Flask-Security Documentation

Release 5.3.3

Matt Wright
Chris Wagner

Dec 30, 2023

CONTENTS

Flask-Security

Flask-Security allows you to quickly add common security mechanisms to your Flask application. They include:

1. Session based authentication
2. Role and Permission management
3. Password hashing
4. Basic HTTP authentication
5. Token based authentication
6. Token based account activation (optional)
7. Token based password recovery / resetting (optional)
8. Two-factor authentication (optional)
9. Unified sign in (optional)
10. User registration (optional)
11. Login tracking (optional)
12. JSON/Ajax Support
13. WebAuthn Support (optional)
14. Use 'social'/Oauth for authentication (e.g. google, github, ..) (optional)

Many of these features are made possible by integrating various Flask extensions and libraries. They include:

- Flask-Login
- Flask-Mailman
- Flask-Principal
- Flask-WTF
- itsdangerous
- passlib
- QRCode
- webauthn
- authlib

Additionally, it assumes you'll be using a common library for your database connections and model definitions. Flask-Security supports the following Flask extensions out of the box for data persistence:

1. Flask-SQLAlchemy
2. MongoEngine

3. Peewee Flask utils
4. PonyORM - NOTE: not currently working - Help needed!.
5. SQLAlchemy sessions

CHAPTER

ONE

GETTING STARTED

1.1 Installation

Installing Flask-Security-Too using:

```
pip install flask-security-too
```

will install the basic package along with its required dependencies:

- Flask
- Flask-Login
- Flask-Principal
- Flask-WTF
- email-validator
- itsdangerous
- passlib
- Blinker

These are not sufficient for a complete application - other packages are required based on features desired, password hash algorithms, storage backend, etc. Flask-Security-Too has additional distribution 'extras' that can reduce the hassle of figuring out all the required packages. You can install these using the standard pip syntax:

```
pip install flask-security-too[extra1,extra2, ...]
```

Supported extras are:

- `babel` - Translation services. It will install babel and Flask-Babel.
- `fsqla` - Use flask-sqlalchemy and sqlalchemy as your storage interface.
- `common` - Install Flask-Mailman, bcrypt (the default password hash), and bleach.
- `mfa` - Install packages used for multi-factor (two-factor, unified signin, WebAuthn): cryptography, qrcode, phonenumberslite (note that for SMS you still need to pick an SMS provider and install appropriate packages), and webauthn.

Your application will also need a database backend:

- Sqlite is supported out of the box.
- For PostgreSQL install psycopg2.
- For MySQL install pymysql.

- For MongoDB install Mongoengine.

For additional details on configuring your database engine connector - refer to sqlalchemy_engine

1.2 Quick Start

There are some complete (but simple) examples available in the *examples* directory of the Flask-Security repo.

Note: The below quickstarts are just that - they don't enable most of the features (such as registration, reset, etc.). They basically create a single user, and you can login as that user… that's it. As you add more features, additional packages (e.g. Flask-Mailman, Flask-Babel, qrcode) might be required and will need to be added to your requirements.txt (or equivalent) file. Flask-Security does some configuration validation and will output error messages to the console for some missing packages.

Note: The default `SECURITY_PASSWORD_HASH` is "bcrypt" - so be sure to install bcrypt. If you opt for a different hash e.g. "argon2" you will need to install the appropriate package e.g. argon_cffi.

Danger: The examples below place secrets in source files. Never do this for your application especially if your source code is placed in a public repo. How you pass in secrets securely will depend on your deployment model - however in most cases (e.g. docker, lambda) using environment variables will be the easiest.

1.2.1 Basic SQLAlchemy Application

SQLAlchemy Install requirements

```
$ python3 -m venv pymyenv
$ . pymyenv/bin/activate
$ pip install flask-security-too[fsqla,common]
```

SQLAlchemy Application

The following code sample illustrates how to get started as quickly as possible using Flask-SQLAlchemy and the built-in model mixins:

```
import os

from flask import Flask, render_template_string
from flask_sqlalchemy import SQLAlchemy
from flask_security import Security, SQLAlchemyUserDatastore, auth_required, hash_
↪password
from flask_security.models import fsqla_v3 as fsqla

# Create app
app = Flask(__name__)
app.config['DEBUG'] = True

# Generate a nice key using secrets.token_urlsafe()
app.config['SECRET_KEY'] = os.environ.get("SECRET_KEY", 'pf9Wkove4IKEAXvy-cQkeDPhv9Cb3Ag-
↪wyJILbq_dFw')
# Bcrypt is set as default SECURITY_PASSWORD_HASH, which requires a salt
# Generate a good salt using: secrets.SystemRandom().getrandbits(128)
app.config['SECURITY_PASSWORD_SALT'] = os.environ.get("SECURITY_PASSWORD_SALT",
↪'146585145368132386173505678016728509634')

# have session and remember cookie be samesite (flask/flask_login)
app.config["REMEMBER_COOKIE_SAMESITE"] = "strict"
app.config["SESSION_COOKIE_SAMESITE"] = "strict"

# Use an in-memory db
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite://'
# As of Flask-SQLAlchemy 2.4.0 it is easy to pass in options directly to the
# underlying engine. This option makes sure that DB connections from the
# pool are still valid. Important for entire application since
# many DBaaS options automatically close idle connections.
app.config["SQLALCHEMY_ENGINE_OPTIONS"] = {
    "pool_pre_ping": True,
}
app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False

# Create database connection object
db = SQLAlchemy(app)

# Define models
fsqla.FsModels.set_db_info(db)

class Role(db.Model, fsqla.FsRoleMixin):
    pass

class User(db.Model, fsqla.FsUserMixin):
    pass

# Setup Flask-Security
```

(continues on next page)

(continued from previous page)

```
user_datastore = SQLAlchemyUserDatastore(db, User, Role)
app.security = Security(app, user_datastore)

# Views
@app.route("/")
@auth_required()
def home():
    return render_template_string("Hello {{ current_user.email }}")

# one time setup
with app.app_context():
    # Create User to test with
    db.create_all()
    if not app.security.datastore.find_user(email="test@me.com"):
        app.security.datastore.create_user(email="test@me.com", password=hash_password(
↪"password"))
    db.session.commit()

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

You can run this either with:

```
flask run
```

or:

```
python app.py
```

1.2.2 Basic SQLAlchemy Application with session

SQLAlchemy Install requirements

```
$ python3 -m venv pymyenv
$ . pymyenv/bin/activate
$ pip install flask-security-too[common] sqlalchemy
```

SQLAlchemy Application (w/o Flask-SQLAlchemy)

The following code sample illustrates how to get started as quickly as possible using SQLAlchemy in a declarative way:

This example shows how to split your application into 3 files: app.py, database.py and models.py.

- app.py

```
import os

from flask import Flask, render_template_string
from flask_security import Security, current_user, auth_required, hash_password, \
     SQLAlchemySessionUserDatastore, permissions_accepted
from database import db_session, init_db
```

(continues on next page)

(continued from previous page)

```python
from models import User, Role

# Create app
app = Flask(__name__)
app.config['DEBUG'] = True

# Generate a nice key using secrets.token_urlsafe()
app.config['SECRET_KEY'] = os.environ.get("SECRET_KEY", 'pf9Wkove4IKEAXvy-
↪cQkeDPhv9Cb3Ag-wyJILbq_dFw')
# Bcrypt is set as default SECURITY_PASSWORD_HASH, which requires a salt
# Generate a good salt using: secrets.SystemRandom().getrandbits(128)
app.config['SECURITY_PASSWORD_SALT'] = os.environ.get("SECURITY_PASSWORD_SALT",
↪'146585145368132386173505678016728509634')
# Don't worry if email has findable domain
app.config["SECURITY_EMAIL_VALIDATOR_ARGS"] = {"check_deliverability": False}

# manage sessions per request - make sure connections are closed and returned
app.teardown_appcontext(lambda exc: db_session.close())

# Setup Flask-Security
user_datastore = SQLAlchemySessionUserDatastore(db_session, User, Role)
app.security = Security(app, user_datastore)

# Views
@app.route("/")
@auth_required()
def home():
    return render_template_string('Hello {{current_user.email}}!')

@app.route("/user")
@auth_required()
@permissions_accepted("user-read")
def user_home():
    return render_template_string("Hello {{ current_user.email }} you are a user!")

# one time setup
with app.app_context():
    init_db()
    # Create a user and role to test with
    app.security.datastore.find_or_create_role(
        name="user", permissions={"user-read", "user-write"}
    )
    db_session.commit()
    if not app.security.datastore.find_user(email="test@me.com"):
        app.security.datastore.create_user(email="test@me.com",
        password=hash_password("password"), roles=["user"])
    db_session.commit()

if __name__ == '__main__':
    # run application (can also use flask run)
    app.run()
```

- database.py

```
from sqlalchemy import create_engine
from sqlalchemy.orm import scoped_session, sessionmaker
from sqlalchemy.ext.declarative import declarative_base

engine = create_engine('sqlite:////tmp/test.db')
db_session = scoped_session(sessionmaker(autocommit=False,
                                         autoflush=False,
                                         bind=engine))
Base = declarative_base()
Base.query = db_session.query_property()

def init_db():
    # import all modules here that might define models so that
    # they will be registered properly on the metadata.  Otherwise
    # you will have to import them first before calling init_db()
    import models
    Base.metadata.create_all(bind=engine)
```

- models.py

```
from database import Base
from flask_security import UserMixin, RoleMixin, AsaList
from sqlalchemy.orm import relationship, backref
from sqlalchemy.ext.mutable import MutableList
from sqlalchemy import Boolean, DateTime, Column, Integer, \
                    String, ForeignKey

class RolesUsers(Base):
    __tablename__ = 'roles_users'
    id = Column(Integer(), primary_key=True)
    user_id = Column('user_id', Integer(), ForeignKey('user.id'))
    role_id = Column('role_id', Integer(), ForeignKey('role.id'))

class Role(Base, RoleMixin):
    __tablename__ = 'role'
    id = Column(Integer(), primary_key=True)
    name = Column(String(80), unique=True)
    description = Column(String(255))
    permissions = Column(MutableList.as_mutable(AsaList()), nullable=True)

class User(Base, UserMixin):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    email = Column(String(255), unique=True)
    username = Column(String(255), unique=True, nullable=True)
    password = Column(String(255), nullable=False)
    last_login_at = Column(DateTime())
    current_login_at = Column(DateTime())
    last_login_ip = Column(String(100))
    current_login_ip = Column(String(100))
    login_count = Column(Integer)
    active = Column(Boolean())
    fs_uniquifier = Column(String(64), unique=True, nullable=False)
```

(continues on next page)

(continued from previous page)

```
    confirmed_at = Column(DateTime())
    roles = relationship('Role', secondary='roles_users',
                         backref=backref('users', lazy='dynamic'))
```

You can run this either with:

```
flask run
```

or:

```
python app.py
```

1.2.3 Basic MongoEngine Application

MongoEngine Install requirements

```
$ python3 -m venv pymyenv
$ . pymyenv/bin/activate
$ pip install flask-security-too[common] mongoengine
```

MongoEngine Application

The following code sample illustrates how to get started as quickly as possible using MongoEngine (of course you have to install and start up a local MongoDB instance):

```
import os

from flask import Flask, render_template_string
from mongoengine import Document, connect
from mongoengine.fields import (
    BinaryField,
    BooleanField,
    DateTimeField,
    IntField,
    ListField,
    ReferenceField,
    StringField,
)
from flask_security import Security, MongoEngineUserDatastore, \
    UserMixin, RoleMixin, auth_required, hash_password, permissions_accepted

# Create app
app = Flask(__name__)
app.config['DEBUG'] = True

# Generate a nice key using secrets.token_urlsafe()
app.config['SECRET_KEY'] = os.environ.get("SECRET_KEY", 'pf9Wkove4IKEAXvy-cQkeDPhv9Cb3Ag-
↪wyJILbq_dFw')
# Bcrypt is set as default SECURITY_PASSWORD_HASH, which requires a salt
# Generate a good salt using: secrets.SystemRandom().getrandbits(128)
```

(continues on next page)

(continued from previous page)

```python
app.config['SECURITY_PASSWORD_SALT'] = os.environ.get("SECURITY_PASSWORD_SALT",
↪'146585145368132386173505678016728509634')
# Don't worry if email has findable domain
app.config["SECURITY_EMAIL_VALIDATOR_ARGS"] = {"check_deliverability": False}

# Create database connection object
db_name = "mydatabase"
db = connect(alias=db_name, db=db_name, host="mongodb://localhost", port=27017)

class Role(Document, RoleMixin):
    name = StringField(max_length=80, unique=True)
    description = StringField(max_length=255)
    permissions = ListField(required=False)
    meta = {"db_alias": db_name}

class User(Document, UserMixin):
    email = StringField(max_length=255, unique=True)
    password = StringField(max_length=255)
    active = BooleanField(default=True)
    fs_uniquifier = StringField(max_length=64, unique=True)
    confirmed_at = DateTimeField()
    roles = ListField(ReferenceField(Role), default=[])
    meta = {"db_alias": db_name}

# Setup Flask-Security
user_datastore = MongoEngineUserDatastore(db, User, Role)
app.security = Security(app, user_datastore)

# Views
@app.route("/")
@auth_required()
def home():
    return render_template_string("Hello {{ current_user.email }}")

@app.route("/user")
@auth_required()
@permissions_accepted("user-read")
def user_home():
    return render_template_string("Hello {{ current_user.email }} you are a user!")

