/verify'`

#### 12.2.28 private_key

**Purpose**: A private key used for generating web tokens, dependent upon which hashing algorithm is used.
**Default**: `None`

#### 12.2.29 public_key

Alias for `secret`

#### 12.2.30 query_string_access_token_name

**Purpose**: The name of the cookie to be set for storing the refresh token if using query string based authentication.
**Default**: `'access_token'`

#### 12.2.31 query_string_refresh_token_name

**Purpose**: The name of the cookie to be set for storing the refresh token if using query string based authentication.
**Default**: `'refresh_token'`

#### 12.2.32 query_string_set

**Purpose**: By default, the application will look for access tokens in the HTTP request headers. If you would instead prefer to send them as a URL query string, enable this to `True`.

**Default**: `False`

### 12.2.33 query_string_strict

**Purpose**: If `query_string_set` is enabled, an exception will be raised if the query string is not present. To allow for an authorization header to be used as a fallback, turn `query_string_strict` to `False`.
**Default**: `True`

### 12.2.34 refresh_token_enabled

**Purpose**: Whether or not you would like to generate and accept refresh tokens.
**Default**: `False`

### 12.2.35 refresh_token_name

**Purpose**: The key to be used by the application to identify the refresh token.
**Default**: `'refresh_token'`

### 12.2.36 scopes_enabled

**Purpose**: Whether or not you would like to use the scopes module and add scopes to the payload.
**Default**: `False`

### 12.2.37 scopes_name

**Purpose**: The key to be used by the application to identify the scopes in the payload.
**Default**: `'scopes'`

### 12.2.38 secret

**Purpose**: The secret used by the hashing algorithm for generating and signing JWTs. This should be a string unique to your application. Keep it safe.
**Default**: `'This is a big secret. Shhhhh'`

#### 12.2.39 strict_slashes

**Purpose**: Whether to enforce strict slashes on endpoints.
**Default**: `False`

#### 12.2.40 url_prefix

**Purpose**: The url prefix used for all URL endpoints. Note, the placement of `/`.
**Default**: `'/auth'`

#### 12.2.41 user_id

**Purpose**: The key or property of your user object that contains a user id.
**Default**: `'user_id'`

#### 12.2.42 verify_exp

**Purpose**: Whether or not to check the expiration on an access token.
**Default**: `True`

> **Warning:** **IMPORTANT**: Changing verify_exp to `False` means that access tokens will **NOT** expire. Make sure you know what you are doing before disabling this.
>
> This is a potential **SECURITY RISK**.

CHAPTER 13

# Examples

CHAPTER 14

# Contributing

## 14.1 Shout out to `@vltr`

A special shout out to @vltr for all his work on Sanic JWT. This project received a nice kick and was taken up a notch due to his generous contribution of time, and skill.

## 14.2 Want to help?

Truly, this project has been a labor of love. We want to empower you to have an easy experience implementing best practices for a secure web application, yet still make it possible to be highly in control.

Do you think we did something wrong? Tell us

Have a better way? We're listening

If you want to get involved, here's how.

CHAPTER 15

# Changelog

The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

## 15.1 Version 1.2.1 - 2018-12-04

**Fixed**
- #143. Security bug resolved on empty tokens

## 15.2 Version 1.2.0 - 2018-08-06

**Added**
- Custom claims
- Extra payload validation
- Configuration option: `SANIC_JWT_DO_PROTECTION`

**Changed**
- Invalid tokens now `401` instead of `403`

## 15.3 Version 1.1.4 - 2018-08-06

**Fixed**
- Bug with `_do_protect` in `@scoped` decorator

## 15.4 Version 1.1.3 - 2018-08-06

**Changed**
- Exception handling to consistently have a `exception` and `reasons` key
- `reasons` in exception handling to be consistently formatted
- `400` responses for `debug` turned off, and `401` when turned on

**Fixed**
- #110. Preflight methods now properly handled
- #114. Proper use of `utils.call` to allow for sync and async `retrieve_user` functions
- #116. Proper error reporting on malformed tokens
- #118. Proper error reporting on expired token for `/auth/me` and `/auth/refresh` by applying `@protected` decorators

## 15.5 Version 1.1.2 - 2018-06-18

**Added**
- Ability to send authorization tokens via query string parameters

## 15.6 Version 1.1.1 - 2018-06-14

**Changed**
- Method of passing rquest object `args` and `kwargs` to scope handler

## 15.7 Version 1.1 - 2018-06-03

**Added**
- New handler method: `override_scope_validator`
- New handler method: `destructure_scopes`
- New decorator method: `inject_user`
- Decorator methods copied to `Initialize` class for convenience
- New convenience method for extracting `user_id` from request
- Feature for decoupling authentication mode for microservices
- Ability to have custom generated refresh tokens
- Subclasses are tested for consistency on `Initialize`

