Simple JWT Documentation

Release 4.7.2.post3+g1c7f005

David Sanders

Jul 07, 2021

Contents

A JSON Web Token authentication plugin for the Django REST Framework.

Simple JWT provides a JSON Web Token authentication backend for the Django REST Framework. It aims to cover the most common use cases of JWTs by offering a conservative set of default features. It also aims to be easily extensible in case a desired feature is not present.

CHAPTER 1

Acknowledgments

This project borrows code from the Django REST Framework as well as concepts from the implementation of another JSON web token library for the Django REST Framework, django-rest-framework-jwt. The licenses from both of those projects have been included in this repository in the "licenses" directory.

CHAPTER 2

Contents

2.1 Getting started

2.1.1 Requirements

- Python (3.7, 3.8, 3.9)
- Django (2.2, 3.1, 3.2)
- Django REST Framework (3.10, 3.11, 3.12)

These are the officially supported python and package versions. Other versions will probably work. You're free to modify the tox config and see what is possible.

2.1.2 Installation

Simple JWT can be installed with pip:

```
pip install djangorestframework-simplejwt
```

Then, your django project must be configured to use the library. In `settings.py`, add `rest_framework_simplejwt.authentication.JWTAuthentication` to the list of authentication classes:

```
REST_FRAMEWORK = {
    ...
    'DEFAULT_AUTHENTICATION_CLASSES': (
        ...
        'rest_framework_simplejwt.authentication.JWTAuthentication',
    )
    ...
}
```

Also, in your root `urls.py` file (or any other url config), include routes for Simple JWT's `TokenObtainPairView` and `TokenRefreshView` views:

```
from rest_framework_simplejwt.views import (
    TokenObtainPairView,
    TokenRefreshView,
)

urlpatterns = [
    ...
    path('api/token/', TokenObtainPairView.as_view(), name='token_obtain_pair'),
    path('api/token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),
    ...
]
```

You can also include a route for Simple JWT's `TokenVerifyView` if you wish to allow API users to verify HMAC-signed tokens without having access to your signing key:

If you wish to use localizations/translations, simply add `rest_framework_simplejwt` to `INSTALLED_APPS`.

```
INSTALLED_APPS = [
    ...
    'rest_framework_simplejwt',
    ...
]
```

2.1.3 Usage

To verify that Simple JWT is working, you can use curl to issue a couple of test requests:

```
curl \
  -X POST \
  -H "Content-Type: application/json" \
  -d '{"username": "davidattenborough", "password": "boatymcboatface"}' \
  http://localhost:8000/api/token/

...
{
  "access":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJ1c2VyX3BrIjoxLCJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiY29sZF9zdHVmZiI6IuKYgyIsImV4cCI6MTIzNDU2LCJqdGkiOi
↪NHlztMGER7UADHZJlxNG0WSi22a2KaYSfd1S-AuT7lU",
  "refresh":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJ1c2VyX3BrIjoxLCJ0b2tlbl90eXBlIjoicmVmcmVzaCIsImNvbGRfc3R1ZmYiOiLimIMiLCJleHAiOjIzNDU2NywianRpIjo
↪aEoAYkSJjoWH1boshQAaTkf8G3yn0kapko6HFRt7Rh4"
}
```

You can use the returned access token to prove authentication for a protected view:

```
curl \
  -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJ1c2VyX3BrIjoxLCJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiY29sZF9zdHVmZiI6IuKYgyIsImV4cCI6MTIzNDU2LCJqdGkiOi
↪NHlztMGER7UADHZJlxNG0WSi22a2KaYSfd1S-AuT7lU" \
  http://localhost:8000/api/some-protected-view/
```

When this short-lived access token expires, you can use the longer-lived refresh token to obtain another access token:

```
curl \
  -X POST \
  -H "Content-Type: application/json" \
  -d '{"refresh":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJ1c2VyX3BrIjoxLCJ0b2tlbl90eXBlIjoicmVmcmVzaCIsImNvbGRfc3R1ZmYiOiLimIMiLCJleHAiOjIzNDU2NywianRpIj
↪aEoAYkSJjoWH1boshQAaTkf8G3yn0kapko6HFRt7Rh4"}' \
  http://localhost:8000/api/token/refresh/

...
{"access":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJ1c2VyX3BrIjoxLCJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiY29sZF9zdHVmZiI6IuKYgyIsImV4cCI6MTIzNTY3LCJqdGkiOi
↪ekxRxgb9OKmHkfy-zs1Ro_xs1eMLXiR17dIDBVxeT-w"}
```