# one time setup
with app.app_context():
    # Create a user and role to test with
    app.security.datastore.find_or_create_role(
        name="user", permissions={"user-read", "user-write"}
    )
    if not app.security.datastore.find_user(email="test@me.com"):
        app.security.datastore.create_user(email="test@me.com",
        password=hash_password("password"), roles=["user"])

if __name__ == '__main__':
    # run application (can also use flask run)
```

(continues on next page)

(continued from previous page)

```
    app.run()
```

1.2.4 Basic Peewee Application

Peewee Install requirements

```
$ python3 -m venv pymyenv
$ . pymyenv/bin/activate
$ pip install flask-security-too[common] peewee
```

Peewee Application

The following code sample illustrates how to get started as quickly as possible using Peewee:

```
import os

from flask import Flask, render_template_string
from playhouse.flask_utils import FlaskDB
from peewee import *
from flask_security import Security, PeeweeUserDatastore, \
    UserMixin, RoleMixin, auth_required, hash_password

# Create app
app = Flask(__name__)
app.config['DEBUG'] = True

# Generate a nice key using secrets.token_urlsafe()
app.config['SECRET_KEY'] = os.environ.get("SECRET_KEY", 'pf9Wkove4IKEAXvy-cQkeDPhv9Cb3Ag-
↪wyJILbq_dFw')
# Bcrypt is set as default SECURITY_PASSWORD_HASH, which requires a salt
# Generate a good salt using: secrets.SystemRandom().getrandbits(128)
app.config['SECURITY_PASSWORD_SALT'] = os.environ.get("SECURITY_PASSWORD_SALT",
↪'146585145368132386173505678016728509634')

app.config['DATABASE'] = {
    'name': 'example.db',
    'engine': 'peewee.SqliteDatabase',
}

# Create database connection object
db = FlaskDB(app)

class Role(RoleMixin, db.Model):
    name = CharField(unique=True)
    description = TextField(null=True)
    permissions = TextField(null=True)

# N.B. order is important since db.Model also contains a get_id() -
# we need the one from UserMixin.
```

(continues on next page)

(continued from previous page)

```
class User(UserMixin, db.Model):
    email = TextField()
    password = TextField()
    active = BooleanField(default=True)
    fs_uniquifier = TextField(null=False)
    confirmed_at = DateTimeField(null=True)

class UserRoles(db.Model):
    # Because peewee does not come with built-in many-to-many
    # relationships, we need this intermediary class to link
    # user to roles.
    user = ForeignKeyField(User, related_name='roles')
    role = ForeignKeyField(Role, related_name='users')
    name = property(lambda self: self.role.name)
    description = property(lambda self: self.role.description)

    def get_permissions(self):
        return self.role.get_permissions()

# Setup Flask-Security
user_datastore = PeeweeUserDatastore(db, User, Role, UserRoles)
app.security = Security(app, user_datastore)

# Views
@app.route('/')
@auth_required()
def home():
    return render_template_string("Hello {{ current_user.email }}")

# one time setup
with app.app_context():
    # Create a user to test with
    for Model in (Role, User, UserRoles):
        Model.drop_table(fail_silently=True)
        Model.create_table(fail_silently=True)
    if not app.security.datastore.find_user(email="test@me.com"):
        app.security.datastore.create_user(email="test@me.com", password=hash_password(
↪"password"))

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

1.2.5 Mail Configuration

Flask-Security integrates with an outgoing mail service via the `mail_util_cls` which is part of initial configuration. The default class *flask_security.MailUtil* utilizes the Flask-Mailman package. Be sure to add flask_mailman to your requirements.txt. The older and no longer maintained package Flask-Mail is also (still) supported.

The following code illustrates a basic setup, which could be added to the basic application code in the previous section:

```
# At top of file
from flask_mailman import Mail

# After 'Create app'
app.config['MAIL_SERVER'] = 'smtp.example.com'
app.config['MAIL_PORT'] = 587
app.config['MAIL_USE_TLS'] = True
app.config['MAIL_USERNAME'] = 'username'
app.config['MAIL_PASSWORD'] = 'password'
mail = Mail(app)
```

To learn more about the various Flask-Mailman settings to configure it to work with your particular email server configuration, please see the Flask-Mailman documentation.

1.2.6 Proxy Configuration

The user tracking features need an additional configuration in HTTP proxy environment. The following code illustrates a setup with a single HTTP proxy in front of the web application:

```
# At top of file
from werkzeug.middleware.proxy_fix import ProxyFix

# After 'Create app'
app.wsgi_app = ProxyFix(app.wsgi_app, x_for=1)
```

To learn more about the `ProxyFix` middleware, please see the Werkzeug documentation.

1.2.7 Unit Testing Your Application

As soon as you add any of the Flask-Security decorators to your API endpoints, it can be frustrating to unit test your basic routing (and roles and permissions). Without getting into the argument of the difference between unit tests and integration tests - you can approach testing in 2 ways:

- 'Pure' unit test - mocking out all lower level objects (such as the data store)
- Complete app with in-memory/temporary DB (with little or no mocking).

Look in the Flask-Security repo *examples* directory for actual code that implements the second approach which is much simpler and with an in-memory DB fairly fast.

You also might want to set the following configurations in your conftest.py:

```
app.config["WTF_CSRF_ENABLED"] = False
# Our test emails/domain isn't necessarily valid
app.config["SECURITY_EMAIL_VALIDATOR_ARGS"] = {"check_deliverability": False}
# Make this plaintext for most tests - reduces unit test time by 50%
app.config["SECURITY_PASSWORD_HASH"] = "plaintext"
```

1.3 Features

Flask-Security allows you to quickly add common security mechanisms to your Flask application. They include:

1.3.1 Session Based Authentication

Session based authentication is fulfilled entirely by the Flask-Login extension. Flask-Security handles the configuration of Flask-Login automatically based on a few of its own configuration values and uses Flask-Login's alternative token feature to associate the value of `fs_uniquifier` with the user. (This enables easily invalidating all existing sessions for a given user without having to change their user id). Flask-WTF integrates with the session as well to provide out of the box CSRF support. Flask-Security extends that to support configurations that would require CSRF for requests that are authenticated via session cookies, but not for requests authenticated using tokens.

1.3.2 Role/Identity Based Access

Flask-Security implements very basic role management out of the box. This means that you can associate a high level role or multiple roles to any user. For instance, you may assign roles such as *Admin*, *Editor*, *SuperUser*, or a combination of said roles to a user. Access control is based on the role name and/or permissions contained within the role; and all roles should be uniquely named. This feature is implemented using the Flask-Principal extension. As with basic RBAC, permissions can be assigned to roles to provide more granular access control. Permissions can be associated with one or more roles (the RoleModel contains a list of permissions). The values of permissions are completely up to the developer - Flask-Security simply treats them as strings. If you'd like to implement even more granular access control (such as per-object), you can refer to the Flask-Principal documentation on this topic.

1.3.3 Password Hashing

Password hashing is enabled with passlib. Passwords are hashed with the bcrypt function by default but you can easily configure the hashing algorithm. You should **always use a hashing algorithm** in your production environment. Hash algorithms not listed in `SECURITY_PASSWORD_SINGLE_HASH` will be double hashed - first an HMAC will be computed, then the selected hash function will be used. In this case - you must provide a `SECURITY_PASSWORD_SALT`. A good way to generate this is:

```
secrets.SystemRandom().getrandbits(128)
```

Bear in mind passlib does not assume which algorithm you will choose and may require additional libraries to be installed.

1.3.4 Password Validation and Complexity

Consult *Password Validation and Complexity*.

1.3.5 Basic HTTP Authentication

Basic HTTP authentication is achievable using a simple view method decorator. This feature expects the incoming authentication information to identify a user in the system. This means that the username must be equal to their email address.

1.3.6 Token Authentication

Token based authentication can be used by retrieving the user auth token from an authentication endpoint (e.g. `/login`, `/us-signin`). Perform an HTTP POST with a query param of `include_auth_token` and the authentication details as JSON data. A successful call will return the authentication token. This token can be used in subsequent requests to protected resources. The auth token should be supplied in the request through an HTTP header or query string parameter. By default the HTTP header name is *Authentication-Token* and the default query string parameter name is *auth_token*.

Authentication tokens are generated using a uniquifier field in the user's UserModel. By default that field is `fs_uniquifier`. This means that if that field is changed (via `UserDatastore.set_uniquifier()`) then any existing authentication tokens will no longer be valid. This value is changed whenever a user changes their password. If this is not the desired behavior then you can add an additional attribute to the UserModel: `fs_token_uniquifier` and that will be used instead, thus isolating password changes from authentication tokens. That attribute can be changed via `UserDatastore.set_token_uniquifier()`. This attribute should have `unique=True`. Unlike `fs_uniquifier`, it can be set to `nullable` - it will automatically be generated at first use if null.

1.3.7 Two-factor Authentication

Two-factor authentication is enabled by generating time-based one time passwords (Tokens). The tokens are generated using the users totp secret, which is unique per user, and is generated both on first login, and when changing the two-factor method (doing this causes the previous totp secret to become invalid). The token is provided by one of 3 methods - email, sms (service is not provided), or an authenticator app such as Google Authenticator, LastPass Authenticator, or Authy. By default, tokens provided by the authenticator app are valid for 2 minutes, tokens sent by mail for up to 5 minute and tokens sent by sms for up to 2 minutes. The QR code used to supply the authenticator app with the secret is generated using the qrcode library. Please read *Theory of Operation* for more details.

The Two-factor feature offers the ability for a user to 'rescue' themselves if they lose track of their secondary factor device. Rescue options include sending a one time code via email, send an email to the application admin, and using a previously generated and downloaded one-time code (see `SECURITY_MULTI_FACTOR_RECOVERY_CODES`).

1.3.8 Unified Sign In

This feature is in Beta - mostly due to it being brand new and little to no production soak time

Unified sign in provides a generalized login endpoint that takes an *identity* and a *passcode*; where (based on configuration):

- *identity* is any of `SECURITY_USER_IDENTITY_ATTRIBUTES` (e.g. email, username, phone)
- *passcode* is a password or a one-time code (delivered via email, SMS, or authenticator app)

Please see this Wikipedia article about multi-factor authentication.

Using this feature, it is possible to not require the user to have a stored password at all, and just require the use of a one-time code. The mechanisms for generating and delivering the one-time code are similar to common two-factor mechanisms.

This one-time code can be configured to be delivered via email, SMS or authenticator app - however be aware that NIST does not recommend email for this purpose (though many web sites do so) due to the fact that a) email may travel through many different servers as part of being delivered - and b) is available from any device.

Using SMS or an authenticator app means you are providing "something you have" (the mobile device) and either "something you know" (passcode to unlock your device) or "something you are" (biometric quality to unlock your device). This effectively means that using a one-time code to sign in, is in fact already two-factor (if using SMS or authenticator app). Many large authentication providers already offer this - here is Microsoft's version.

Note that by configuring *SECURITY_US_ENABLED_METHODS* an application can use this endpoint JUST with identity/password or in fact disallow passwords altogether.

Unified sign in is integrated with two-factor authentication. Since in general there is no need for a second factor if the initial authentication was with SMS or an authenticator application, the *SECURITY_US_MFA_REQUIRED* configuration determines which primary authentication mechanisms require a second factor. By default limited to `email` and `password` (if two-factor is enabled).

Be aware that by default, the *SECURITY_US_SETUP_URL* endpoint is protected with a freshness check (see *flask_security.auth_required()*) which means it requires a session cookie to function properly. This is true even if using JSON payload or token authentication. If you disable the freshness check then sessions aren't required.

Current Limited Functionality:

- Change password does not work if a user registers without a password. However forgot-password will allow the user to set a new password.
- Registration and Confirmation only work with email - so while you can enable multiple authentication methods, you still have to register with email.

1.3.9 WebAuthn

This feature is in Beta - mostly due to it being brand new and little to no production soak time

WebAuthn is a standardized protocol that connects authenticators (such as YubiKey and mobile biometrics) with websites. Flask-Security supports using WebAuthn keys as either 'first' or 'secondary' authenticators. Please read *WebAuthn* for more details.

1.3.10 Email Confirmation

If desired you can require that new users confirm their email address. Flask-Security will send an email message to any new users with a confirmation link. Upon navigating to the confirmation link, the user will be automatically logged in. There is also view for resending a confirmation link to a given email if the user happens to try to use an expired token or has lost the previous email. Confirmation links can be configured to expire after a specified amount of time.

1.3.11 Password Reset/Recovery

Password reset and recovery is available for when a user forgets their password. Flask-Security sends an email to the user with a link to a view which allows them to reset their password. Once the password is reset they are redirected to the login page where they need to authenticate using the new password. Password reset links can be configured to expire after a specified amount of time.

As with password change - this will update the the user's `fs_uniquifier` attribute which will invalidate all existing sessions AND (by default) all authentication tokens.

1.3.12 User Registration

Flask-Security comes packaged with a basic user registration view. This view is very simple and new users need only supply an email address and their password. This view can be overridden if your registration process requires more fields. User email is validated and normalized using the email_validator package.

The *SECURITY_USERNAME_ENABLE* configuration option, when set to `True`, will add support for the user to register a username in addition to an email. By default, the user will be able to authenticate with EITHER email or username - however that can be changed via the *SECURITY_USER_IDENTITY_ATTRIBUTES*.

1.3.13 Password Change

Flask-Security comes packaged with a basic change user password view. Unlike password recovery, this endpoint is used when the user is already authenticated. The result of a successful password change is not only a new password, but a new value for `fs_uniquifier`. This has the effect is immediately invalidating all existing sessions. The change request itself effectively re-logs in the user so a new session is created. Note that since the user is effectively re-logged in, the same signals are sent as when the user normally authenticates.

NOTE: The `fs_uniquifier` by default, controls both sessions and authenticated tokens. Thus changing the password also invalidates all authentication tokens. This may not be desirable behavior, so if the UserModel contains an attribute `fs_token_uniquifier`, then that will be used when generating authentication tokens and so won't be affected by password changes.

1.3.14 Login Tracking

Flask-Security can, if configured, keep track of basic login events and statistics. They include:

- Last login date
- Current login date
- Last login IP address
- Current login IP address
- Total login count

1.3.15 JSON/Ajax Support

Flask-Security supports JSON/Ajax requests where appropriate. Please look at *CSRF* for details on how to work with JSON and Single Page Applications. More specifically JSON is supported for the following operations:

- Login requests
- Unified sign in requests
- Registration requests
- Change password requests
- Confirmation requests
- Forgot password requests
- Passwordless login requests
- Two-factor login requests
- Change two-factor method requests

- WebAuthn registration and signin requests
- Two-Factor recovery code requests

In addition, Single-Page-Applications (like those built with Vue, Angular, and React) are supported via customizable redirect links.

