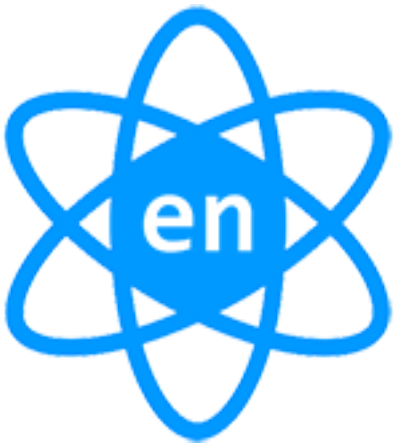
EverNode

Release 1.6.1

Sep 03, 2021

Contents



Welcome to EverNode's documentation. Get started with *Installation* and then get an overview with the *Getting Started*. EverNode folders and file patterns are described in the *Folder Structure* section. The rest of the docs describe each component of EverNode in detail, with a full reference in the *API* section.

CHAPTER 1

User's Guide

This part of the documentation, which is mostly prose, begins with some background information about EverNode, then focuses on step-by-step instructions for web development with EverNode.

1.1 Foreword

Read this before you get started with EverNode. This will hopefully answer some questions about the purpose and goals of the project, and when you should or should not be using it.

1.1.1 What is EverNode?

EverNode adds features to Flask to make API development easier. EverNode does not modify Flask directly, but just adds nice features to help build APIs faster. EverNode is built with stateless data transfer in mind and does not use sessions.

Highlights of the project include:

- Modular Design
- Sateless Data Transfer
- Easy Authentication with JWT tokens
- RESTful approach
- JSON first approach
- Easy Internationalization with Language Packs
- CORs support
- And much more

1.1.2 Configuration and Conventions

EverNode has many configuration values that can be easily set in the config.json file. To learn more about the EverNode and configuration visit the *Getting Started* section. When building with EverNode you should follow a few conventions, please refer to *Folder Structure*

1.2 Installation

EverNode requries a web server and Python3.6. It assumes you already have Python3.6 installed.

1.2.1 Python Version

We currently only support Python 3.6 and above. Our application replies on certain features only available in python 3.6, such as the secrets module added to core in 3.6.

Python 3.6 & pip

Debian and Ubuntu

```
wget https://www.python.org/ftp/python/3.6.5/Python-3.6.5.tar.xz && tar xvf Python-3.
↪6.5.tar.xz && cd Python-3.6.5 \
&& ./configure --enable-optimizations --with-ensurepip=install && make -j8 && make␣
↪altinstall
```

optional, update default python to python 3.6.

```
update-alternatives --install /usr/bin/python python /usr/local/bin/python3.6 50
```

1.2.2 Install EverNode

Within the activated environment, use the following command to install EverNode:

```
pip install evernode
```

1.2.3 Install NGINX

```
sudo apt-get install nginx nginx-extras
```

1.2.4 Install uWSGI

```
sudo pip install uwsgi
```

1.3 Getting Started

Ready to get started? This page gives a good introduction to EverNode. It assumes you already have EverNode, Python3.6, uWSGI and NGINX installed. If you do not, head over to the *Installation* section.

1.3.1 Minimal Application

The minimal EverNode application looks something like this:

```
from evernode.classes import App

# --- @boot ---
evernode_app = App(__name__)  # this is the EverNode app
app = evernode_app.app  # this is the Flask app
# --- @boot ---


# uWSGI entry point
if __name__ == '__main__':
    app.run()
```

1.3.2 EverNode Console

New App

You can easily create a new EverNode application by using the `evernode init` command via command line.

```
$ evernode init <app-name>
```

Once the files have been downloaded they will be in a evernode_<app-name> folder, relative to where the command was run. It's optional to download the docker files and the mock module.

1. Update the database connection string in the config file. `evernode_<app-name>/app/config.json`

   ```
   "SQLALCHEMY_BINDS": {
       "DEFAULT": "mysql://<db_user>:<password>@<host>/<db_name>"
   }
   ```

   Note:

   - EverNode can run without a database, but cannot use JWT sessions / DB Models.
   - You can change `mysql` to your odbc database connector.

2. Process models and Migrate the database. Navigate to `evernode_<app-name>/app`. Run the following commands in the terminal:

   ```
   $ flask db init
   $ flask db migrate
   $ flask db upgrade
   ```

3. If you downloaded the Docker files, you can run `docker-compose up --build` in the `evernode_<app-name>/docker` directory.