2.2 Settings

Some of Simple JWT's behavior can be customized through settings variables in `settings.py`:

```
# Django project settings.py

from datetime import timedelta

...

SIMPLE_JWT = {
    'ACCESS_TOKEN_LIFETIME': timedelta(minutes=5),
    'REFRESH_TOKEN_LIFETIME': timedelta(days=1),
    'ROTATE_REFRESH_TOKENS': False,
    'BLACKLIST_AFTER_ROTATION': True,
    'UPDATE_LAST_LOGIN': False,

    'ALGORITHM': 'HS256',
    'SIGNING_KEY': settings.SECRET_KEY,
    'VERIFYING_KEY': None,
    'AUDIENCE': None,
    'ISSUER': None,

    'AUTH_HEADER_TYPES': ('Bearer',),
    'AUTH_HEADER_NAME': 'HTTP_AUTHORIZATION',
    'USER_ID_FIELD': 'id',
    'USER_ID_CLAIM': 'user_id',
    'USER_AUTHENTICATION_RULE': 'rest_framework_simplejwt.authentication.default_user_
↪authentication_rule',

    'AUTH_TOKEN_CLASSES': ('rest_framework_simplejwt.tokens.AccessToken',),
    'TOKEN_TYPE_CLAIM': 'token_type',

    'JTI_CLAIM': 'jti',

    'SLIDING_TOKEN_REFRESH_EXP_CLAIM': 'refresh_exp',
    'SLIDING_TOKEN_LIFETIME': timedelta(minutes=5),
    'SLIDING_TOKEN_REFRESH_LIFETIME': timedelta(days=1),
}
```

Above, the default values for these settings are shown.

2.2.1 ACCESS_TOKEN_LIFETIME

A `datetime.timedelta` object which specifies how long access tokens are valid. This `timedelta` value is added to the current UTC time during token generation to obtain the token's default "exp" claim value.

2.2.2 REFRESH_TOKEN_LIFETIME

A `datetime.timedelta` object which specifies how long refresh tokens are valid. This `timedelta` value is added to the current UTC time during token generation to obtain the token's default "exp" claim value.

2.2.3 ROTATE_REFRESH_TOKENS

When set to `True`, if a refresh token is submitted to the `TokenRefreshView`, a new refresh token will be returned along with the new access token. This new refresh token will be supplied via a "refresh" key in the JSON response. New refresh tokens will have a renewed expiration time which is determined by adding the timedelta in the `REFRESH_TOKEN_LIFETIME` setting to the current time when the request is made. If the blacklist app is in use and the `BLACKLIST_AFTER_ROTATION` setting is set to `True`, refresh tokens submitted to the refresh view will be added to the blacklist.

2.2.4 BLACKLIST_AFTER_ROTATION

When set to `True`, causes refresh tokens submitted to the `TokenRefreshView` to be added to the blacklist if the blacklist app is in use and the `ROTATE_REFRESH_TOKENS` setting is set to `True`. You need to add `'rest_framework_simplejwt.token_blacklist',` to your `INSTALLED_APPS` in the settings file to use this setting.

Learn more about *Blacklist app*.

2.2.5 UPDATE_LAST_LOGIN

When set to `True`, last_login field in the auth_user table is updated upon login (TokenObtainPairView).

> Warning: Updating last_login will dramatically increase the number of database transactions. People abusing the views could slow the server and this could be a security vulnerability. If you really want this, throttle the endpoint with DRF at the very least.

2.2.6 ALGORITHM

The algorithm from the PyJWT library which will be used to perform signing/verification operations on tokens. To use symmetric HMAC signing and verification, the following algorithms may be used: `'HS256'`, `'HS384'`, `'HS512'`. When an HMAC algorithm is chosen, the `SIGNING_KEY` setting will be used as both the signing key and the verifying key. In that case, the `VERIFYING_KEY` setting will be ignored. To use asymmetric RSA signing and verification, the following algorithms may be used: `'RS256'`, `'RS384'`, `'RS512'`. When an RSA algorithm is chosen, the `SIGNING_KEY` setting must be set to a string that contains an RSA private key. Likewise, the `VERIFYING_KEY` setting must be set to a string that contains an RSA public key.