Note: All registration requests done through JSON/Ajax utilize the `confirm_register_form`.

1.3.16 Command Line Interface

Basic Click commands for managing users and roles are automatically registered. They can be completely disabled or their names can be changed. Run `flask --help` and look for users and roles.

1.3.17 Social/Oauth Authentication

Flask-Security provides a thin layer which integrates authlib with Flask-Security views and features (such as two-factor authentication). Flask-Security is shipped with support for github and google - others can be added by the application (see loginpass for many examples).

See *`flask_security.OAuthGlue`*

Please note - this is for authentication only, and the authenticating user must already be a registered user in your application. Once authenticated, all further authorization uses Flask-Security role/permission mechanisms.

See Flask OAuth Client for details. Note in particular, that you must setup and provide provider specific information - and most importantly - XX_CLIENT_ID and XX_CLIENT_SECRET should be specified as environment variables.

A very simple example of configuring social auth with Flask-Security is available in the *examples* directory.

1.4 Configuration

The following configuration values are used by Flask-Security:

1.4.1 Core

These configuration keys are used globally across all features.

`SECRET_KEY`

This is actually part of Flask - but is used by Flask-Security to sign all tokens. It is critical this is set to a strong value. For python3 consider using: `secrets.token_urlsafe()`

`SECURITY_BLUEPRINT_NAME`

Specifies the name for the Flask-Security blueprint.

Default: `"security"`.

`SECURITY_URL_PREFIX`

Specifies the URL prefix for the Flask-Security blueprint.

Default: `None`.

SECURITY_STATIC_FOLDER

Specifies the folder name for static files (webauthn).

Default: "static".

New in version 5.1.0.

SECURITY_STATIC_FOLDER_URL

Specifies the URL for static files used by Flask-Security (webauthn). See Flask documentation https://flask.palletsprojects.com/en/latest/blueprints/#static-files

Default: "/fs-static".

New in version 5.1.0.

SECURITY_SUBDOMAIN

Specifies the subdomain for the Flask-Security blueprint. If your authenticated content is on a different subdomain, also enable *SECURITY_REDIRECT_ALLOW_SUBDOMAINS*.

Default: None.

SECURITY_FLASH_MESSAGES

Specifies whether or not to flash messages during security procedures.

Default: True.

SECURITY_I18N_DOMAIN

Specifies the name for domain used for translations.

Default: "flask_security".

SECURITY_I18N_DIRNAME

Specifies the directory containing the MO files used for translations. When using flask-babel this can also be a list of directory names - this enables application to override a subset of messages if desired. The default builtin uses translations shipped with Flask-Security.

Default: "builtin".

Changed in version 5.2.0: "builtin" is a special name which will be interpreted as the translations directory within the installation of Flask-Security.

SECURITY_PASSWORD_HASH

Specifies the password hash algorithm to use when hashing passwords. Recommended values for production systems are bcrypt, argon2, sha512_crypt, or pbkdf2_sha512. Some algorithms require the installation of a backend package (e.g. bcrypt, argon2).

Default: "bcrypt".

SECURITY_PASSWORD_SCHEMES

List of support password hash algorithms. SECURITY_PASSWORD_HASH must be from this list. Passwords encrypted with any of these schemes will be honored.

SECURITY_DEPRECATED_PASSWORD_SCHEMES

List of password hash algorithms that are considered weak and will be accepted, however on first use, will be re-hashed to the current setting of SECURITY_PASSWORD_HASH.

Default: ["auto"] which means any password found that wasn't hashed using SECURITY_PASSWORD_HASH will be re-hashed.

`SECURITY_PASSWORD_SALT`

Specifies the HMAC salt. This is required for all schemes that are configured for double hashing. A good salt can be generated using: `secrets.SystemRandom().getrandbits(128)`.

Default: `None`.

`SECURITY_PASSWORD_SINGLE_HASH`

A list of schemes that should not be hashed twice. By default, passwords are hashed twice, first with `SECURITY_PASSWORD_SALT`, and then with a random salt.

Default: a list of known schemes not working with double hashing (*django_{digest}*, *plaintext*).

`SECURITY_HASHING_SCHEMES`

List of algorithms used for encrypting/hashing sensitive data within a token (Such as is sent with confirmation or reset password).

Default: `["sha256_crypt", "hex_md5"]`.

`SECURITY_DEPRECATED_HASHING_SCHEMES`

List of deprecated algorithms used for creating and validating tokens.

Default: `["hex_md5"]`.

`SECURITY_PASSWORD_HASH_OPTIONS`

Specifies additional options to be passed to the hashing method. This is deprecated as of passlib 1.7.

Deprecated since version 3.4.0: see: *SECURITY_PASSWORD_HASH_PASSLIB_OPTIONS*

`SECURITY_PASSWORD_HASH_PASSLIB_OPTIONS`

Pass additional options to the various hashing methods. This is a dict of the form `{<scheme>__<option>: <value>, ..}` e.g. {“argon2__rounds”: 10}.

New in version 3.3.1.

`SECURITY_PASSWORD_LENGTH_MIN`

Minimum required length for passwords.

Default: `8`

New in version 3.4.0.

`SECURITY_PASSWORD_COMPLEXITY_CHECKER`

Set to complexity checker to use (Only `zxcvbn` supported).

Default: `None`

New in version 3.4.0.

`SECURITY_ZXCVBN_MINIMUM_SCORE`

Required `zxcvbn` password complexity score (0-4). Refer to https://github.com/dropbox/zxcvbn#usage for exact meanings of different score values.

Default: `3` (Good or Strong)

New in version 5.0.0.

`SECURITY_PASSWORD_CHECK_BREACHED`

If not `None` new/changed passwords will be checked against the database of breached passwords at https://api.pwnedpasswords.com. If set to `strict` then if the site can’t be reached, validation will fail. If set to `best-effort` failure to reach the site will continue with the rest of password validation.

Default: `None`

New in version 3.4.0.

`SECURITY_PASSWORD_BREACHED_COUNT`

Passwords with counts greater than or equal to this value are considered breached.

Default: 1 - which might be to burdensome for some applications.

New in version 3.4.0.

`SECURITY_PASSWORD_NORMALIZE_FORM`

Passwords are normalized prior to changing or comparing. This satisfies the NIST requirement: 5.1.1.2 Memorized Secret Verifiers. Normalization is performed using the Python unicodedata.normalize() method.

Default: `"NFKD"`

New in version 4.0.0.

`SECURITY_PASSWORD_REQUIRED`

If set to `False` then a user can register with an empty password. This requires *`SECURITY_UNIFIED_SIGNIN`* to be enabled. By default, the user will be able to authenticate using an email link. Please note: this does not mean a user can sign in with an empty password - it means that they must have some OTHER means of authenticating.

Default: `True`

New in version 5.0.0.

`SECURITY_TOKEN_AUTHENTICATION_KEY`

Specifies the query string parameter to read when using token authentication.

Default: `"auth_token"`.

`SECURITY_TOKEN_AUTHENTICATION_HEADER`

Specifies the HTTP header to read when using token authentication.

Default: `"Authentication-Token"`.

`SECURITY_TOKEN_MAX_AGE`

Specifies the number of seconds before an authentication token expires.

Default: `None`, meaning the token never expires.

`SECURITY_EMAIL_VALIDATOR_ARGS`

Email address are validated and normalized via the `mail_util_cls` which defaults to *`MailUtil`*. That uses the email_validator package whose methods have configurable options - these can be set here and will be passed in. For example setting this to: `{"check_deliverability":  False}` is useful when unit testing if the emails are fake.

`mail_util_cls` has 2 methods - `normalize` and `validate`. Both ensure the passed value is a valid email address, and returns a normalized version. `validate` additionally, by default, verifies that the email address can likely actually receive an email.

Default: `None`, meaning use the defaults from email_validator package.

New in version 4.0.0.

`SECURITY_DEFAULT_HTTP_AUTH_REALM`

Specifies the default authentication realm when using basic HTTP auth.

Default: `Login Required`

`SECURITY_REDIRECT_BEHAVIOR`

Passwordless login, confirmation, reset password, unified signin, and oauth signin have GET endpoints that validate the passed token and redirect to an action form. For Single-Page-Applications style UIs which need to control their own internal URL routing these redirects need to not contain forms, but contain relevant information as query parameters. Setting this to `spa` will enable that behavior.

Default: `None` which is existing html-style form redirects.

New in version 3.3.0.

`SECURITY_REDIRECT_HOST`

Mostly for development purposes, the UI is often developed separately and is running on a different port than the Flask application. In order to test redirects, the *netloc* of the redirect URL needs to be rewritten. Setting this to e.g. *localhost:8080* does that.

Default: `None`.

New in version 3.3.0.

`SECURITY_REDIRECT_ALLOW_SUBDOMAINS`

If `True` then subdomains (and the root domain) of the top-level host set by Flask's `SERVER_NAME` configuration will be allowed as post-view redirect targets. This is beneficial if you wish to place your authentiation on one subdomain and authenticated content on another, for example `auth.domain.tld` and `app.domain.tld`.

Default: `False`.

New in version 4.0.0.

`SECURITY_REDIRECT_VALIDATE_MODE`

Defines how Flask-Security will attempt to mitigate an open redirect vulnerability w.r.t. client supplied *next* parameters. Please see *Open Redirect Exposure* for a complete discussion.

Current options include *"absolute"* and *"regex"*. A list is allowed.

Default: `["absolute"]`

New in version 4.0.2.

Changed in version 5.3.3: Default is now *"absolute"* and now takes a list.

`SECURITY_REDIRECT_VALIDATE_RE`

This regex handles known patterns that can be exploited. Basically, don't allow control characters or white-space followed by slashes (or back slashes).

Default: `r"^/{4,}|\\{3,}|[\s\000-\037][/\\]{2,}(?![/\\])|[/\\]([^/\\]|/[^/\\])*[/\\].*"`

New in version 4.0.2.

`SECURITY_CSRF_PROTECT_MECHANISMS`

Authentication mechanisms that require CSRF protection. These are the same mechanisms as are permitted in the `@auth_required` decorator.

Default: `("basic", "session", "token")`.

`SECURITY_CSRF_IGNORE_UNAUTH_ENDPOINTS`

If `True` then CSRF will not be required for endpoints that don't require authentication (e.g. login, logout, register, forgot_password).

Default: `False`.

`SECURITY_CSRF_COOKIE_NAME`

The name for the CSRF cookie. This usually should be dictated by your client-side code - more information can be found at *CSRF*

Default: `None` - meaning no cookie will be sent.

SECURITY_CSRF_COOKIE

A dict that defines the parameters required to set a CSRF cookie. The complete set of parameters is described in Flask's set_cookie documentation.

Default: `{"samesite": "Strict", "httponly": False, "secure": False}`

Changed in version 4.1.0: The 'key' attribute was deprecated in favor of a separate configuration variable `SECURITY_CSRF_COOKIE_NAME`.

SECURITY_CSRF_HEADER

The HTTP Header name that will contain the CSRF token. `X-XSRF-Token` is used by packages such as axios.

Default: `"X-XSRF-Token"`.

SECURITY_CSRF_COOKIE_REFRESH_EACH_REQUEST

By default, csrf_tokens have an expiration (controlled by the configuration variable `WTF_CSRF_TIME_LIMIT`. This can cause CSRF failures if say an application is left idle for a long time. You can set that time limit to `None` or have the CSRF cookie sent on every request (which will give it a new expiration time).

Default: `False`.

SECURITY_EMAIL_SENDER

Specifies the email address to send emails as.

Default: value set to `MAIL_DEFAULT_SENDER` if Flask-Mail is used otherwise `no-reply@localhost`.

SECURITY_USER_IDENTITY_ATTRIBUTES

Specifies which attributes of the user object can be used for credential validation.

Defines the order and matching that will be applied when validating login credentials (either via standard login form or the unified sign in form). The identity field in the form will be matched in order using this configuration - the FIRST match will then be used to look up the user in the DB.

Mapping functions take a single argument - `identity` from the form and should return `None` if the `identity` argument isn't in a format suitable for the attribute. If the `identity` argument format matches, it should be returned, optionally having had some canonicalization performed. The returned result will be used to look up the identity in the UserDataStore using the column name specified in the key.

The provided *`flask_security.uia_phone_mapper()`* for example performs phone number normalization using the `phonenumbers` package.

Tip: If your mapper performs any sort of canonicalization/normalization, make sure you apply the exact same transformation in your form validator when setting the field.

Danger: Make sure that any attributes listed here are marked Unique in your UserDataStore model.

Danger: Make sure your mapper methods guard against malicious user input. For example, if you allow `username` as an identity method you could use bleach:

```python
def uia_username_mapper(identity):
    # we allow pretty much anything - but we bleach it.
    return bleach.clean(identity, strip=True)
```

Default:

```
[
    {"email": {"mapper": uia_email_mapper, "case_insensitive": True}},
]
```

If you enable *SECURITY_UNIFIED_SIGNIN* and set `sms` as a *SECURITY_US_ENABLED_METHODS* and your *SECURITY_USER_IDENTITY_ATTRIBUTES* contained:

```
[
    {"email": {"mapper": uia_email_mapper, "case_insensitive": True}},
    {"us_phone_number": {"mapper": uia_phone_mapper}},
]
```

Then after the user sets up their SMS - they could login using their phone number and get a text with the authentication code.

Changed in version 4.0.0: Changed from list to list of dict.

SECURITY_USER_IDENTITY_MAPPINGS

New in version 3.4.0.

Deprecated since version 4.0.0: Superseded by *SECURITY_USER_IDENTITY_ATTRIBUTES*

SECURITY_API_ENABLED_METHODS

Various endpoints of Flask-Security require the caller to be authenticated. This variable controls which of the methods - `token`, `session`, `basic` will be allowed. The default does NOT include `basic` since if `basic` is in the list, and if the user is NOT authenticated, then the standard/required response of 401 with the `WWW-Authenticate` header is returned. This is rarely what the client wants.