   - Please add the host below to your HOSTS file:

   ```
   127.0.0.1           api.localhost
   ```

4. If you downloaded the Mock Module, once the docker image has started you can navigate to `https://api.localhost/v1/hello-world`.

New Module

You can easily create new EverNode modules by using the `evernode module init` command via command line.

WARNING: Make sure to navigate to the `app/modules` folder. This command will make a folder relative to your command line `cd` location.

```
$ evernode module init <module-name>
```

1.3.3 Config

This section covers the configuration of EverNode. Applications need some kind of configuration. There are different settings you might want to change depending on the application environment like toggling the debug mode, setting the secret key, and other such environment-specific things.

Overview

Example | *config.json*

```
{
  "NAME": "<app-name>",
  "DEBUG": true,
  "SERECT": "<generate-fernet-key>",
  "KEY": "<generate-fernet-key>",
  "DEFAULT_LANGUAGE": "en",
  "DATETIME": {
    "TIMEZONE": "UTC",
    "DATE_FORMAT": "%Y-%m-%d",
    "TIME_FORMAT": "%H:%M:%S",
    "SEPARATOR": " "
  },
  "API": {
    "VERSION": "1",
    "PREFIX": "v{v}"
  },
  "UPLOADS": {
    "FOLDER": "/srv/uploads",
    "EXTENSIONS": [
      "png",
      "jpg"
    ]
  },
  "CORS": {
    "ALLOW_HEADERS": [
      "Origin",
      "Content-Type",
      "Accept",
      "Authorization",
      "X-Request-With",
      "Content-Language"
    ]
  },
  "EMAIL": {
    "HOST": "<stmp.example.com>",
    "PORT": 587,
```

(continues on next page)

(continued from previous page)

```
    "EMAIL": "<example@atomhash.org>",
    "NAME": "<email-sending-name>",
    "AUTH": "true",
    "TRANSPORT": "tls",
    "USERNAME": "<example@atomhash.org>",
    "PASSWORD": "<password>"
  },
  "AUTH": {
    "JWT": {
      "TOKENS": {
        "VALID_FOR": 7200
      },
      "REFRESH_TOKENS": {
        "ENABLED": false,
        "VALID_FOR": 86400
      }
    },
    "FAST_SESSIONS": false,
    "MAX_SESSIONS": 3,
    "USERNAME_FIELD": "<http-body-form-username-field-name>",
    "PASSWORD_FIELD": "<http-body-form-password-field-name>",
    "PASSWORD_HASHING": "pbkdf2:sha512"
  },
  "MAX_CONTENT_LENGTH": 2000000,
  "SQLALCHEMY_TRACK_MODIFICATIONS": false,
  "SQLALCHEMY_ECHO": true,
  "SQLALCHEMY_POOL_SIZE": 100,
  "SQLALCHEMY_POOL_RECYCLE": 280,
  "SQLALCHEMY_BINDS": {
    "DEFAULT": "mysql://api_user:password@ip/api"
  }
}
```

Ambiguous Types

DATE_FORMAT is strftime python format
TIME_FORMAT is strftime python format
MAX_CONTENT_LENGTH(FLASK) is in bytes
JWT TOKENS -> VALID_FOR is in seconds
JWT REFRESH_TOKENS -> VALID_FOR is in seconds

Debug Values

The following settings should be used in a development enviroment:

```
{
  "DEBUG": true,
  "SQLALCHEMY_TRACK_MODIFICATIONS": false,
  "SQLALCHEMY_ECHO": true,
}
```

Production Values

The following settings are values best suited for a production enviroment:

```
{
  "DEBUG": false,
  "SQLALCHEMY_TRACK_MODIFICATIONS": false,
  "SQLALCHEMY_ECHO": false,
}
```

1.3.4 uWSGI

This section will cover how to setup EverNode with uWSGI.

uwsgi.ini

Example | *uwsgi.ini*

```
[uwsgi]
uid=www-data
gid=www-data
chdir=/srv/app
pythonpath=/srv/app/
wsgi-file=/srv/app/app.py
callable=app
master=true
processes=4
threads=2
socket=/run/uwsgi/uwsgi.sock
chmod-socket=664
max-requests=5000
py-autoreload=1
logto = /srv/logs/%n.log
ignore-sigpipe=true
ignore-write-errors=true
disable-write-exception=true
```

- `wsgi-file=/srv/app/app.py` set the absolute path to your evernode app.py file.
- `callable=app` app is the variable that Flask is running as in your uwsgi-file.
- `pythonpath=/srv/app/` set this to your root application folder of the evernode_app.
- `pythonpath=/srv/app/` set chdir of uwsgi to root application path

Learn more about uWSGI configuration: http://uwsgi-docs.readthedocs.io/en/latest/Configuration.html.