2.2.7 SIGNING_KEY

The signing key that is used to sign the content of generated tokens. For HMAC signing, this should be a random string with at least as many bits of data as is required by the signing protocol. For RSA signing, this should be a string that contains an RSA private key that is 2048 bits or longer. Since Simple JWT defaults to using 256-bit HMAC signing, the `SIGNING_KEY` setting defaults to the value of the `SECRET_KEY` setting for your django project. Although this is the most reasonable default that Simple JWT can provide, it is recommended that developers change this setting to a value that is independent from the django project secret key. This will make changing the signing key used for tokens easier in the event that it is compromised.

2.2.8 VERIFYING_KEY

The verifying key which is used to verify the content of generated tokens. If an HMAC algorithm has been specified by the `ALGORITHM` setting, the `VERIFYING_KEY` setting will be ignored and the value of the `SIGNING_KEY` setting will be used. If an RSA algorithm has been specified by the `ALGORITHM` setting, the `VERIFYING_KEY` setting must be set to a string that contains an RSA public key.

2.2.9 AUDIENCE

The audience claim to be included in generated tokens and/or validated in decoded tokens. When set to `None`, this field is excluded from tokens and is not validated.

2.2.10 ISSUER

The issuer claim to be included in generated tokens and/or validated in decoded tokens. When set to `None`, this field is excluded from tokens and is not validated.

2.2.11 AUTH_HEADER_TYPES

The authorization header type(s) that will be accepted for views that require authentication. For example, a value of `'Bearer'` means that views requiring authentication would look for a header with the following format: `Authorization: Bearer <token>`. This setting may also contain a list or tuple of possible header types (e.g. `('Bearer', 'JWT')`). If a list or tuple is used in this way, and authentication fails, the first item in the collection will be used to build the "WWW-Authenticate" header in the response.

2.2.12 AUTH_HEADER_NAME

The authorization header name to be used for authentication. The default is `HTTP_AUTHORIZATION` which will accept the `Authorization` header in the request. For example if you'd like to use `X_Access_Token` in the header of your requests please specify the `AUTH_HEADER_NAME` to be `HTTP_X_ACCESS_TOKEN` in your settings.

2.2.13 USER_ID_FIELD

The database field from the user model that will be included in generated tokens to identify users. It is recommended that the value of this setting specifies a field that does not normally change once its initial value is chosen. For example, specifying a "username" or "email" field would be a poor choice since an account's username or email might change depending on how account management in a given service is designed. This could allow a new account to be created with an old username while an existing token is still valid which uses that username as a user identifier.

2.2.14 USER_ID_CLAIM

The claim in generated tokens which will be used to store user identifiers. For example, a setting value of `'user_id'` would mean generated tokens include a "user_id" claim that contains the user's identifier.

2.2.15 USER_AUTHENTICATION_RULE

Callable to determine if the user is permitted to authenticate. This rule is applied after a valid token is processed. The user object is passed to the callable as an argument. The default rule is to check that the `is_active` flag is still `True`. The callable must return a boolean, `True` if authorized, `False` otherwise resulting in a 401 status code.

2.2.16 AUTH_TOKEN_CLASSES

A list of dot paths to classes that specify the types of token that are allowed to prove authentication. More about this in the "Token types" section below.

2.2.17 TOKEN_TYPE_CLAIM

The claim name that is used to store a token's type. More about this in the "Token types" section below.

2.2.18 JTI_CLAIM

The claim name that is used to store a token's unique identifier. This identifier is used to identify revoked tokens in the blacklist app. It may be necessary in some cases to use another claim besides the default "jti" claim to store such a value.

2.2.19 SLIDING_TOKEN_LIFETIME

A `datetime.timedelta` object which specifies how long sliding tokens are valid to prove authentication. This `timedelta` value is added to the current UTC time during token generation to obtain the token's default "exp" claim value. More about this in the "Sliding tokens" section below.