Default: `["session", "token"]`.

New in version 4.0.0.

SECURITY_DEFAULT_REMEMBER_ME

Specifies the default "remember me" value used when logging in a user.

Default: `False`.

SECURITY_RETURN_GENERIC_RESPONSES

If set to `True` Flask-Security will return generic responses to endpoints that could be used to enumerate users. Please see *Generic Responses - Avoiding User Enumeration*.

New in version 5.0.0.

SECURITY_BACKWARDS_COMPAT_UNAUTHN

If set to `True` then the default behavior for authentication failures from one of Flask-Security's decorators will be restored to be compatible with releases prior to 3.3.0 (return 401 and some static html).

Default: `False`.

SECURITY_BACKWARDS_COMPAT_AUTH_TOKEN

If set to `True` then an Authentication-Token will be returned on every successful call to login, reset-password, change-password as part of the JSON response. This was the default prior to release 3.3.0 - however sending Authentication-Tokens (which by default don't expire) to session based UIs is a bad security practice.

Default: `False`.

1.4.2 Core - Multi-factor

These are used by the Two-Factor and Unified Signin features.

SECURITY_TOTP_SECRETS

Secret used to encrypt the totp_password both into DB and into the session cookie. Best practice is to set this to:

```
from passlib import totp
"{1: <result of totp.generate_secret()>}"
```

See: Totp for details.

New in version 3.4.0.

SECURITY_TOTP_ISSUER

Specifies the name of the service or application that the user is authenticating to. This will be the name displayed by most authenticator apps.

Default: `None`.

New in version 3.4.0.

SECURITY_SMS_SERVICE

Specifies the name of the sms service provider. Out of the box "Twilio" is supported. For other sms service providers you will need to subclass *SmsSenderBaseClass* and register it:

```
SmsSenderFactory.senders[<service-name>] = <service-class>
```

Default: `Dummy` which does nothing.

New in version 3.4.0.

SECURITY_SMS_SERVICE_CONFIG

Specifies a dictionary of basic configurations needed for use of a sms service. For "Twilio" the following keys are required (fill in from your Twilio dashboard):

Default: `{'ACCOUNT_SID': NONE, 'AUTH_TOKEN': NONE, 'PHONE_NUMBER': NONE}`

New in version 3.4.0.

SECURITY_PHONE_REGION_DEFAULT

Assigns a default 'region' for phone numbers used for two-factor or unified sign in. All other phone numbers will require a region prefix to be accepted.

Default: `"US"`

New in version 3.4.0.

SECURITY_FRESHNESS

A timedelta used to protect endpoints that alter sensitive information. This is used to protect the following endpoints:

- *SECURITY_US_SETUP_URL*
- *SECURITY_TWO_FACTOR_SETUP_URL*
- *SECURITY_WAN_REGISTER_URL*
- *SECURITY_MULTI_FACTOR_RECOVERY_CODES*

Setting this to a negative number will disable any freshness checking and the endpoints:

- *SECURITY_VERIFY_URL*

- *SECURITY_US_VERIFY_URL*
- *SECURITY_US_VERIFY_SEND_CODE_URL*
- *SECURITY_WAN_VERIFY_URL*

won't be registered. Setting this to 0 results in undefined behavior. Please see *flask_security.check_and_update_authn_fresh()* for details.

Default: timedelta(hours=24)

New in version 3.4.0.

SECURITY_FRESHNESS_GRACE_PERIOD

A timedelta that provides a grace period when altering sensitive information. This is used to protect the endpoints:

- *SECURITY_US_SETUP_URL*
- *SECURITY_TWO_FACTOR_SETUP_URL*
- *SECURITY_WAN_REGISTER_URL*

N.B. To avoid strange behavior, be sure to set the grace period less than the freshness period. Please see *flask_security.check_and_update_authn_fresh()* for details.

Default: timedelta(hours=1)

New in version 3.4.0.

1.4.3 Core - rarely need changing

SECURITY_DATETIME_FACTORY

Specifies the default datetime factory.

Default:`datetime.datetime.utcnow`.

SECURITY_CONFIRM_SALT

Specifies the salt value when generating confirmation links/tokens.

Default: `"confirm-salt"`.

SECURITY_RESET_SALT

Specifies the salt value when generating password reset links/tokens.

Default: `"reset-salt"`.

SECURITY_LOGIN_SALT

Specifies the salt value when generating login links/tokens.

Default: `"login-salt"`.

SECURITY_REMEMBER_SALT

Specifies the salt value when generating remember tokens. Remember tokens are used instead of user ID's as it is more secure.

Default: `"remember-salt"`.

SECURITY_TWO_FACTOR_VALIDITY_SALT

Specifies the salt value when generating two factor validity tokens.

Default: `"tf-validity-salt"`.

SECURITY_US_SETUP_SALT

Default: "us-setup-salt"

SECURITY_WAN_SALT

Default: "wan-salt"

SECURITY_EMAIL_PLAINTEXT

Sends email as plaintext using `*.txt` template.

Default: `True`.

SECURITY_EMAIL_HTML

Sends email as HTML using `*.html` template.

Default: `True`.

SECURITY_CLI_USERS_NAME

Specifies the name for the command managing users. Disable by setting `False`.

Default: `"users"`.

SECURITY_CLI_ROLES_NAME

Specifies the name for the command managing roles. Disable by setting `False`.

Default: `"roles"`.

SECURITY_JOIN_USER_ROLES

Specifies whether to set the `UserModel.roles` loading relationship to `joined` when a `roles` attribute is present for a SQLAlchemy Datastore. Setting this to `False` restores pre 3.3.0 behavior and is required if the `roles` attribute is not a joinable attribute on the `UserModel`. The default setting improves performance by only requiring a single DB call.

Default: `True`.

New in version 3.4.0.

1.4.4 Login/Logout

SECURITY_LOGIN_URL

Specifies the login URL.

Default: `"/login"`.

SECURITY_LOGOUT_URL

Specifies the logout URL.

Default:`"/logout"`.

SECURITY_LOGOUT_METHODS

Specifies the HTTP request methods that the logout URL accepts. Specify `None` to disable the logout URL (and implement your own). Configuring with just `["POST"]` is slightly more secure. The default includes `"GET"` for backwards compatibility.

Default: `["GET", "POST"]`.

SECURITY_POST_LOGIN_VIEW

Specifies the default view to redirect to after a user logs in. This value can be set to a URL or an endpoint name. Defaults to the Flask config `APPLICATION_ROOT` value which itself defaults to `"/"`. Note that if the request URL or form has a `next` parameter, that will take precedence.

Default: `APPLICATION_ROOT`.

`SECURITY_POST_LOGOUT_VIEW`

Specifies the default view to redirect to after a user logs out. This value can be set to a URL or an endpoint name. Defaults to the Flask config `APPLICATION_ROOT` value which itself defaults to `"/"`. Note that if the request URL or form has a `next` parameter, that will take precedence.

Default: `APPLICATION_ROOT`.

`SECURITY_UNAUTHORIZED_VIEW`

Specifies the view to redirect to if a user attempts to access a URL/endpoint that they do not have permission to access. If this value is `None`, the user is presented with a default HTTP 403 response.

Default: `None`.

`SECURITY_LOGIN_USER_TEMPLATE`

Specifies the path to the template for the user login page.

Default: `"security/login_user.html"`.

`SECURITY_VERIFY_URL`

Specifies the re-authenticate URL. If *`SECURITY_FRESHNESS`* evaluates to < 0; this endpoint won't be registered.

Default: `"/verify"`

New in version 3.4.0.

`SECURITY_VERIFY_TEMPLATE`

Specifies the path to the template for the verify password page.

Default: `"security/verify.html"`.

New in version 3.4.0.

`SECURITY_POST_VERIFY_URL`

Specifies the default view to redirect to after a user successfully re-authenticates either via the *`SECURITY_VERIFY_URL`* or the *`SECURITY_US_VERIFY_URL`*. Normally this won't need to be set and after the verification/re-authentication, the referring view (held in the `next` parameter) will be redirected to.

Default: `None`.

New in version 3.4.0.

1.4.5 Registerable

`SECURITY_REGISTERABLE`

Specifies if Flask-Security should create a user registration endpoint.

Default: `False`

`SECURITY_SEND_REGISTER_EMAIL`

Specifies whether registration email is sent.

Default: `True`.

`SECURITY_EMAIL_SUBJECT_REGISTER`

Sets the subject for the confirmation email.

Default: `_("Welcome")`.

`SECURITY_REGISTER_USER_TEMPLATE`

Specifies the path to the template for the user registration page.

Default: `"security/register_user.html"`.

`SECURITY_POST_REGISTER_VIEW`

Specifies the view to redirect to after a user successfully registers. This value can be set to a URL or an endpoint name. If this value is `None`, the user is redirected to the value of `SECURITY_POST_LOGIN_VIEW`. Note that if the request URL or form has a `next` parameter, that will take precedence.

Default: `None`.

`SECURITY_REGISTER_URL`

Specifies the register URL.

Default: `"/register"`.

`SECURITY_USERNAME_ENABLE`

If set to True, the default registration form and template, and login form and template will have a username field added. This requires that your user model contain the field `username`. It MUST be set as 'unique' and if you don't want to require a username, it should be set as 'nullable'.

If you already have added a username field to your forms, don't set this option - the system will throw an exception at init_app time.

Validation and normalization is encapsulated in *`UsernameUtil`*. Note that the default validation restricts username input to be unicode letters and numbers. It also uses `bleach` to scrub any risky input. Be sure your application requirements includes bleach.

Default: `False`

New in version 4.1.0.

`SECURITY_USERNAME_REQUIRED`

If username is enabled, is it required as part of registration?

Default: `False`

New in version 4.1.0.

`SECURITY_USERNAME_MIN_LENGTH`

Minimum length of a username.

Default: `4`

New in version 4.1.0.

`SECURITY_USERNAME_MAX_LENGTH`

Maximum length of a username.

Default: `32`

New in version 4.1.0.

`SECURITY_USERNAME_NORMALIZE_FORM`

Usernames, by default, are normalized using the Python unicodedata.normalize() method.

Default: `"NFKD"`

New in version 4.1.0.

1.4.6 Confirmable

`SECURITY_CONFIRMABLE`

Specifies if users are required to confirm their email address when registering a new account. If this value is *True*, Flask-Security creates an endpoint to handle confirmations and requests to resend confirmation instructions.

Default: `False`.

`SECURITY_CONFIRM_EMAIL_WITHIN`

Specifies the amount of time a user has before their confirmation link expires. Always pluralize the time unit for this value.

Default: `"5 days"`.

`SECURITY_CONFIRM_URL`

Specifies the email confirmation URL.

Default: `"/confirm"`.

`SECURITY_SEND_CONFIRMATION_TEMPLATE`

Specifies the path to the template for the resend confirmation instructions page.

Default: `"security/send_confirmation.html"`.

`SECURITY_EMAIL_SUBJECT_CONFIRM`

Sets the subject for the email confirmation message.

Default: `_("Please confirm your email")`.

`SECURITY_CONFIRM_ERROR_VIEW`

Specifies the view to redirect to if a confirmation error occurs. This value can be set to a URL or an endpoint name. If this value is `None`, the user is presented the default view to resend a confirmation link. In the case of `SECURITY_REDIRECT_BEHAVIOR == spa` query params in the redirect will contain the error.

Default: `None`.

`SECURITY_POST_CONFIRM_VIEW`

Specifies the view to redirect to after a user successfully confirms their email. This value can be set to a URL or an endpoint name. If this value is `None`, the user is redirected to the value of `SECURITY_POST_LOGIN_VIEW`.

Default: `None`.

`SECURITY_AUTO_LOGIN_AFTER_CONFIRM`

If `True`, then the user corresponding to the confirmation token will be automatically signed in. If `False` (the default) then the user will be requires to authenticate using the usual mechanism(s). Note that the confirmation token is not valid after being used once.

Default: `False`.

Deprecated since version 5.3.0.

`SECURITY_LOGIN_WITHOUT_CONFIRMATION`

Specifies if a user may login before confirming their email when the value of `SECURITY_CONFIRMABLE` is set to `True`.

Default: `False`.

`SECURITY_REQUIRES_CONFIRMATION_ERROR_VIEW`

Specifies a redirect page if the users tries to login, reset password or us-signin with an unconfirmed account. If an URL endpoint is specified, flashes an error messages and redirects. Default behavior is to reload the form with an error message without redirecting to an other page.

Default: `None`.

1.4.7 Changeable

Configuration variables for the `SECURITY_CHANGEABLE` feature:

`SECURITY_CHANGEABLE`

Specifies if Flask-Security should enable the change password endpoint.

Default: `False`.

`SECURITY_CHANGE_URL`

Specifies the password change URL.

Default: `"/change"`.

`SECURITY_POST_CHANGE_VIEW`

Specifies the view to redirect to after a user successfully changes their password. This value can be set to a URL or an endpoint name. If this value is `None`, the user is redirected to the value of `SECURITY_POST_LOGIN_VIEW`.

Default: `None`.

`SECURITY_CHANGE_PASSWORD_TEMPLATE`

Specifies the path to the template for the change password page.

Default: `"security/change_password.html"`.

`SECURITY_SEND_PASSWORD_CHANGE_EMAIL`

Specifies whether password change email is sent.

Default: `True`.

`SECURITY_EMAIL_SUBJECT_PASSWORD_CHANGE_NOTICE`

Sets the subject for the password change notice.

Default: `_("Your password has been changed")`.

1.4.8 Recoverable

`SECURITY_RECOVERABLE`

Specifies if Flask-Security should create a password reset/recover endpoint.

Default: `False`.

`SECURITY_RESET_URL`

Specifies the password reset URL.

Default: `"/reset"`.

`SECURITY_RESET_PASSWORD_TEMPLATE`

Specifies the path to the template for the reset password page.

Default: `"security/reset_password.html"`.

`SECURITY_FORGOT_PASSWORD_TEMPLATE`

Specifies the path to the template for the forgot password page.