1.3.5 NGINX

This section covers a basic nginx conf to start hosting your API.

Virtual Host File

Example | /etc/nginx/conf.d/*[website-domain].conf*

```
server {
    listen 80;
    listen 443 ssl;
    server_name [website-domain];
    ssl_certificate     ssls/[website-domain].crt;
    ssl_certificate_key ssls/[website-domain].key;
    root /srv/public;

    location / {
        include uwsgi_params;
        uwsgi_pass unix:///run/uwsgi/uwsgi.sock;
        uwsgi_read_timeout 1800;
        uwsgi_send_timeout 1800;
    }

    location ~ /\.ht {
        deny all;
    }
}
```

Replace `[website-domain]` with your domain name.

Learn more about NGINX configuration: http://nginx.org/en/docs/beginners_guide.html.

Generate Self-Signed Certificate

```
openssl req -new -sha256 -x509 -newkey rsa:4096 \
-nodes -keyout [website-domain].key -out [website-domain].crt -days 365
```

Replace `[website-domain]` with your domain name.

Generate Signing Request Certificate

```
openssl req -new -sha256 -newkey rsa:4096 \
-nodes -keyout [website-domain].key -out [website-domain].csr -days 365
```

Replace `[website-domain]` with your domain name.

1.4 Folder Structure

This section will cover how to structure your EverNode application.

1.4.1 Overview Structure

Your EverNode app should look like this:

```
evernode_<app-name>/
    app/
        modules/
            <module-name>/
```

(continues on next page)

(continued from previous page)

```
                controllers/
                    __init__.py
                    <ctrl-name>_controller.py
                    <ctrl-name>_ctrl.py  # or ctrl for short
                models/
                    __init__.py
                    <model-name>_model.py
                    <model-name>_mdl.py  # or mdl for short
                resources/
                    lang/
                        en/
                            <file-name>.lang
                    templates/
                        ...
                __init__.py
                routes.py
        resources/
            lang/
                en/
                    http_messages.lang
            templates/
                emails/
                    ...
                ...
        config.json
        app.py
    docker/
    logs/
    public/
        static/
    uploads/
    uwsgi.ini
```

1.4.2 Root Structure

Your EverNode root should look like this:

```
evernode_<app-name>/
    app/
    docker/
    logs/
    public/
        static/
    uploads/
    uwsgi.ini
```

1.4.3 Module Structure

Your EverNode module should look like this:

```
<module-name>/
    controllers/
        __init__.py
```

(continues on next page)

(continued from previous page)

```
        <ctrl-name>_controller.py
        <ctrl-name>_ctrl.py  # or ctrl for short
    models/
        __init__.py
        <model-name>_model.py
        <model-name>_mdl.py  # or mdl for short
    resources/
        lang/
            en/
                <file-name>.lang
        templates/
            ...
    __init__.py
    routes.py
```

1.5 Modules

EverNode uses modular design to build out APIs. How you use modules are entirely up to you. This means that after following the set folder and file names listed below, everything else is up to you. You can make one giant module for the whole API or many smaller modules ect...

However, our rule of thumb is a module should have one purpose.

Example: So if you have Users for your application, make a `evernode_<app-name>/app/modules/users` module. This module will handle User logins, signups, password resets, profiles and most things User Related.

1.5.1 Creating a Module

All modules for EverNode belong in the `evernode_<app-name>/app/modules` folder.