2.2.20 SLIDING_TOKEN_REFRESH_LIFETIME

A `datetime.timedelta` object which specifies how long sliding tokens are valid to be refreshed. This `timedelta` value is added to the current UTC time during token generation to obtain the token's default "exp" claim value. More about this in the "Sliding tokens" section below.

2.2.21 SLIDING_TOKEN_REFRESH_EXP_CLAIM

The claim name that is used to store the expiration time of a sliding token's refresh period. More about this in the "Sliding tokens" section below.

2.3 Customizing token claims

If you wish to customize the claims contained in web tokens which are generated by the `TokenObtainPairView` and `TokenObtainSlidingView` views, create a subclass for the desired view as well as a subclass for its corresponding serializer. Here's an example of how to customize the claims in tokens generated by the `TokenObtainPairView`:

```
from rest_framework_simplejwt.serializers import TokenObtainPairSerializer
from rest_framework_simplejwt.views import TokenObtainPairView

class MyTokenObtainPairSerializer(TokenObtainPairSerializer):
    @classmethod
    def get_token(cls, user):
        token = super().get_token(user)

        # Add custom claims
        token['name'] = user.name
        # ...

        return token

class MyTokenObtainPairView(TokenObtainPairView):
    serializer_class = MyTokenObtainPairSerializer
```

Note that the example above will cause the customized claims to be present in both refresh *and* access tokens which are generated by the view. This follows from the fact that the `get_token` method above produces the *refresh* token for the view, which is in turn used to generate the view's access token.

As with the standard token views, you'll also need to include a url route to your subclassed view.

2.4 Creating tokens manually

Sometimes, you may wish to manually create a token for a user. This could be done as follows:

```
from rest_framework_simplejwt.tokens import RefreshToken

def get_tokens_for_user(user):
    refresh = RefreshToken.for_user(user)

    return {
        'refresh': str(refresh),
        'access': str(refresh.access_token),
    }
```

The above function `get_tokens_for_user` will return the serialized representations of new refresh and access tokens for the given user. In general, a token for any subclass of `rest_framework_simplejwt.tokens.Token` can be created in this way.

2.5 Token types

Simple JWT provides two different token types that can be used to prove authentication. In a token's payload, its type can be identified by the value of its token type claim, which is "token_type" by default. This may have a value of

"access", "sliding", or "refresh" however refresh tokens are not considered valid for authentication at this time. The claim name used to store the type can be customized by changing the `TOKEN_TYPE_CLAIM` setting.

By default, Simple JWT expects an "access" token to prove authentication. The allowed auth token types are determined by the value of the `AUTH_TOKEN_CLASSES` setting. This setting contains a list of dot paths to token classes. It includes the `'rest_framework_simplejwt.tokens.AccessToken'` dot path by default but may also include the `'rest_framework_simplejwt.tokens.SlidingToken'` dot path. Either or both of those dot paths may be present in the list of auth token classes. If they are both present, then both of those token types may be used to prove authentication.

2.5.1 Sliding tokens

Sliding tokens offer a more convenient experience to users of tokens with the trade-offs of being less secure and, in the case that the blacklist app is being used, less performant. A sliding token is one which contains both an expiration claim and a refresh expiration claim. As long as the timestamp in a sliding token's expiration claim has not passed, it can be used to prove authentication. Additionally, as long as the timestamp in its refresh expiration claim has not passed, it may also be submitted to a refresh view to get another copy of itself with a renewed expiration claim.

If you want to use sliding tokens, change the `AUTH_TOKEN_CLASSES` setting to `('rest_framework_simplejwt.tokens.SlidingToken',)`. (Alternatively, the `AUTH_TOKEN_CLASSES` setting may include dot paths to both the `AccessToken` and `SlidingToken` token classes in the `rest_framework_simplejwt.tokens` module if you want to allow both token types to be used for authentication.)

Also, include urls for the sliding token specific `TokenObtainSlidingView` and `TokenRefreshSlidingView` views alongside or in place of urls for the access token specific `TokenObtainPairView` and `TokenRefreshView` views:

```
from rest_framework_simplejwt.views import (
    TokenObtainSlidingView,
    TokenRefreshSlidingView,
)

urlpatterns = [
    ...
    path('api/token/', TokenObtainSlidingView.as_view(), name='token_obtain'),
    path('api/token/refresh/', TokenRefreshSlidingView.as_view(), name='token_refresh
↪'),
    ...
]
```

Be aware that, if you are using the blacklist app, Simple JWT will validate all sliding tokens against the blacklist for each authenticated request. This will reduce the performance of authenticated API views.