Default: `"security/forgot_password.html"`.

`SECURITY_POST_RESET_VIEW`

Specifies the view to redirect to after a user successfully resets their password. This value can be set to a URL or an endpoint name. If this value is `None`, the user is redirected to the value of `.login` if *SECURITY_AUTO_LOGIN_AFTER_RESET* is `False` or *SECURITY_POST_LOGIN_VIEW* if `True`

Default: `None`.

`SECURITY_RESET_VIEW`

Specifies the view/URL to redirect to after a GET reset-password link. This is only valid if *SECURITY_REDIRECT_BEHAVIOR* == `spa`. Query params in the redirect will contain the `token`.

Default: `None`.

`SECURITY_AUTO_LOGIN_AFTER_RESET`

If `False` then on successful reset the user will be required to signin again. Note that the reset token is not valid after being used once. If `True`, then the user corresponding to the reset token will be automatically signed in. Note: auto-login is contrary to OWASP best security practices. This option is for backwards compatibility and is deprecated.

Default: `False`.

New in version 5.3.0.

Deprecated since version 5.3.0.

`SECURITY_RESET_ERROR_VIEW`

Specifies the view/URL to redirect to after a GET reset-password link when there is an error. This is only valid if *SECURITY_REDIRECT_BEHAVIOR* == `spa`. Query params in the redirect will contain the error.

Default: `None`.

`SECURITY_RESET_PASSWORD_WITHIN`

Specifies the amount of time a user has before their password reset link expires. Always pluralize the time unit for this value.

Default: `"1 days"`.

`SECURITY_SEND_PASSWORD_RESET_EMAIL`

Specifies whether password reset email is sent. These are instructions including a link that can be clicked on.

Default: `True`.

`SECURITY_SEND_PASSWORD_RESET_NOTICE_EMAIL`

Specifies whether password reset notice email is sent. This is sent once a user's password was successfully reset.

Default: `True`.

`SECURITY_EMAIL_SUBJECT_PASSWORD_RESET`

Sets the subject for the password reset email.

Default: `_("Password reset instructions")`.

`SECURITY_EMAIL_SUBJECT_PASSWORD_NOTICE`

Sets subject for the password notice.

Default: `_("Your password has been reset")`.

1.4.9 Two-Factor

Configuration related to the two-factor authentication feature.

New in version 3.2.0.

SECURITY_TWO_FACTOR

Specifies if Flask-Security should enable the two-factor login feature. If set to `True`, in addition to their passwords, users will be required to enter a code that is sent to them. Note that unless `SECURITY_TWO_FACTOR_REQUIRED` is set - this is opt-in.

Default: `False`.

SECURITY_TWO_FACTOR_REQUIRED

If set to `True` then all users will be required to setup and use two factor authorization.

Default: `False`.

SECURITY_TWO_FACTOR_ENABLED_METHODS

Specifies the default enabled methods for two-factor authentication.

Default: `['email', 'authenticator', 'sms']` which are the only currently supported methods.

SECURITY_TWO_FACTOR_SECRET

Deprecated since version 3.4.0: see: *SECURITY_TOTP_SECRETS*

SECURITY_TWO_FACTOR_URI_SERVICE_NAME

Deprecated since version 3.4.0: see: *SECURITY_TOTP_ISSUER*

SECURITY_TWO_FACTOR_SMS_SERVICE

Deprecated since version 3.4.0: see: *SECURITY_SMS_SERVICE*

SECURITY_TWO_FACTOR_SMS_SERVICE_CONFIG

Deprecated since version 3.4.0: see: *SECURITY_SMS_SERVICE_CONFIG*

SECURITY_TWO_FACTOR_AUTHENTICATOR_VALIDITY

Specifies the number of seconds access token is valid.

Default: `120`.

SECURITY_TWO_FACTOR_MAIL_VALIDITY

Specifies the number of seconds access token is valid.

Default: `300`.

SECURITY_TWO_FACTOR_SMS_VALIDITY

Specifies the number of seconds access token is valid.

Default: `120`.

SECURITY_TWO_FACTOR_RESCUE_MAIL

Specifies the email address users send mail to when they can’t complete the two-factor authentication login.

Default: `"no-reply@localhost"`.

SECURITY_EMAIL_SUBJECT_TWO_FACTOR

Sets the subject for the two factor feature.

Default: `_("Two-factor Login")`

`SECURITY_EMAIL_SUBJECT_TWO_FACTOR_RESCUE`

Sets the subject for the two factor help function.

Default: `_("Two-factor Rescue")`

`SECURITY_TWO_FACTOR_VERIFY_CODE_TEMPLATE`

Specifies the path to the template for the verify code page for the two-factor authentication process.

Default: `"security/two_factor_verify_code.html"`.

`SECURITY_TWO_FACTOR_SETUP_TEMPLATE`

Specifies the path to the template for the setup page for the two factor authentication process.

Default: `"security/two_factor_setup.html"`.

`SECURITY_TWO_FACTOR_SETUP_URL`

Specifies the two factor setup URL.

Default: `"/tf-setup"`.

`SECURITY_TWO_FACTOR_TOKEN_VALIDATION_URL`

Specifies the two factor token validation URL.

Default: `"/tf-validate"`.

`SECURITY_TWO_FACTOR_RESCUE_URL`

Specifies the two factor rescue URL.

Default: `"/tf-rescue"`.

`SECURITY_TWO_FACTOR_SELECT_URL`

Specifies the two factor select URL. This is used when the user has setup more than one second factor.

Default: `"/tf-select"`.

New in version 5.0.0.

`SECURITY_TWO_FACTOR_ERROR_VIEW`

Specifies a URL or endpoint to redirect to if the system detects that a two-factor endpoint is being accessed without the proper state. For example if `tf-validate` is accessed but the caller hasn't yet successfully passed the primary authentication.

Default: `".login"`

New in version 5.1.0.

`SECURITY_TWO_FACTOR_POST_SETUP_VIEW`

Specifies the view to redirect to after a user successfully setups a two-factor method (non-json). This value can be set to a URL or an endpoint name.

Default: `".two_factor_setup"`

New in version 5.1.0.

`SECURITY_TWO_FACTOR_SELECT_TEMPLATE`

Specifies the path to the template for the select method page for the two-factor authentication process. This is used when more than one two-factor method has been setup (e.g. SMS and Webauthn).

Default: `"security/two_factor_select.html"`.

New in version 5.0.0.

`SECURITY_TWO_FACTOR_ALWAYS_VALIDATE`

Specifies whether the application should require a two factor code upon every login. If set to `False` then the 2 values below are used to determine when a code is required. Note that this is cookie based - so a new browser session will always require a fresh two-factor code.

Default: `True`.

`SECURITY_TWO_FACTOR_LOGIN_VALIDITY`

Specifies the expiration of the two factor validity cookie and verification of the token.

Default: `"30 Days"`.

`SECURITY_TWO_FACTOR_VALIDITY_COOKIE`

A dictionary containing the parameters of the two factor validity cookie. The complete set of parameters is described in Flask's set_cookie documentation.

Default: `{'httponly': True, 'secure': False, 'samesite': None}`.

`SECURITY_TWO_FACTOR_IMPLEMENTATIONS`

A dictionary of supported second factor implementations. All of these must implement the TfPluginBase interface.

Default: `{"code": "flask_security.twofactor.CodeTfPlugin", "webauthn": "flask_security.webauthn.WebAuthnTfPlugin",}`

New in version 5.0.0.

`SECURITY_TWO_FACTOR_RESCUE_EMAIL`

If True, then the 'email' option for two-factor rescue is enabled - allowing a user to recover a missing/inoperable second factor device by requesting a one time code sent to their email. While this is very convenient is has the downside that if a user's email is hacked, their second factor is useless to protect their account.

Default: `True`

New in version 5.0.0.

1.4.10 Unified Signin

Unified sign in provides a generalized sign in endpoint that takes an *identity* and a *passcode*.

New in version 3.4.0.

`SECURITY_UNIFIED_SIGNIN`

To enable this feature - set this to `True`.

Default: `False`

`SECURITY_US_SIGNIN_URL`

Sign in a user with an identity and a passcode.

Default: `"/us-signin"`

`SECURITY_US_SIGNIN_SEND_CODE_URL`

Endpoint that given an identity, and a previously setup authentication method, will generate and return a one time code. This isn't necessary when using an authenticator app.

Default: `"/us-signin/send-code"`

`SECURITY_US_SETUP_URL`

Endpoint for setting up and validating SMS or an authenticator app for use in receiving one-time codes.

Default: `"/us-setup"`

`SECURITY_US_VERIFY_LINK_URL`

This endpoint handles the 'magic link' that is sent when the user requests a code via email. It is mostly just accessed via a `GET` from an email reader.

Default: `"/us-verify-link"`

`SECURITY_US_VERIFY_URL`

This endpoint handles re-authentication, the caller must be already authenticated and then enter in their primary credentials (password/passcode) again. This is used when an endpoint (such as `/us-setup`) fails freshness checks. This endpoint won't be registered if *SECURITY_FRESHNESS* evaluates to < 0.

Default: `"/us-verify"`

`SECURITY_US_VERIFY_SEND_CODE_URL`

As part of `/us-verify`, this endpoint will send the appropriate code. This endpoint won't be registered if *SECURITY_FRESHNESS* evaluates to < 0.

Default: `"/us-verify/send-code"`

`SECURITY_US_POST_SETUP_VIEW`

Specifies the view to redirect to after a user successfully setups an authentication method (non-json). This value can be set to a URL or an endpoint name.

Default: `".us-setup"`

`SECURITY_US_SIGNIN_TEMPLATE`

Default: `"security/us_signin.html"`

`SECURITY_US_SETUP_TEMPLATE`

Default: `"security/us_setup.html"`

`SECURITY_US_VERIFY_TEMPLATE`

Default: `"security/us_verify.html"`

`SECURITY_US_ENABLED_METHODS`

Specifies the default enabled methods for unified signin authentication. Be aware that `password` only affects this `SECURITY_US_SIGNIN_URL` endpoint. Removing it from here won't stop users from using the `SECURITY_LOGIN_URL` endpoint (unless you replace the login endpoint using *SECURITY_US_SIGNIN_REPLACES_LOGIN*).

This config variable defines which methods can be used to provide authentication data. *SECURITY_USER_IDENTITY_ATTRIBUTES* controls what sorts of identities can be used.

Default: `["password", "email", "authenticator", "sms"]` - which are the only supported options.

`SECURITY_US_MFA_REQUIRED`

A list of `US_ENABLED_METHODS` that will require two-factor authentication. This is of course dependent on the settings of *SECURITY_TWO_FACTOR* and *SECURITY_TWO_FACTOR_REQUIRED*. Note that even with REQUIRED, only methods listed here will trigger a two-factor cycle.

Default: `["password", "email"]`.

`SECURITY_US_TOKEN_VALIDITY`

Specifies the number of seconds access token/code is valid.

Default: `120`

SECURITY_US_EMAIL_SUBJECT

Sets the email subject when sending the verification code via email.

Default: `_("Verification Code")`

SECURITY_US_SETUP_WITHIN

Specifies the amount of time a user has before their setup token expires. Always pluralize the time unit for this value.

Default: `"30 minutes"`

SECURITY_US_SIGNIN_REPLACES_LOGIN

If set, then the *SECURITY_LOGIN_URL* will be registered to the `us-signin` endpoint. Doing this will mean that logout will properly redirect to the us-signin endpoint.

Default: `False`

Additional relevant configuration variables:

- *SECURITY_USER_IDENTITY_ATTRIBUTES* - Defines the order and methods for parsing and validating identity.
- *SECURITY_PASSWORD_REQUIRED* - Can a user register w/o a password?
- *SECURITY_DEFAULT_REMEMBER_ME*
- *SECURITY_SMS_SERVICE* - When SMS is enabled in *SECURITY_US_ENABLED_METHODS*.
- *SECURITY_SMS_SERVICE_CONFIG*
- *SECURITY_TOTP_SECRETS*
- *SECURITY_TOTP_ISSUER*
- *SECURITY_PHONE_REGION_DEFAULT*
- *SECURITY_LOGIN_ERROR_VIEW* - The user is redirected here if *SECURITY_US_VERIFY_LINK_URL* has an error and the request is json and *SECURITY_REDIRECT_BEHAVIOR* equals `"spa"`.
- *SECURITY_FRESHNESS* - Used to protect /us-setup.
- *SECURITY_FRESHNESS_GRACE_PERIOD* - Used to protect /us-setup.

1.4.11 Passwordless

This feature is DEPRECATED as of 5.0.0. Please use unified signin feature instead.

SECURITY_PASSWORDLESS

Specifies if Flask-Security should enable the passwordless login feature. If set to `True`, users are not required to enter a password to login but are sent an email with a login link. **This feature is being replaced with a more generalized passwordless feature that includes using SMS or authenticator applications for generating codes.**

Default: `False`.

SECURITY_SEND_LOGIN_TEMPLATE

Specifies the path to the template for the send login instructions page for passwordless logins.

Default:`"security/send_login.html"`.

SECURITY_EMAIL_SUBJECT_PASSWORDLESS

Sets the subject for the passwordless feature.

Default: `_("Login instructions")`.

`SECURITY_LOGIN_WITHIN`

Specifies the amount of time a user has before a login link expires. Always pluralize the time unit for this value.

Default: `"1 days"`.

`SECURITY_LOGIN_ERROR_VIEW`

Specifies the view/URL to redirect to after the following login/authentication errors:

- GET passwordless link where the link is expired/incorrect
- GET unified sign in magic link when there is an error.
- GET on oauthresponse where there was an OAuth protocol error.
- GET on oauthresponse where the returned identity isn't registered.

This is only valid if *`SECURITY_REDIRECT_BEHAVIOR`* `== spa`. Query params in the redirect will contain the error.

Default: `None`.

1.4.12 Trackable

`SECURITY_TRACKABLE`

Specifies if Flask-Security should track basic user login statistics. If set to `True`, ensure your models have the required fields/attributes and make sure to commit changes after calling `login_user`. Be sure to use ProxyFix if you are using a proxy.