Basic Structure

Required folder and file names are as following:

```
<module-name>/
    controllers/
        __init__.py
        <ctrl-name>_controller.py
        <ctrl-name>_ctrl.py  # or ctrl for short
    models/
        __init__.py
        <model-name>_model.py
        <model-name>_mdl.py  # or mdl for short
    resources/
        lang/
            en/
                <file-name>.lang
        templates/
            ...
    __init__.py
    routes.py
```

Basic Routing

There are no standards for routing in EverNode. Just be smart about it. If a module is called `users`, name the routes users-<route> and the url will be /v1/users/<route>. If you follow this approach create a 'core'/'root' module to have all routes that start at /v1/<route>.

RESTful design, controllers should be used for singular objects(nouns). Each function/route will belong to different HTTP methods.

RESTful Example:

```
# modules/books/routes.py
from .controllers import BookController

routes = [
    {
        'url': '/books/<book_id>',
        'name': 'book-get',
        'methods': ['GET'],
        'function': BookController.get}]
# ------------------------------------------------------
# modules/books/models/book_model.py
from evernode.models import BaseModel

class BookModel(BaseModel):
""" Book DB Model """

    __tablename__ = 'books'

# ------------------------------------------------------
# modules/books/controllers/book_controller.py
from flask import current_app # noqa
from evernode.classes import JsonResponse
from ..models import BookModel

class BookController:
    """ RESTful example, BookController """

    @staticmethod
    def get(book_id):
        """ Get a book by id """
        return JsonResponse(200, None, BookModel.where_id(book_id))
```

1.6 Translator

This section will cover the basics of EverNode language translations.

Language translations are key to support internationalization. EverNode makes translations easy with its built in Translator class.

1.6.1 Translator Class

The Translator class uses a an index from the Content-Language HTTP header. If Content-Language is not set the DEFAULT_LANGUAGE set in your config.json will be used. The Translator class defaults to parse language packs from the root resoucres/lang folder.

Translator Class
class: *evernode.classes.Translator*

```
from evernode.classes import Translator

translator = Translator()
```

1.6.2 What are Language Packs?

Language Packs are files that end in `.lang` that contain valid json. These files can be parsed to translate certain messages from different languages.

Root Folder: Let's create a file called hello-world.lang.

English Example(resources/lang/en):

```
{
    "message": "Hello World"
}
```

French Example(resources/lang/fr):

```
{
    "message": "Bonjour le monde"
}
```

Module Folder: Let's create a file called hello-world.lang.

English Example(modules/<module-name>/resources/lang/en):

```
{
    "message": "Hello World"
}
```

French Example(modules/<module-name>/resources/lang/fr):

```
{
    "message": "Bonjour le monde"
}
```

You just created your first Language Pack. Now how do you use it? Simple.

1.6.3 How to use a Language Pack?

Root Folder: Init the Translator app:

```
from evernode.classes import Translator

translator = Translator()
print(translator.trans('hello-world.message'))
```

Output:

```
# Content-language: en
output: 'Hello World'

# Content-language: fr
output: 'Bonjour le monde'
```

Module Folder: Init the Translator app:

```
from evernode.classes import Translator

translator = Translator(module_name='<module-name>')
print(translator.trans('hello-world.message'))
```

Output:

```
# Content-language: en
output: 'Hello World'

# Content-language: fr
output: 'Bonjour le monde'
```

1.6.4 What Language is Used?

EverNode will choose the index set in Content-Language. If the Content-Language(HTTP header) is `fr`, lang/fr is used. If Content-Language is not set, your `DEFAULT_LANGUAGE` index is used. EverNode's default config uses `en`, lang/en will be used. Content-Language should use an ISO 639-1 code but can be anything.

1.7 Routes

EverNode requries a web server and Python3.6. It assumes you already have Python3.6 installed.

1.7.1 routes.py

Each module needs a routes.py file. A basic routes.py file can look like:

```
from .controllers import MockController

routes = [
    {
        'url': '/hello-world',
        'name': 'hello-world',
        'methods': ['GET'],  # POST, PUT, PATCH, UPDATE
        'function': MockController.hello_world}]
```

1.7.2 Authorization on Routes

If you would like to lock a route to a logged in user. An `Authorization: Bearer <token>` HTTP header must be supplied.