**Changed**
- `Authentication.is_authenticated` to `Authentication._check_authentication`
- `Authentication.verify` to `Authentication._verify`
- `Authentication.get_access_token` to `Authentication.generate_access_token`
- `Authentication.get_refresh_token` to `Authentication.generate_refresh_token`
- `Authentication.retrieve_scopes` to `Authentication.extract_scopes`
- Method for getting and setting configurations made dynamic

**Fixed**
- Verification that a custom payload extender supplies all of the enabled claims
- `abort` bug when using Sanic's convenience method for exceptions

## 15.8 Version 1.0.2 - 2018-03-04

**Fixed**
- Typo in docs for refresh token page
- Custom endpoints passing parameters to `BaseEndpoint`

## 15.9 Version 1.0.1 - 2018-02-27

**Added**
- `OPTIONS` handler method for `BaseEndpoint`

**Fixed**
- Some tests for claims that were not using UTC timestamps
- Consistency of docs with `class_views`

## 15.10 Version 1.0.0 - 2018-02-25

**Added**
- `Initialize` class
- New methods for adding configuration settings
- Customizable components
- Customizable responses
- Ability to fallback to header based authentication if cookie based fails
- Initialize on a blueprint and isolate configuration

**Fixed**
- `@protected` implementation on class based views
- Usage of signing algorithms with public and private keys

**Deprecated**
- `SANIC_JWT_PAYLOAD_HANDLER`
- `SANIC_JWT_HANDLER_PAYLOAD_EXTEND`
- `SANIC_JWT_HANDLER_PAYLOAD_SCOPES`

## 15.11 Legend

- **Added** for new features.
- **Changed** for changes in existing functionality.
- **Deprecated** for once-stable features removed in upcoming releases.
- **Removed** for deprecated features removed in this release.
- **Fixed** for any bug fixes.
- **Security** to invite users to upgrade in case of vulnerabilities.

---

# CHAPTER 16

## What is new in Version 1.2?

**Version 1.2** saw a a few minor fixes and changes. Some new features were added to allow custom claims and additional payload verifications. **Version 1.1.4** is still stable, so there is no need to upgrade to **1.2**.

In addition, there was a change that will (unfortunately) have some potential impact on users. Expired and invalid tokens will report as `HTTP 401`.

CHAPTER 17

# What is new in Version 1.1?

The biggest changes are under the hood relating to how configuration settings are implemented. They are now fully dynamic allowing you to not only dynamically set them at run time, but also have them evaluated at the last minute to give you flexibility when needed.

Flexibility is really the name of the game for v. 1.1. Most of the features are to enable the developer that wants to dig deeper and gain more control. For example, the `Authentication` now has a number of new renamed methods. Checkout the source code to see what they are (hint: they are the ones NOT with an _ at the beginning.)

Checkout the changelog for a more detailed description.

**Note:** It is recommended that you are at least using **version 1.1.2**. You can upgrade to **version 1.1.4** for improved exception handling and some bug fixes.

# CHAPTER 18

## What is new in Version 1.0?

If you have been using Sanic JWT, there should really not be that much different, although under the hood **a lot** has changed. For starters, the `initialize` method still works. But, the new recommended way to start Sanic JWT is to use the new `Initialize` class as seen above.

Using this class allows you to subclass it and really dive deep into modifying and configuring your project just the way you need it. Want to change the authentication responses? No problem. Want to add some new authentication endpoints? Easy.

One of the bigger changes is that we have enabled a new way to add configuration settings. You can of course continue to set them as recommended by Sanic by making them in all capital letters, and giving it a `SANIC_JWT_` prefix.

```
app.config.SANIC_JWT_ACCESS_TOKEN_NAME = 'mytoken'
```

Or, you can simply pass your *configurations* into the `Initialize` class as keyword arguments.

```
Initialize(
    app,
    access_token_name='mytoken'
)
```

Do you need some more complicated logic, or control? Then perhaps you want to subclass the `Configuration` class.

```
class MyConfig(Configuration):
    access_token_name='mytoken'
    def get_refresh_token_name(self):
        return some_crazy_logic_to_get_token_name()

Initialize(
    app,
    configuration_class=MyConfig
)
```

The point is, with Version 1, we made the entire package extremely adaptable and extensible for you to get done what you need without making decisions for you.

Have fun, and happy coding.