Default: `False`

1.4.13 WebAuthn

New in version 5.0.0.

`SECURITY_WEBAUTHN`

To enable this feature - set this to `True`. Please see *Models* for required additions to your database models.

Default: `False`

`SECURITY_WAN_REGISTER_URL`

Endpoint for registering WebAuthn credentials.

Default: `"/wan-register"`

`SECURITY_WAN_SIGNIN_URL`

Endpoint for signing in using a WebAuthn credential.

Default: `"/wan-signin"`

`SECURITY_WAN_DELETE_URL`

Endpoint for removing a WebAuthn credential.

Default: `"/wan-delete"`

`SECURITY_WAN_VERIFY_URL`

Endpoint for re-authenticating using a WebAuthn credential.

Default: `"/wan-verify"`

`SECURITY_WAN_POST_REGISTER_VIEW`

Specifies the view to redirect to after a user successfully registers a new WebAuthn key (non-json). This value can be set to a URL or an endpoint name.

Default: `".wan-register"`

`SECURITY_WAN_REGISTER_TEMPLATE`

Default: `"security/wan_register.html"`

`SECURITY_WAN_SIGNIN_TEMPLATE`

Default: `"security/wan_signin.html"`

`SECURITY_WAN_VERIFY_TEMPLATE`

Default: `"security/wan_verify.html"`

`SECURITY_WAN_RP_NAME`

The Relying Party (that's us!) name passed as part of credential creation. Defined in the spec.

Default: `"My Flask App"`

`SECURITY_WAN_REGISTER_WITHIN`

Specifies the amount of time a user has before their register token expires. Always pluralize the time unit for this value.

Default: `"30 minutes"`

`SECURITY_WAN_REGISTER_TIMEOUT`

Specifies the timeout that is passed as part of PublicKeyCredentialCreationOptions. In milliseconds.

Default: `60000`

`SECURITY_WAN_SIGNIN_WITHIN`

Specifies the amount of time a user has before their signin token expires. Always pluralize the time unit for this value.

Default: `"1 minutes"`

`SECURITY_WAN_SIGNIN_TIMEOUT`

Specifies the timeout that is passed as part of PublicKeyCredentialRequestOptions. In milliseconds.

Default: `60000`

`SECURITY_WAN_ALLOW_AS_FIRST_FACTOR`

If True then a WebAuthn credential/key may be registered for use as the first (or only) authentication factor. This will set the default `AuthenticatorSelectionCriteria` to require a cross-platform key.

Default: `True`

`SECURITY_WAN_ALLOW_AS_MULTI_FACTOR`

If True then a WebAuthn credential/key can be used as both a primary and a secondary factor. This requires that the key supports 'UserVerification'.

Default: `True`

`SECURITY_WAN_ALLOW_USER_HINTS`

If True then an unauthenticated user can request a list of registered WebAuthn credentials/keys. This allows the use of non-resident (non-discoverable) keys, but has the possible security concern that it allows 'user discovery'. Look at https://www.w3.org/TR/2021/REC-webauthn-2-20210408/#sctn-username-enumeration for a good writeup.

If this is `False` and *`SECURITY_WAN_ALLOW_AS_FIRST_FACTOR`* is `True` (the default) then by default, `AuthenticatorSelectionCriteria` will be set to require a Resident key.

Default: `True`

`SECURITY_WAN_ALLOW_AS_VERIFY`

Sets which type of WebAuthn security credential, if any, may be used for reauthentication/verify events. This is a list with possible values:

- `"first"` - just keys registered as "first" usage are allowed
- `"secondary"` - just keys registered as "secondary" are allowed

If list is empty or `None` WebAuthn keys aren't allowed. This also means that the
:py:data:`SECURITY_WAN_VERIFY` endpoint won't be registered.

Default: `["first", "secondary"]`

Additional relevant configuration variables:

- *`SECURITY_FRESHNESS`* - Used to protect /us-setup.
- *`SECURITY_FRESHNESS_GRACE_PERIOD`* - Used to protect /us-setup.

1.4.14 Recovery Codes

New in version 5.0.0.

`SECURITY_MULTI_FACTOR_RECOVERY_CODES`

To enable this feature - set this to `True`. Please see *Models* for required additions to your database models. This enables a user to generate and use a recovery code for two-factor authentication. This works for all two-factor mechanisms - including WebAuthn. Note that these code are single use and the user should be advised to write them down and store in a safe place.

`SECURITY_MULTI_FACTOR_RECOVERY_CODES_N`

How many recovery codes to generate.

Default:: `5`

`SECURITY_MULTI_FACTOR_RECOVERY_CODES_URL`

Endpoint for displaying and generating recovery codes.

Default: `"/mf-recovery-codes"`

`SECURITY_MULTI_FACTOR_RECOVERY_CODES_TEMPLATE`

Default: `"security/mf_recovery_codes.html"`

`SECURITY_MULTI_FACTOR_RECOVERY_URL`

Endpoint for entering a recovery code.

Default: `"/mf-recovery"`

`SECURITY_MULTI_FACTOR_RECOVERY_TEMPLATE`

Default: `"security/mf_recovery.html"`

`SECURITY_MULTI_FACTOR_RECOVERY_CODES_KEYS`

A list of keys used to encrypt the recovery codes at rest (i.e. in the database). The default implementation uses cryptography.fernet (https://cryptography.io/en/latest/fernet/#cryptography.fernet.Fernet) - so the keys should be generated by:

```python
from cryptography.fernet import Fernet
key = Fernet.generate_key()
```

Multiple keys can be configured allowing for key rotation.

Default: `None` - recovery codes will NOT be encrypted on disk

New in version 5.1.0.

`SECURITY_MULTI_FACTOR_RECOVERY_CODE_TTL`

An integer passed to decrypt specifying the maximum age of the code.

Default: `None` - no TTL will be enforced.

New in version 5.1.0.

Additional relevant configuration variables:

- *SECURITY_FRESHNESS* - Used to protect /mf-recovery-codes.
- *SECURITY_FRESHNESS_GRACE_PERIOD* - Used to protect /mf-recovery-codes.
- *SECURITY_TOTP_SECRETS* - TOTP/passlib is used to generate the codes.
- *SECURITY_TOTP_ISSUER*

1.4.15 Social Oauth

New in version 5.1.0.

`SECURITY_OAUTH_ENABLE`

To enable using external Oauth providers - set this to `True`.

`SECURITY_OAUTH_BUILTIN_PROVIDERS`

A list of built-in providers to register.

Default: `["google", "github"]`

`SECURITY_OAUTH_START_URL`

Endpoint for starting an Oauth authentication operation.

Default: `"/login/oauthstart"`

`SECURITY_OAUTH_RESPONSE_URL`

Endpoint used as Oauth redirect.

Default: `"/login/oauthresponse"`

1.4.16 Feature Flags

All feature flags. By default all are 'False'/not enabled.

- *SECURITY_CONFIRMABLE*
- *SECURITY_REGISTERABLE*
- *SECURITY_RECOVERABLE*
- *SECURITY_TRACKABLE*
- *SECURITY_PASSWORDLESS*

- *SECURITY_CHANGEABLE*
- *SECURITY_TWO_FACTOR*
- *SECURITY_UNIFIED_SIGNIN*
- *SECURITY_WEBAUTHN*
- *SECURITY_MULTI_FACTOR_RECOVERY_CODES*
- *SECURITY_OAUTH_ENABLE*

1.4.17 URLs and Views

A list of all URLs and Views:

- *SECURITY_LOGIN_URL*
- *SECURITY_LOGOUT_URL*
- *SECURITY_VERIFY_URL*
- *SECURITY_REGISTER_URL*
- *SECURITY_RESET_URL*
- *SECURITY_CHANGE_URL*
- *SECURITY_CONFIRM_URL*
- *SECURITY_MULTI_FACTOR_RECOVERY_CODES_URL*
- *SECURITY_MULTI_FACTOR_RECOVERY_URL*
- *SECURITY_OAUTH_START_URL*
- *SECURITY_OAUTH_RESPONSE_URL*
- *SECURITY_TWO_FACTOR_SELECT_URL*
- *SECURITY_TWO_FACTOR_SETUP_URL*
- *SECURITY_TWO_FACTOR_TOKEN_VALIDATION_URL*
- *SECURITY_TWO_FACTOR_RESCUE_URL*
- *SECURITY_TWO_FACTOR_ERROR_VIEW*
- *SECURITY_TWO_FACTOR_POST_SETUP_VIEW*
- *SECURITY_POST_LOGIN_VIEW*
- *SECURITY_POST_LOGOUT_VIEW*
- *SECURITY_CONFIRM_ERROR_VIEW*
- *SECURITY_POST_REGISTER_VIEW*
- *SECURITY_POST_CONFIRM_VIEW*
- *SECURITY_POST_RESET_VIEW*
- *SECURITY_POST_CHANGE_VIEW*
- *SECURITY_UNAUTHORIZED_VIEW*
- *SECURITY_RESET_VIEW*
- *SECURITY_RESET_ERROR_VIEW*

- *SECURITY_LOGIN_ERROR_VIEW*
- *SECURITY_US_SIGNIN_URL*
- *SECURITY_US_SETUP_URL*
- *SECURITY_US_SIGNIN_SEND_CODE_URL*
- *SECURITY_US_VERIFY_LINK_URL*
- *SECURITY_US_VERIFY_URL*
- *SECURITY_US_VERIFY_SEND_CODE_URL*
- *SECURITY_US_POST_SETUP_VIEW*
- *SECURITY_WAN_REGISTER_URL*
- *SECURITY_WAN_SIGNIN_URL*
- *SECURITY_WAN_DELETE_URL*
- *SECURITY_WAN_VERIFY_URL*
- *SECURITY_WAN_POST_REGISTER_VIEW*

1.4.18 Template Paths

A list of all templates:

- *SECURITY_FORGOT_PASSWORD_TEMPLATE*
- *SECURITY_LOGIN_USER_TEMPLATE*
- *SECURITY_VERIFY_TEMPLATE*
- *SECURITY_REGISTER_USER_TEMPLATE*
- *SECURITY_RESET_PASSWORD_TEMPLATE*
- *SECURITY_CHANGE_PASSWORD_TEMPLATE*
- *SECURITY_MULTI_FACTOR_RECOVERY_TEMPLATE*
- *SECURITY_MULTI_FACTOR_RECOVERY_CODES_TEMPLATE*
- *SECURITY_SEND_CONFIRMATION_TEMPLATE*
- *SECURITY_SEND_LOGIN_TEMPLATE*
- *SECURITY_TWO_FACTOR_VERIFY_CODE_TEMPLATE*
- *SECURITY_TWO_FACTOR_SELECT_TEMPLATE*
- *SECURITY_TWO_FACTOR_SETUP_TEMPLATE*
- *SECURITY_US_SIGNIN_TEMPLATE*
- *SECURITY_US_SETUP_TEMPLATE*
- *SECURITY_US_VERIFY_TEMPLATE*
- *SECURITY_WAN_REGISTER_TEMPLATE*
- *SECURITY_WAN_SIGNIN_TEMPLATE*
- *SECURITY_WAN_VERIFY_TEMPLATE*

1.4.19 Messages

The following are the messages Flask-Security uses. They are tuples; the first element is the message and the second element is the error level.

The default messages and error levels can be found in `core.py`.

- `SECURITY_MSG_ALREADY_CONFIRMED`
- `SECURITY_MSG_API_ERROR`
- `SECURITY_MSG_ANONYMOUS_USER_REQUIRED`
- `SECURITY_MSG_CODE_HAS_BEEN_SENT`
- `SECURITY_MSG_CONFIRMATION_EXPIRED`
- `SECURITY_MSG_CONFIRMATION_REQUEST`
- `SECURITY_MSG_CONFIRMATION_REQUIRED`
- `SECURITY_MSG_CONFIRM_REGISTRATION`
- `SECURITY_MSG_DISABLED_ACCOUNT`
- `SECURITY_MSG_EMAIL_ALREADY_ASSOCIATED`
- `SECURITY_MSG_EMAIL_CONFIRMED`
- `SECURITY_MSG_EMAIL_NOT_PROVIDED`
- `SECURITY_MSG_FAILED_TO_SEND_CODE`
- `SECURITY_MSG_FORGOT_PASSWORD`
- `SECURITY_MSG_GENERIC_AUTHN_FAILED`
- `SECURITY_MSG_GENERIC_RECOVERY`
- `SECURITY_MSG_GENERIC_US_SIGNIN`
- `SECURITY_MSG_IDENTITY_ALREADY_ASSOCIATED`
- `SECURITY_MSG_IDENTITY_NOT_REGISTERED`
- `SECURITY_MSG_INVALID_CODE`
- `SECURITY_MSG_INVALID_CONFIRMATION_TOKEN`
- `SECURITY_MSG_INVALID_EMAIL_ADDRESS`
- `SECURITY_MSG_INVALID_LOGIN_TOKEN`
- `SECURITY_MSG_INVALID_PASSWORD`
- `SECURITY_MSG_INVALID_PASSWORD_CODE`
- `SECURITY_MSG_INVALID_RECOVERY_CODE`
- `SECURITY_MSG_INVALID_REDIRECT`
- `SECURITY_MSG_INVALID_RESET_PASSWORD_TOKEN`
- `SECURITY_MSG_LOGIN`
- `SECURITY_MSG_LOGIN_EMAIL_SENT`
- `SECURITY_MSG_LOGIN_EXPIRED`
- `SECURITY_MSG_NO_RECOVERY_CODES_SETUP`