Example:

```
# modules/<module-name>/routes.py
from .controllers import MockController
from evernode.middleware import SessionMiddleware # noqa

routes = [
    {
        'url': '/hello-world',
        'name': 'hello-world',
        'methods': ['GET'],
        'middleware': [SessionMiddleware],  # returns a 401 response if not authorized
        'function': MockController.protected}]
# ------------------------------------------------------
# modules/<module-name>/controllers/mock_controller.py
from flask import current_app # noqa
from evernode.classes import JsonResponse, Render, Security, Email, UserAuth,␣
↪FormData, Translator # noqa
from evernode.decorators import middleware # noqa

class MockController:
    """ Mock Module, Mock Controller """

    @staticmethod
    @middleware  # this is required!
    def protected():
        """ Hello World Controller Protected """
        return JsonResponse(200, None, "Hello World, you're authorized!")
```

1.8 User Authentication

This section will cover the basics of user authentication in EverNode.

1.8.1 Simple Usage

Below is an example of how to use the UserAuth class. UserAuth has a method called `session()` which returns a JWT token that is stored into the database. The config.json has an option called `FAST_SESSIONS` which just turns off database validation. It is recommended in a production/ high security enviroment to turn this option to false. This protects users from sessions that have been hijacked, because if the session is removed from the database the session is no longer valid. The JWT token is encrypted by the `SERECT` string set in your config.json, then encrypted by your `KEY` string. For `FAST_SESSIONS` tokens to be valid it must be decrypted without error and not expired. The validity period is set in seconds by `JWT_EXP_SECS` config.json setting.

UserAuth Class

class: *evernode.classes.UserAuth*

```
from evernode.classes import UserAuth
from evernode.models import BaseUserModel

userAuth = UserAuth(
        BaseUserModel,
        username_error='Please enter your email',  # username empty
```

(continues on next page)

(continued from previous page)

```
        password_error='Please enter your password')  # password empty
    session = userAuth.session()
    if session is None:
        # return a 400 bad request HTTP status, password incorrect/username incorrect
        return JsonResponse(400)
    # return 200 successful HTTP status with a authorization token
    return JsonResponse(200, None, session)
```

CHAPTER 2

API Reference

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

2.1 API

The technical part of the documentation that covers what certain interfaces do within EverNode.

2.1.1 Classes

This part of the documentation covers all the classes of EverNode.

App

class evernode.classes.**App**(*name*, *prefix=None*, *middleware=None*, *root_path=None*, *config_path=None*)

Creates a Custom Flask App

prefix
Makes every URL require ending prefix. eg. prefix="v1", https://example.com/v1/

middleware
Set before middleware for all routes(environ, wsgi app)

root_path
Set a root path with an absolute system folder path

config_path
Set a config path(config.json) with an absolute system folder path

app
alias of `flask.app.Flask`

get_modules() → list
Get the module names(folders) in root modules folder

load_before_middleware(*before_middleware*)
Set before app middleware

load_config()
Load EverNode config.json

load_cors()
Default cors allow all

load_database()
Load default database, init flask-SQLAlchemy

load_language_files()
Load language fiels in resources/lang dirs

load_modules()
Load folders(custom modules) in modules folder

BaseResponse

class evernode.classes.**BaseResponse**(*status_code*, *message=None*, *data=None*, *environ=None*)
Base class for a HTTP response

content() → str
Get full content of response

data(*data=None*)
Set response data

message(*message=None*)
Set response message

mimetype() → str
Return private minetype

quick_response(*status_code*)
Quickly construct response using a status code

response
alias of `flask.wrappers.Response`

response_model
alias of `evernode.models.response_model.ResponseModel`

status(*status_code=None*)
Set status or Get Status

Cron

class evernode.classes.**Cron**

All you need to do is init Cron class: cron = Cron()

Then add tasks to schedule cron.schedule.every(1).seconds.do(test_job)

running()
Display running time and if Cron was enabled

schedule = <module 'schedule' from '/home/docs/checkouts/readthedocs.org/user_builds/e

Email

class evernode.classes.**Email**(*send_as_one=False*)
Set up an email

add_address(*address*)
Add email address

add_addresses(*addresses*)
Add addresses from array

add_cc(*address*)
Add carbon copy

add_ccs(*addresses*)
Add carbon copy with array of addresses

add_file(*absolute_file_path*)
Add attachment to email

encode(*value*)
Encode parts of email to base64 for transfer

html(*html*)
Set html content

send()
Construct and execute sendemail.py script Finds python binary by os.py, then uses the /usr/bin/python to execute email script

subject(*subject*)
Set email subject

text(*text*)
Set text content

FormData

class evernode.classes.**FormData**

add_field(*name*, *default=None*, *required=False*, *error=None*)
Add a text/non-file field to parse for a value in the request

add_file(*name*, *required=False*, *error=None*, *extensions=None*)
Add a file field to parse on request (uploads)

file_save(*name*, *filename=None*, *folder=''*, *keep_ext=True*) → bool
Easy save of a file

get_json_form_data()
Load request Json Data, if any

parse(*fail_callback=None*)
Parse text fields and file fields for values and files