2.6 Blacklist app

Simple JWT includes an app that provides token blacklist functionality. To use this app, include it in your list of installed apps in `settings.py`:

```
# Django project settings.py

...

INSTALLED_APPS = (
```

(continues on next page)

(continued from previous page)

```
    ...
    'rest_framework_simplejwt.token_blacklist',
    ...
)
```

Also, make sure to run `python manage.py migrate` to run the app's migrations.

If the blacklist app is detected in `INSTALLED_APPS`, Simple JWT will add any generated refresh or sliding tokens to a list of outstanding tokens. It will also check that any refresh or sliding token does not appear in a blacklist of tokens before it considers it as valid.

The Simple JWT blacklist app implements its outstanding and blacklisted token lists using two models: `OutstandingToken` and `BlacklistedToken`. Model admins are defined for both of these models. To add a token to the blacklist, find its corresponding `OutstandingToken` record in the admin and use the admin again to create a `BlacklistedToken` record that points to the `OutstandingToken` record.

Alternatively, you can blacklist a token by creating a `BlacklistMixin` subclass instance and calling the instance's `blacklist` method:

```
from rest_framework_simplejwt.tokens import RefreshToken

token = RefreshToken(base64_encoded_token_string)
token.blacklist()
```

This will create unique outstanding token and blacklist records for the token's "jti" claim or whichever claim is specified by the `JTI_CLAIM` setting.

The blacklist app also provides a management command, `flushexpiredtokens`, which will delete any tokens from the outstanding list and blacklist that have expired. You should set up a cron job on your server or hosting platform which runs this command daily.

2.7 Experimental features

2.7.1 JWTTokenUserAuthentication backend

The `JWTTokenUserAuthentication` backend's `authenticate` method does not perform a database lookup to obtain a user instance. Instead, it returns a `rest_framework_simplejwt.models.TokenUser` instance which acts as a stateless user object backed only by a validated token instead of a record in a database. This can facilitate developing single sign-on functionality between separately hosted Django apps which all share the same token secret key. To use this feature, add the `rest_framework_simplejwt.authentication.JWTTokenUserAuthentication` backend (instead of the default `JWTAuthentication` backend) to the Django REST Framework's `DEFAULT_AUTHENTICATION_CLASSES` config setting:

```
REST_FRAMEWORK = {
    ...
    'DEFAULT_AUTHENTICATION_CLASSES': (
        ...
        'rest_framework_simplejwt.authentication.JWTTokenUserAuthentication',
    )
    ...
}
```

2.8 Development and contributing

To do development work for Simple JWT, make your own fork on Github, clone it locally, make and activate a virtualenv for it, then from within the project directory:

```
pip install --upgrade pip setuptools
pip install -e .[dev]
```

If you're running a Mac and/or with zsh, you need to escape the brackets:

```
pip install -e .\[dev\]
```

To run the tests:

```
pytest
```

To run the tests in all supported environments with tox, first install pyenv. Next, install the relevant Python minor versions and create a `.python-version` file in the project directory:

```
pyenv install 3.9.x
pyenv install 3.8.x
cat > .python-version <<EOF
3.9.x
3.8.x
EOF
```

Above, the `x` in each case should be replaced with the latest corresponding patch version. The `.python-version` file will tell pyenv and tox that you're testing against multiple versions of Python. Next, run tox:

```
tox
```

2.9 `drf-yasg` Integration

drf-yasg is a library that automatically generates an OpenAPI schema by inspecting DRF `Serializer` definitions. Because `django-rest-framework-simplejwt` serializers are not symmetric, if you want to generate correct OpenAPI schemas for your JWT token endpoints, use the following code to decorate your JWT `View` definitions.

```
from drf_yasg.utils import swagger_auto_schema
from rest_framework import serializers, status
from rest_framework_simplejwt.views import (
    TokenObtainPairView, TokenRefreshView, TokenVerifyView)


class TokenObtainPairResponseSerializer(serializers.Serializer):
    access = serializers.CharField()
    refresh = serializers.CharField()

    def create(self, validated_data):
        raise NotImplementedError()

    def update(self, instance, validated_data):
        raise NotImplementedError()
```