- SECURITY_MSG_OAUTH_HANDSHAKE_ERROR
- SECURITY_MSG_PASSWORDLESS_LOGIN_SUCCESSFUL
- SECURITY_MSG_PASSWORD_BREACHED
- SECURITY_MSG_PASSWORD_BREACHED_SITE_ERROR
- SECURITY_MSG_PASSWORD_CHANGE
- SECURITY_MSG_PASSWORD_INVALID_LENGTH
- SECURITY_MSG_PASSWORD_IS_THE_SAME
- SECURITY_MSG_PASSWORD_MISMATCH
- SECURITY_MSG_PASSWORD_NOT_PROVIDED
- SECURITY_MSG_PASSWORD_REQUIRED
- SECURITY_MSG_PASSWORD_RESET
- SECURITY_MSG_PASSWORD_RESET_EXPIRED
- SECURITY_MSG_PASSWORD_RESET_NO_LOGIN
- SECURITY_MSG_PASSWORD_RESET_REQUEST
- SECURITY_MSG_PASSWORD_TOO_SIMPLE
- SECURITY_MSG_PHONE_INVALID
- SECURITY_MSG_REAUTHENTICATION_REQUIRED
- SECURITY_MSG_REAUTHENTICATION_SUCCESSFUL
- SECURITY_MSG_REFRESH
- SECURITY_MSG_RETYPE_PASSWORD_MISMATCH
- SECURITY_MSG_TWO_FACTOR_INVALID_TOKEN
- SECURITY_MSG_TWO_FACTOR_LOGIN_SUCCESSFUL
- SECURITY_MSG_TWO_FACTOR_CHANGE_METHOD_SUCCESSFUL
- SECURITY_MSG_TWO_FACTOR_PERMISSION_DENIED
- SECURITY_MSG_TWO_FACTOR_METHOD_NOT_AVAILABLE
- SECURITY_MSG_TWO_FACTOR_DISABLED
- SECURITY_MSG_UNAUTHORIZED
- SECURITY_MSG_UNAUTHENTICATED
- SECURITY_MSG_US_METHOD_NOT_AVAILABLE
- SECURITY_MSG_US_SETUP_EXPIRED
- SECURITY_MSG_US_SETUP_SUCCESSFUL
- SECURITY_MSG_US_SPECIFY_IDENTITY
- SECURITY_MSG_USE_CODE
- SECURITY_MSG_USER_DOES_NOT_EXIST
- SECURITY_MSG_USERNAME_INVALID_LENGTH
- SECURITY_MSG_USERNAME_ILLEGAL_CHARACTERS

- SECURITY_MSG_USERNAME_DISALLOWED_CHARACTERS
- SECURITY_MSG_USERNAME_NOT_PROVIDED
- SECURITY_MSG_USERNAME_ALREADY_ASSOCIATED
- SECURITY_MSG_WEBAUTHN_EXPIRED
- SECURITY_MSG_WEBAUTHN_NAME_REQUIRED
- SECURITY_MSG_WEBAUTHN_NAME_INUSE
- SECURITY_MSG_WEBAUTHN_NAME_NOT_FOUND
- SECURITY_MSG_WEBAUTHN_CREDENTIAL_DELETED
- SECURITY_MSG_WEBAUTHN_REGISTER_SUCCESSFUL
- SECURITY_MSG_WEBAUTHN_CREDENTIAL_ID_INUSE
- SECURITY_MSG_WEBAUTHN_UNKNOWN_CREDENTIAL_ID
- SECURITY_MSG_WEBAUTHN_ORPHAN_CREDENTIAL_ID
- SECURITY_MSG_WEBAUTHN_NO_VERIFY
- SECURITY_MSG_WEBAUTHN_CREDENTIAL_WRONG_USAGE
- SECURITY_MSG_WEBAUTHN_MISMATCH_USER_HANDLE

1.5 Models

Flask-Security assumes you'll be using libraries such as SQLAlchemy, MongoEngine, Peewee or PonyORM to define a *User* and *Role* data model. The fields on your models must follow a particular convention depending on the functionality your app requires. Aside from this, you're free to add any additional fields to your model(s) if you want.

As more features are added to Flask-Security, the list of required fields and tables grow. As you use these features, and therefore require these fields and tables, database migrations are required; which are a bit of a pain. To make things easier - Flask-Security includes mixins that contain ALL the fields and tables required for all features. They also contain various *best practice* fields - such as update and create times. These mixins can be easily extended to add any sort of custom fields and can be found in the *models* module (today there is just one for using Flask-SQLAlchemy).

The provided models are versioned since they represent actual DB models, and any changes require a schema migration (and perhaps a data migration). Applications must specifically import the version they want (and handle any required migration).

Your *User* model needs a Primary Key - Flask-Security doesn't actually reference this - so it can be any name or type your application needs. It should be used in the foreign relationship between *User* and *Role*. The *WebAuthn* model also references this primary key (which can be overridden by providing a suitable implementation of `get_user_mapping`).

At the bare minimum your *User* and *Role* model should include the following fields:

User

- primary key
- `email` (for most features - unique, non-nullable)
- `password` (string, nullable)
- `active` (boolean, non-nullable)
- `fs_uniquifier` (string, 64 bytes, unique, non-nullable)

Role

- primary key
- `name` (unique, non-nullable)
- `description` (string)

1.5.1 Additional Functionality

Depending on the application's configuration, additional fields may need to be added to your database models. Note some fields are specified as 'list of string' the ORM you are using is responsible for translating the list of string to a suitable DB data type. For standard SQL-like databases, Flask-Security provides a utility method *`AsaList`*.

Confirmable

If you enable account confirmation by setting your application's *`SECURITY_CONFIRMABLE`* configuration value to *True*, your *User* model will require the following additional field:

- `confirmed_at` (datetime)

Trackable

If you enable user tracking by setting your application's *`SECURITY_TRACKABLE`* configuration value to *True*, your *User* model will require the following additional fields:

- `last_login_at` (datetime)
- `current_login_at` (datetime)
- `last_login_ip` (string)
- `current_login_ip` (string)
- `login_count` (integer)

Two_Factor

If you enable two-factor by setting your application's *`SECURITY_TWO_FACTOR`* configuration value to *True*, your *User* model will require the following additional fields:

- `tf_totp_secret` (string, 255 bytes, nullable)
- `tf_primary_method` (string)

If you include 'sms' in *`SECURITY_TWO_FACTOR_ENABLED_METHODS`*, your *User* model will require the following additional field:

- `tf_phone_number` (string, 128 bytes, nullable)

Unified Sign In

If you enable unified sign in by setting your application's *SECURITY_UNIFIED_SIGNIN* configuration value to *True*, your *User* model will require the following additional fields:

- `us_totp_secrets` (an arbitrarily long Text field)

If you include 'sms' in *SECURITY_US_ENABLED_METHODS*, your *User* model will require the following additional field:

- `us_phone_number` (string, 64 bytes, nullable, unique)

Separate Identity Domains

If you want authentication tokens to not be invalidated when the user changes their password add the following to your *User* model:

- `fs_token_uniquifier` (string, 64 bytes, unique, non-nullable)

Username

If you set *SECURITY_USERNAME_ENABLE* to *True*, then your *User* model requires the following additional field:

- `username` (string, 64 bytes, unique, nullable)

Permissions

If you want to protect endpoints with permissions, and assign permissions to roles that are then assigned to users, the `Role` model requires:

- `permissions` (list of UnicodeText, nullable)

WebAuthn

Flask Security can act as a WebAuthn Relying Party by enabling *SECURITY_WEBAUTHN*. This requires an additional table as well as references from the User model. Users can have many WebAuthn credentials, and Flask-Security must be able to locate a User record based on a credential id.

Important: It is important that you maintain data consistency when deleting WebAuthn records or users.

The 'WebAuthn' model requires the following fields:

- `id` (primary key)
- `credential_id` (binary, 1024 bytes, indexed, non-nullable, unique)
- `public_key` (binary, 1024 bytes, non-nullable)
- `sign_count` (integer, default=0, non-nullable)
- `transports` (list of string/UnicodeText, nullable)
- `extensions` (string, 255 bytes)
- `lastuse_datetime` (datetime, non-nullable)
- `name` (string, 64 bytes, non-nullable)

- `usage` (string, 64 bytes, non-nullable)
- `backup_state` (boolean, non-nullable)
- `device_type` (string, 64 bytes, non-nullable) (The spec calls this `Backup Eligibility`)

There needs to be a bi-directional relationship between the WebAuthn record and the User record (since we need to look up the `User` based on a WebAuthn `credential_id`.

For SQLAlchemy:

```
Add the following to the WebAuthn model (assuming your primary key is named ``id``):

    @declared_attr
    def user_id(cls):
        return Column(
            Integer,
            ForeignKey("user.id", ondelete="CASCADE"),
            nullable=False,
        )

Add the following to the User model:

    @declared_attr
    def webauthn(cls):
        return relationship("WebAuthn", backref="users", cascade="all, delete")
```

For mongoengine:

```
Add the following to the WebAuthn model:

    user = ReferenceField("User")
    def get_user_mapping(self) -> t.Dict[str, str]:
        """Return the mapping from webauthn back to User"""
        return dict(id=self.user.id)

Add the following to the User model:

    webauthn = ListField(ReferenceField(WebAuthn, reverse_delete_rule=PULL), default=[])

To make sure all WebAuthn objects are deleted if the User is deleted:

    User.register_delete_rule(WebAuthn, "user", CASCADE)
```

For peewee:

```
Add the following to the WebAuthn model:

    user = ForeignKeyField(User, backref="webauthn")

This will add a column called ``user_id`` that references the User model's
``id`` primary key field. It will also create a virtual column ``webauthn``
as part of the User model. Note that the default Peewee datastore implementation
calls ``delete_instance(recursive=True)`` which correctly deals with ensuring
that WebAuthn records get deleted if a User is deleted.
```

The *User* model needs the following additional fields:

- `fs_webauthn_user_handle` (string, 64 bytes, unique). This is used as the *PublicKeyCredentialUserEntity id* value.

Recovery Codes

If *SECURITY_MULTI_FACTOR_RECOVERY_CODES* is set to `True` then the *User* model needs the following field:

- `mf_recovery_codes` (list of string/UnicodeText, nullable)

A recovery code can be used in place of any configured second-factor authenticator (e.g. SMS, WebAuthn, . . .).

Custom User Payload

If you want a custom payload for JSON API responses, define the method *get_security_payload* in your User model. The method must return a serializable object:

```
class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    email = TextField()
    password = TextField()
    active = BooleanField(default=True)
    confirmed_at = DateTimeField(null=True)
    name = db.Column(db.String(80))

    # Custom User Payload
    def get_security_payload(self):
        rv = super().get_security_payload()
        # :meth:`User.calc_username`
        rv["username"] = self.calc_username()
        rv["confirmation_needed"] = self.confirmed_at is None
        return rv
```

CHAPTER

TWO

CUSTOMIZING AND USAGE PATTERNS

2.1 Customizing

Flask-Security bootstraps your application with various views for handling its configured features to get you up and running as quickly as possible. However, you'll probably want to change the way these views look to be more in line with your application's visual design.

2.1.1 Views

Flask-Security is packaged with a default template for each view it presents to a user. Templates are located within a subfolder named `security`. The following is a list of view templates:

- *security/forgot_password.html*
- *security/login_user.html*
- *security/mf_recovery.html*
- *security/mf_recovery_codes.html*
- *security/register_user.html*
- *security/reset_password.html*
- *security/change_password.html*
- *security/send_confirmation.html*
- *security/send_login.html*
- *security/verify.html*
- *security/two_factor_select.html*
- *security/two_factor_setup.html*
- *security/two_factor_verify_code.html*
- *security/us_signin.html*
- *security/us_setup.html*
- *security/us_verify.html*
- *security/wan_register.html*
- *security/wan_signin.html*
- *security/wan_verify.html*

Overriding these templates is simple:

1. Create a folder named `security` within your application's templates folder
2. Create a template with the same name for the template you wish to override

You can also specify custom template file paths in the *configuration*.

Each template is passed a template context object that includes the following, including the objects/values that are passed to the template by the main Flask application context processor:

- `<template_name>_form`: A form object for the view
- `security`: The Flask-Security extension object

To add more values to the template context, you can specify a context processor for all views or a specific view. For example:

```
security = Security(app, user_datastore)

# This processor is added to all templates
@security.context_processor
def security_context_processor():
    return dict(hello="world")

# This processor is added to only the register view
@security.register_context_processor
def security_register_processor():
    return dict(something="else")
```

The following is a list of all the available context processor decorators:

- `context_processor`: All views
- `forgot_password_context_processor`: Forgot password view
- `login_context_processor`: Login view
- `mf_recovery_codes_context_processor`: Setup recovery codes view
- `mf_recovery_context_processor`: Use recovery code view
- `register_context_processor`: Register view
- `reset_password_context_processor`: Reset password view
- `change_password_context_processor`: Change password view
- `send_confirmation_context_processor`: Send confirmation view
- `send_login_context_processor`: Send login view
- `mail_context_processor`: Whenever an email will be sent
- `tf_select_context_processor`: Two factor select view
- `tf_setup_context_processor`: Two factor setup view
- `tf_token_validation_context_processor`: Two factor token validation view
- `us_signin_context_processor`: Unified sign in view
- `us_setup_context_processor`: Unified sign in setup view
- `wan_register_context_processor`: WebAuthn registration view
- `wan_signin_context_processor`: WebAuthn sign in view

- `wan_verify_context_processor`: WebAuthn verify view

2.1.2 Forms

All forms can be overridden. For each form used, you can specify a replacement class. This allows you to add extra fields to any form or override validators. For example it is often desired to add additional personal information fields to the registration form:

```python
from flask_security import RegisterForm
from wtforms import StringField
from wtforms.validators import DataRequired

class ExtendedRegisterForm(RegisterForm):
    first_name = StringField('First Name', [DataRequired()])
    last_name = StringField('Last Name', [DataRequired()])

security = Security(app, user_datastore,
         register_form=ExtendedRegisterForm)
```

For the `register_form` and `confirm_register_form`, only fields that exist in the user model are passed (as kwargs) to *`UserDatastore.create_user()`*. Thus, in the above case, the `first_name` and `last_name` fields will only be passed if the model looks like:

```python
class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(255), unique=True)
    password = db.Column(db.String(255))
    first_name = db.Column(db.String(255))
    last_name = db.Column(db.String(255))
```

Warning: Adding fields is fine - however re-defining existing fields could cause various views to no longer function. Many fields have complex (and not publicly exposed) validators that have side effects.