Json

class evernode.classes.**Json**(*value*)
help break down and construct json objects

camel_case(*snake_case*)
Convert snake case to camel case

static from_file(*file_path*) → dict
Load JSON file

static parse(*string*, *is_file=False*, *obj=False*)
Convert a JSON string to dict/object

safe_object() → dict
Create an object ready for JSON serialization

safe_values(*value*)
Parse non-string values that will not serialize

static string(*value*) → str
string dict/object/value to JSON

JsonResponse

class evernode.classes.**JsonResponse**(*status_code=200*, *message=None*, *data=None*, *environ=None*)
JsonResponse is a wrapper for BaseResponse

content() → str
Get full content of response

data(*data=None*)
Set response data

message(*message=None*)
Set response message

mimetype() → str
Return private minetype

quick_response(*status_code*)
Quickly construct response using a status code

response
alias of flask.wrappers.Response

response_model
alias of evernode.models.response_model.ResponseModel

status(*status_code=None*)
Set status or Get Status

JWT

class evernode.classes.**JWT**
Gets request information to validate JWT

create_token(*data*, *token_valid_for=180*) → str
Create encrypted JWT

create_token_with_refresh_token(*data*, *token_valid_for=180*, *refresh_token_valid_for=86400*)
Create an encrypted JWT with a refresh_token

verify_http_auth_refresh_token() → bool
Use expired token to check refresh token information

verify_http_auth_token() → bool
Use request information to validate JWT

verify_refresh_token(*expired_token*) → bool
Use request information to validate refresh JWT

verify_token(*token*) → bool
Verify encrypted JWT

Middleware

class evernode.classes.**Middleware**
Provides useful defaults for creating fast middleware

app
alias of `flask.app.Flask`

condition() → bool
A condition to validate or invalidate a request

create_response()
Default response for if condition is false(invalid)

response
alias of `flask.wrappers.Response`

Render

class evernode.classes.**Render**(*module_name=None*)
Compile templates from root_path resources or module resources

compile(*name*, *folder=None*, *data=None*)
renders template_name + self.extension file with data using jinja

templates = {}
dict that contains compiled templates

Security

class evernode.classes.**Security**
Static functions to help app security

static decrypt(*crypt_text*) → str
Use config.json key to decrypt

static encrypt(*clear_text*) → str
Use config.json key to encrypt

static generate_key() → str
Generate a Fernet key

static hash(*clear_text*) → str
Hash clear text

static random_string(*length*) → str
Create a random string for security purposes

static verify_hash(*clear_text*, *hashed_text*) → bool
Check a hash

Session

class evernode.classes.**Session**
Helper class for app state-less sessions

classmethod create_session(*session_id*, *user_id*)
Save a new session to the database Using the ['AUTH']['MAX_SESSIONS'] config setting a session with be created within the MAX_SESSIONS limit. Once this limit is hit, delete the earliest session.

static create_session_id() → str
Create a session token

static current_session() → str
Return session id in app globals, only current request

static set_current_session(*session_id*) → bool
Add session_id to flask globals for current request

Singleton

class evernode.classes.**Singleton**

mro()
Return a type's method resolution order.

Translator

class evernode.classes.**Translator**(*module_name=None*, *environ=None*)
Uses dot-key syntax to translate phrases to words

trans(*key*) → str
Root Example: Translator() Translator.trans('messages.hello') resources/lang/en/messages.lang will be opened and parsed for { 'hello': 'Some english text' } If language is fr, resources/lang/fr/messages.lang will be opened and parsed for { 'hello': 'Some french text' } Module Example: Translator('[module-name]') Translator.trans('messages.hello')

UserAuth

class evernode.classes.**UserAuth**(*user_model*, *username_error=None*, *password_error=None*)
Helper class for creating user based authentication

session() → str
Generate a session(authorization Bearer) JWT token

verify_password(*hashed_password*) → bool
Check a password hash

2.1.2 Models

This part of the documentation covers all the models of EverNode.