(continues on next page)

(continued from previous page)

```
class DecoratedTokenObtainPairView(TokenObtainPairView):
    @swagger_auto_schema(
        responses={
            status.HTTP_200_OK: TokenObtainPairResponseSerializer})
    def post(self, request, *args, **kwargs):
        return super().post(request, *args, **kwargs)


class TokenRefreshResponseSerializer(serializers.Serializer):
    access = serializers.CharField()

    def create(self, validated_data):
        raise NotImplementedError()

    def update(self, instance, validated_data):
        raise NotImplementedError()


class DecoratedTokenRefreshView(TokenRefreshView):
    @swagger_auto_schema(
        responses={
            status.HTTP_200_OK: TokenRefreshResponseSerializer})
    def post(self, request, *args, **kwargs):
        return super().post(request, *args, **kwargs)


class TokenVerifyResponseSerializer(serializers.Serializer):
    def create(self, validated_data):
        raise NotImplementedError()

    def update(self, instance, validated_data):
        raise NotImplementedError()


class DecoratedTokenVerifyView(TokenVerifyView):
    @swagger_auto_schema(
        responses={
            status.HTTP_200_OK: TokenVerifyResponseSerializer})
    def post(self, request, *args, **kwargs):
        return super().post(request, *args, **kwargs)
```

2.10 rest_framework_simplejwt package

2.10.1 Submodules

2.10.2 rest_framework_simplejwt.authentication module

class rest_framework_simplejwt.authentication.**JWTAuthentication**(**args*, ***kwargs*)

Bases: rest_framework.authentication.BaseAuthentication

An authentication plugin that authenticates requests through a JSON web token provided in a request header.

authenticate(*request*)

Authenticate the request and return a two-tuple of (user, token).

authenticate_header(*request*)
Return a string to be used as the value of the *WWW-Authenticate* header in a *401 Unauthenticated* response, or *None* if the authentication scheme should return *403 Permission Denied* responses.

get_header(*request*)
Extracts the header containing the JSON web token from the given request.

get_raw_token(*header*)
Extracts an unvalidated JSON web token from the given "Authorization" header value.

get_user(*validated_token*)
Attempts to find and return a user using the given validated token.

get_validated_token(*raw_token*)
Validates an encoded JSON web token and returns a validated token wrapper object.

media_type = 'application/json'

www_authenticate_realm = 'api'

class rest_framework_simplejwt.authentication.**JWTTokenUserAuthentication**(**args*, ***kwargs*)
Bases: *rest_framework_simplejwt.authentication.JWTAuthentication*

get_user(*validated_token*)
Returns a stateless user object which is backed by the given validated token.

rest_framework_simplejwt.authentication.**default_user_authentication_rule**(*user*)

2.10.3 rest_framework_simplejwt.models module

class rest_framework_simplejwt.models.**TokenUser**(*token*)
Bases: object

A dummy user class modeled after django.contrib.auth.models.AnonymousUser. Used in conjunction with the *JWTTokenUserAuthentication* backend to implement single sign-on functionality across services which share the same secret key. *JWTTokenUserAuthentication* will return an instance of this class instead of a *User* model instance. Instances of this class act as stateless user objects which are backed by validated tokens.

check_password(*raw_password*)

delete()

get_all_permissions(*obj=None*)

get_group_permissions(*obj=None*)

get_username()

groups

has_module_perms(*module*)

has_perm(*perm*, *obj=None*)

has_perms(*perm_list*, *obj=None*)

id

is_active = True

is_anonymous

is_authenticated

is_staff

is_superuser

pk

save()

set_password(*raw_password*)

user_permissions

username

2.10.4 rest_framework_simplejwt.serializers module

class rest_framework_simplejwt.serializers.**PasswordField**(**args*, ***kwargs*)
Bases: rest_framework.fields.CharField

class rest_framework_simplejwt.serializers.**TokenObtainPairSerializer**(**args*, ***kwargs*)
Bases: *rest_framework_simplejwt.serializers.TokenObtainSerializer*