BaseModel

class evernode.models.**BaseModel**
Adds usefull custom attributes for applciation use

add(*name*, *value*)
Add attribute

delete()
Easy delete for db models

exists()
Checks if item already exists in database

query_class
alias of flask_sqlalchemy.BaseQuery

remove(*name*)
Remove attribute

save()
Easy save(insert or update) for db models

updated()
Update updated_at timestamp

classmethod where_id(*id*)
Get db model by id

BaseUserModel

class evernode.models.**BaseUserModel**
User db model

add(*name*, *value*)
Add attribute

classmethod by_current_session()
Returns current user session

classmethod create_password_reset(*email*, *valid_for=3600*) → str
Create a password reset request in the user_password_resets database table. Hashed code gets stored in the database. Returns unhashed reset code

delete()
Easy delete for db models

exists()
Checks if item already exists in database

query_class
alias of flask_sqlalchemy.BaseQuery

remove(*name*)
Remove attribute

save()
Easy save(insert or update) for db models

set_password(*password*)
Set user password with hash

updated()
Update updated_at timestamp

classmethod validate_password_reset(*code*, *new_password*)
Validates an unhashed code against a hashed code. Once the code has been validated and confirmed new_password will replace the old users password

classmethod where_email(*email*)
Get db model by email

classmethod where_id(*id*)
Get db model by id

DatabaseModel

class evernode.models.**DatabaseModel**(**kwargs*)
Abstract class for db models

query_class
alias of `flask_sqlalchemy.BaseQuery`

JsonModel

class evernode.models.**JsonModel**
easy class to JSON conversion

add(*name*, *value*)
Add attribute

remove(*name*)
Remove attribute

PasswordResetModel

class evernode.models.**PasswordResetModel**
Password Reset db Model

add(*name*, *value*)
Add attribute

delete()
Easy delete for db models

classmethod delete_where_user_id(*user_id*)
delete by email

exists()
Checks if item already exists in database

query_class
alias of `flask_sqlalchemy.BaseQuery`

remove(*name*)
Remove attribute

save()
Easy save(insert or update) for db models

updated()
Update updated_at timestamp

classmethod where_code(*code*)
get by code

classmethod where_id(*id*)
Get db model by id

classmethod where_token(*token*)
get by token

classmethod where_user_id(*user_id*)
get by user_id

ResponseModel

class evernode.models.**ResponseModel**(*status*, *message*, *data=None*)
Class for a response a centralized response structure

SessionModel

class evernode.models.**SessionModel**
class to handle db model for session

add(*name*, *value*)
Add attribute

classmethod count(*user_id*)
Count sessions with user_id

delete()
Easy delete for db models

exists()
Checks if item already exists in database

query_class
alias of flask_sqlalchemy.BaseQuery

remove(*name*)
Remove attribute

save()
Easy save(insert or update) for db models

updated()
Update updated_at timestamp

classmethod where_earliest(*user_id*)
Get earilest session by created_at timestamp

classmethod where_id(*id*)
Get db model by id

classmethod where_lastest(*user_id*)
Get lastest session by created_at timestamp

classmethod where_session_id(*session_id*)
Easy way to query by session id

classmethod where_user_id(*user_id*)
Easy way to query by user id

2.1.3 Decorators

This part of the documentation covers all the decorators of EverNode.

@middleware

evernode.decorators.**middleware**(*func*)
Executes routes.py route middleware

2.1.4 Functions

This part of the documentation covers all the functions of EverNode.

get_subdirectories()

evernode.functions.**get_subdirectories**(*directory*)
Get subdirectories without pycache

get_python_path()

evernode.functions.**get_python_path**() → str
Accurately get python executable

2.1.5 Scripts

This part of the documentation covers all the scripts of EverNode.

SendEmail

class evernode.scripts.**SendEmail**
send email from the command line with async abilities

construct_message(*email=None*)
construct the email message

create_email()
main function to construct and send email

decode(*value*)
decode args

multipart
alias of email.mime.multipart.MIMEMultipart

parse()
parses args json

send(*email=None*)
send email message

smtp
alias of `smtplib.SMTP`

Python Module Index

e

Index

F

G

H

J

L

M

P

Q

R

S

T

U

V

W