classmethod get_token(*user*)

validate(*attrs*)

class rest_framework_simplejwt.serializers.**TokenObtainSerializer**(**args*, ***kwargs*)
Bases: rest_framework.serializers.Serializer

default_error_messages = {'no_active_account': 'No active account found with the give

classmethod get_token(*user*)

username_field = 'username'

validate(*attrs*)

class rest_framework_simplejwt.serializers.**TokenObtainSlidingSerializer**(**args*, ***kwargs*)
Bases: *rest_framework_simplejwt.serializers.TokenObtainSerializer*

classmethod get_token(*user*)

validate(*attrs*)

class rest_framework_simplejwt.serializers.**TokenRefreshSerializer**(*instance=None*, *data=<class 'rest_framework.fields.empty'>*, ***kwargs*)
Bases: rest_framework.serializers.Serializer

validate(*attrs*)

class rest_framework_simplejwt.serializers.**TokenRefreshSlidingSerializer**(*instance=None*, *data=<class 'rest_framework.fields.em*
***kwargs*)
Bases: rest_framework.serializers.Serializer

validate(*attrs*)

class rest_framework_simplejwt.serializers.**TokenVerifySerializer**(*instance=None*, *data=<class 'rest_framework.fields.empty'>*, ***kwargs*)

Bases: rest_framework.serializers.Serializer

validate(*attrs*)

2.10.5 rest_framework_simplejwt.tokens module

class rest_framework_simplejwt.tokens.**AccessToken**(*token=None*, *verify=True*)

Bases: *rest_framework_simplejwt.tokens.Token*

lifetime = datetime.timedelta(seconds=300)

token_type = 'access'

class rest_framework_simplejwt.tokens.**BlacklistMixin**

Bases: object

If the *rest_framework_simplejwt.token_blacklist* app was configured to be used, tokens created from *BlacklistMixin* subclasses will insert themselves into an outstanding token list and also check for their membership in a token blacklist.

blacklist()

Ensures this token is included in the outstanding token list and adds it to the blacklist.

check_blacklist()

Checks if this token is present in the token blacklist. Raises *TokenError* if so.

classmethod for_user(*user*)

Adds this token to the outstanding token list.

verify(**args*, ***kwargs*)

class rest_framework_simplejwt.tokens.**RefreshToken**(*token=None*, *verify=True*)

Bases: *rest_framework_simplejwt.tokens.BlacklistMixin*, *rest_framework_simplejwt.tokens.Token*

access_token

Returns an access token created from this refresh token. Copies all claims present in this refresh token to the new access token except those claims listed in the *no_copy_claims* attribute.

lifetime = datetime.timedelta(days=1)

no_copy_claims = ('token_type', 'exp', 'jti', 'jti')

token_type = 'refresh'

class rest_framework_simplejwt.tokens.**SlidingToken**(**args*, ***kwargs*)

Bases: *rest_framework_simplejwt.tokens.BlacklistMixin*, *rest_framework_simplejwt.tokens.Token*

lifetime = datetime.timedelta(seconds=300)

token_type = 'sliding'

class rest_framework_simplejwt.tokens.**Token**(*token=None*, *verify=True*)

Bases: object

A class which validates and wraps an existing JWT or can be used to build a new JWT.

check_exp(*claim='exp'*, *current_time=None*)
Checks whether a timestamp value in the given claim has passed (since the given datetime value in *current_time*). Raises a TokenError with a user-facing error message if so.

classmethod for_user(*user*)
Returns an authorization token for the given user that will be provided after authenticating the user's credentials.

get(*key*, *default=None*)

get_token_backend()

lifetime = None

set_exp(*claim='exp'*, *from_time=None*, *lifetime=None*)
Updates the expiration time of a token.

set_jti()
Populates the configured jti claim of a token with a string where there is a negligible probability that the same string will be chosen at a later time.

See here: https://tools.ietf.org/html/rfc7519#section-4.1.7

token_type = None

verify()
Performs additional validation steps which were not performed when this token was decoded. This method is part of the "public" API to indicate the intention that it may be overridden in subclasses.

verify_token_type()
Ensures that the token type claim is present and has the correct value.

class rest_framework_simplejwt.tokens.**UntypedToken**(*token=None*, *verify=True*)
Bases: *rest_framework_simplejwt.tokens.Token*

lifetime = datetime.timedelta(0)

token_type = 'untyped'

verify_token_type()
Untyped tokens do not verify the "token_type" claim. This is useful when performing general validation of a token's signature and other properties which do not relate to the token's intended use.

2.10.6 rest_framework_simplejwt.utils module

rest_framework_simplejwt.utils.**aware_utcnow**()

rest_framework_simplejwt.utils.**datetime_from_epoch**(*ts*)

rest_framework_simplejwt.utils.**datetime_to_epoch**(*dt*)

rest_framework_simplejwt.utils.**format_lazy**(*s*, **args*, ***kwargs*)

rest_framework_simplejwt.utils.**make_utc**(*dt*)

2.10.7 rest_framework_simplejwt.views module

class rest_framework_simplejwt.views.**TokenObtainPairView**(***kwargs*)
Bases: *rest_framework_simplejwt.views.TokenViewBase*

Takes a set of user credentials and returns an access and refresh JSON web token pair to prove the authentication of those credentials.

serializer_class
alias of *rest_framework_simplejwt.serializers.TokenObtainPairSerializer*

class rest_framework_simplejwt.views.**TokenObtainSlidingView**(***kwargs*)
Bases: *rest_framework_simplejwt.views.TokenViewBase*

Takes a set of user credentials and returns a sliding JSON web token to prove the authentication of those credentials.

serializer_class
alias of *rest_framework_simplejwt.serializers.TokenObtainSlidingSerializer*

class rest_framework_simplejwt.views.**TokenRefreshSlidingView**(***kwargs*)
Bases: *rest_framework_simplejwt.views.TokenViewBase*

Takes a sliding JSON web token and returns a new, refreshed version if the token's refresh period has not expired.

serializer_class
alias of *rest_framework_simplejwt.serializers.TokenRefreshSlidingSerializer*

class rest_framework_simplejwt.views.**TokenRefreshView**(***kwargs*)
Bases: *rest_framework_simplejwt.views.TokenViewBase*

Takes a refresh type JSON web token and returns an access type JSON web token if the refresh token is valid.

serializer_class
alias of *rest_framework_simplejwt.serializers.TokenRefreshSerializer*

class rest_framework_simplejwt.views.**TokenVerifyView**(***kwargs*)
Bases: *rest_framework_simplejwt.views.TokenViewBase*

Takes a token and indicates if it is valid. This view provides no information about a token's fitness for a particular use.

serializer_class
alias of *rest_framework_simplejwt.serializers.TokenVerifySerializer*

class rest_framework_simplejwt.views.**TokenViewBase**(***kwargs*)
Bases: rest_framework.generics.GenericAPIView

authentication_classes = ()

get_authenticate_header(*request*)
If a request is unauthenticated, determine the WWW-Authenticate header to use for 401 responses, if any.

permission_classes = ()

post(*request*, **args*, ***kwargs*)

serializer_class = None

www_authenticate_realm = 'api'

rest_framework_simplejwt.views.**token_obtain_pair**(*self*, *request*, **args*, ***kwargs*)
Takes a set of user credentials and returns an access and refresh JSON web token pair to prove the authentication of those credentials.

rest_framework_simplejwt.views.**token_obtain_sliding**(*self*, *request*, **args*, ***kwargs*)
Takes a set of user credentials and returns a sliding JSON web token to prove the authentication of those credentials.

rest_framework_simplejwt.views.**token_refresh**(*self*, *request*, **args*, ***kwargs*)
Takes a refresh type JSON web token and returns an access type JSON web token if the refresh token is valid.

rest_framework_simplejwt.views.**token_refresh_sliding**(*self*, *request*, **args*, ***kwargs*)
Takes a sliding JSON web token and returns a new, refreshed version if the token's refresh period has not expired.

rest_framework_simplejwt.views.**token_verify**(*self*, *request*, **args*, ***kwargs*)
Takes a token and indicates if it is valid. This view provides no information about a token's fitness for a particular use.

2.10.8 Module contents

CHAPTER 3

Indices and tables

- genindex
- modindex

Python Module Index

r

Index

T

U

V

W