Warning: It is important to ALWAYS subclass the base Flask-Security form and not attempt to just redefine the class. This is due to the validation method of many of the forms performs critical additional validation AND will change or add values to the form as a side-effect. See below for how to do this.

If you need to override an existing field in a form (to override/add validators), and you want to define a re-usable validator - use multiple inheritance - be extremely careful about the order of the inherited classes:

```python
from wtforms import PasswordField, ValidationError
from wtforms.validators import DataRequired

def password_validator(form, field):
    if field.data.startswith("PASS"):
        raise ValidationError("Really - don't start a password with PASS")

class NewPasswordFormMixinEx:
    password = PasswordField("password",
```

(continues on next page)

(continued from previous page)

```
                              validators=[DataRequired(message="PASSWORD_NOT_PROVIDED"),
                                          password_validator])

class MyRegisterForm(NewPasswordFormMixinEx, ConfirmRegisterForm):
    pass

app.config["SECURITY_CONFIRM_REGISTER_FORM"] = MyRegisterForm
```

The following is a list of all the available form overrides:

- `login_form`: Login form
- `verify_form`: Verify form
- `confirm_register_form`: Confirmable register form
- `register_form`: Register form
- `forgot_password_form`: Forgot password form
- `reset_password_form`: Reset password form
- `change_password_form`: Change password form
- `send_confirmation_form`: Send confirmation form
- `mf_recovery_codes_form`: Setup recovery codes form
- `mf_recovery_form`: Use recovery code form
- `passwordless_login_form`: Passwordless login form
- `two_factor_verify_code_form`: Two-factor verify code form
- `two_factor_select_form`: Two-factor select form
- `two_factor_setup_form`: Two-factor setup form
- `two_factor_rescue_form`: Two-factor help user form
- `us_signin_form`: Unified sign in form
- `us_setup_form`: Unified sign in setup form
- `us_setup_validate_form`: Unified sign in setup validation form
- `us_verify_form`: Unified sign in verify form
- `wan_delete_form`: WebAuthn delete a registered key form
- `wan_register_form`: WebAuthn initiate registration ceremony form
- `wan_register_response_form`: WebAuthn registration ceremony form
- `wan_signin_form`: WebAuthn initiate sign in ceremony form
- `wan_signin_response_form`: WebAuthn sign in ceremony form
- `wan_verify_form`: WebAuthn verify form

Tip: Changing/extending the form class won't directly change how it is displayed. You need to ALSO provide your own template and explicitly add the new fields you want displayed.

Controlling Form Instantiation

This is an advanced concept! Please see *Security.set_form_info()* and *FormInfo*.

This is an example of providing your own form instantiator using the 'form clone' pattern. In this example we are injecting an external *service* into the form for use in validation:

```
from flask_security import FormInfo

class MyLoginForm(LoginForm):
    def __init__(self, *args, service=None, **kwargs):
        super().__init__(*args, **kwargs)
        self.myservice = service

    def instantiator(self, form_name, form_cls, *args, **kwargs):
        return MyLoginForm(*args, service=self.myservice, **kwargs)

    def validate(self, **kwargs: t.Any) -> bool:
        if not super().validate(**kwargs):  # pragma: no cover
            return False
        if not self.myservice(self.email.data):
            self.email.errors.append("Not happening")
            return False
        return True

# A silly service that only allows 'matt" log in!
def login_checker(email):
    return True if email == "matt@lp.com" else False

with app.test_request_context():
    # Flask-WTForms require a request context.
    fi = MyLoginForm(formdata=None, service=login_checker)
app.security.set_form_info("login_form", FormInfo(fi.instantiator))
```

Customizing the Login Form

This is an example of how to modify the registration and login form to add support for a single input field to accept both email and username (mimicking legacy Flask-Security behavior). Flask-Security supports username as a configuration option so this is not strictly needed any more, however, Flask-Security's LoginForm uses 2 different input fields (so that appropriate input attributes can be set):

```
from flask_security import (
        RegisterForm,
        LoginForm,
        Security,
        lookup_identity,
        uia_username_mapper,
        unique_identity_attribute,
    )
    from werkzeug.local import LocalProxy
    from wtforms import StringField, ValidationError, validators

    def username_validator(form, field):
```

(continues on next page)

(continued from previous page)

```
        # Side-effect - field.data is updated to normalized value.
        # Use proxy to we can declare this prior to initializing Security.
        _security = LocalProxy(lambda: app.extensions["security"])
        msg, field.data = _security._username_util.validate(field.data)
        if msg:
            raise ValidationError(msg)

    class MyRegisterForm(RegisterForm):
        # Note that unique_identity_attribute uses the defined field 'mapper' to
        # normalize. We validate before that to give better error messages and
        # to set the normalized value into the form for saving.
        username = StringField(
            "Username",
            validators=[
                validators.data_required(),
                username_validator,
                unique_identity_attribute,
            ],
        )

    class MyLoginForm(LoginForm):
        email = StringField("email", [validators.data_required()])

        def validate(self, **kwargs):
            self.user = lookup_identity(self.email.data)
            # Setting 'ifield' informs the default login form validation
            # handler that the identity has already been confirmed.
            self.ifield = self.email
            if not super().validate(**kwargs):
                return False
            return True

    # Allow registration with email, but login only with username
    app.config["SECURITY_USER_IDENTITY_ATTRIBUTES"] = [
        {"username": {"mapper": uia_username_mapper}}
    ]
    security = Security(
        datastore=sqlalchemy_datastore,
        register_form=MyRegisterForm,
        login_form=MyLoginForm,
    )
    security.init_app(app)
```

2.1.3 Localization

All messages, form labels, and form strings are localizable. Flask-Security uses Flask-Babel or Flask-BabelEx to manage its messages.

Tip: Be sure to explicitly initialize your babel extension:

```
import flask_babel

flask_babel.Babel(app)
```

All translations are tagged with a domain, as specified by the configuration variable `SECURITY_I18N_DOMAIN` (default: "flask_security"). For messages and labels all this works seamlessly. For strings inside templates it is necessary to explicitly ask for the "flask_security" domain, since your application itself might have its own domain. Flask-Security places the method `_fsdomain` in jinja2's global environment and uses that in all templates. In order to reference a Flask-Security translation from ANY template (such as if you copied and modified an existing security template) just use that method:

```
{{ _fsdomain("Login") }}
```

Be aware that Flask-Security will validate and normalize email input using the email_validator package. The normalized form is stored in the DB.

Overriding Messages

It is possible to change one or more messages (either the original default english and/or a specific translation). Adding the following to your app:

```
app.config["SECURITY_MSG_INVALID_PASSWORD"] = ("Password no-worky", "error")
```

will change the default message in english.

Tip: The string messages themselves are a 'key' into the translation .po/.mo files. Do not pass in gettext('string') or lazy_gettext('string).

If you need translations then you need to create your own `translations` directory and add the appropriate .po files and compile them. Finally, add your translations directory path to the configuration. In this example, create a file `flask_security.po` under a directory: `translations/fr_FR/LC_MESSAGES` (for french) with the following contents:

```
msgid ""
msgstr ""

msgid "Password no-worky"
msgstr "Passe - no-worky"
```

Then compile it with:

```
pybabel compile -d translations/ -i translations/fr_FR/LC_MESSAGES/flask_security.po -l␣
↪fr_FR -D flask_security
```

Finally add your translations directory to your configuration:

```
app.config["SECURITY_I18N_DIRNAME"] = ["builtin", "translations"]
```

Note: This only works when using Flask-Babel since Flask-BabelEx doesn't support a list of translation directories.

2.1.4 Emails

Flask-Security is also packaged with a default template for each email that it may send. Templates are located within the subfolder named `security/email`. The following is a list of email templates:

- *security/email/confirmation_instructions.html*
- *security/email/confirmation_instructions.txt*
- *security/email/login_instructions.html*
- *security/email/login_instructions.txt*
- *security/email/reset_instructions.html*
- *security/email/reset_instructions.txt*
- *security/email/reset_notice.html*
- *security/email/reset_notice.txt*
- *security/email/change_notice.txt*
- *security/email/change_notice.html*
- *security/email/welcome.html*
- *security/email/welcome.txt*
- *security/email/welcome_existing.html*
- *security/email/welcome_existing.txt*
- *security/email/welcome_existing_username.html*
- *security/email/welcome_existing_username.txt*
- *security/email/two_factor_instructions.html*
- *security/email/two_factor_instructions.txt*
- *security/email/two_factor_rescue.html*
- *security/email/two_factor_rescue.txt*
- *security/email/us_instructions.html*
- *security/email/us_instructions.txt*

Overriding these templates is simple:

1. Create a folder named `security` within your application's templates folder
2. Create a folder named `email` within the `security` folder
3. Create a template with the same name for the template you wish to override

Each template is passed a template context object that includes values as described below. In addition, the `security` object is always passed - you can for example render any security configuration variable via `security.lower_case_variable_name` and don't include the prefix `security_` (e.g. `{{ security.confirm_url }}`). If you require more values in the templates, you can specify an email context processor with the `mail_context_processor` decorator. For example:

```
security = Security(app, user_datastore)

# This processor is added to all emails
@security.mail_context_processor
def security_mail_processor():
    return dict(hello="world")
```

There are many configuration variables associated with emails, and each template will receive a slightly different context. The `Gate Config` column are configuration variables that if set to `False` will bypass sending of the email (they all default to `True`). In most cases, in addition to an email being sent, a *Signal* is sent. The table below summarizes all this:

Template Name	Gate Config	Subject Config	Context Vars	Signal Sent
welcome	SECU-RITY_SEND_REGIS	SECU-RITY_EMAIL_SUBJ	• user • confirma-tion_link • confirma-tion_token	user_registered
confirma-tion_instructions	N/A	SECU-RITY_EMAIL_SUBJ	• user • confirma-tion_link • confirma-tion_token	con-firm_instructions_sent
login_instructions	N/A	SECU-RITY_EMAIL_SUBJ	• user • login_link • login_token	lo-gin_instructions_sent
reset_instructions	SEND_PASSWORD_	SECU-RITY_EMAIL_SUBJ	• user • reset_link • reset_token	re-set_password_instructions_sent
reset_notice	SEND_PASSWORD_	SECU-RITY_EMAIL_SUBJ	• user	password_reset
change_notice	SEND_PASSWORD_	SECU-RITY_EMAIL_SUBJ	• user	password_changed
two_factor_instructio	N/A	SECU-RITY_EMAIL_SUBJ	• user • token • username	tf_security_token_sent
two_factor_rescue	N/A	SECU-RITY_EMAIL_SUBJ	• user	N/A
us_instructions	N/A	SECU-RITY_US_EMAIL_S	• user • login_token • login_link • username	us_security_token_sent
welcome_existing	SECU-RITY_SEND_REGIS SECU-RITY_RETURN_GE	SECU-RITY_EMAIL_SUBJ	• user • recovery_link	user_not_registered
wel-come_existing_userna	SECU-RITY_SEND_REGIS SECU-RITY_RETURN_GE	SECU-RITY_EMAIL_SUBJ	• email • username	user_not_registered

When sending an email, Flask-Security goes through the following steps:

1. Calls the email context processor as described above
2. Calls `render_template` (as configured at Flask-Security initialization time) with the context and template to

produce a text and/or html version of the message

3. Calls *MailUtil.send_mail()* with all the required parameters.

The default implementation of `MailUtil.send_mail` uses flask-mailman to create and send the message. By providing your own implementation, you can use any available python email handling package.

Email subjects are by default localized - see above section on Localization to learn how to customize them.

Emails with Celery

Sometimes it makes sense to send emails via a task queue, such as Celery. This is supported by providing your own implementation of the *MailUtil* class:

```
from flask_security import MailUtil
class MyMailUtil(MailUtil):

    def send_mail(self, template, subject, recipient, sender, body, html, **kwargs):
        send_flask_mail.delay(
            subject=subject,
            from_email=sender,
            to=[recipient],
            body=body,
            html=html,
        )
```

Then register your class as part of Flask-Security initialization:

```
from flask import Flask
from flask_mailman import EmailMultiAlternatives, Mail
from flask_security import Security, SQLAlchemyUserDatastore
from celery import Celery

mail = Mail()
security = Security()
celery = Celery()


@celery.task
def send_flask_mail(**kwargs):
    with app.app_context():
        with mail.get_connection() as connection:
            html = kwargs.pop("html", None)
            msg = EmailMultiAlternatives(**kwargs, connection=connection)
            if html:
                msg.attach_alternative(html, "text/html")
            msg.send()

def create_app(config):
    """Initialize Flask instance."""

    app = Flask(__name__)
    app.config.from_object(config)
```

(continues on next page)

(continued from previous page)

```
    mail.init_app(app)
    datastore = SQLAlchemyUserDatastore(db, User, Role)
    security.init_app(app, datastore, mail_util_cls=MyMailUtil)

    return app
```

2.1.5 Responses

Flask-Security will likely be a very small piece of your application, so Flask-Security makes it easy to override all aspects of API responses.

JSON Response

Applications that support a JSON based API need to be able to have a uniform API response. Flask-Security has a default way to render its API responses - which can be easily overridden by providing a callback function via `Security.render_json()`. Be aware that Flask-Security subclasses Flask's JSONProvider interface and sets it on *app.json_provider_cls*.

401, 403, Oh My

For a very long read and discussion; look at this. Out of the box, Flask-Security in tandem with Flask-Login, behave as follows:

- If authentication fails as the result of a *@login_required*, *@auth_required("session", "token")*, or *@token_auth_required* then if the request 'wants' a JSON response, `Security.render_json()` is called with a 401 status code. If not then flask_login.LoginManager.unauthorized() is called. By default THAT will redirect to a login view.
- If authentication fails as the result of a *@http_auth_required* or *@auth_required("basic")* then a 401 is returned along with the http header `WWW-Authenticate` set to `Basic realm="xxxx"`. The realm name is defined by `SECURITY_DEFAULT_HTTP_AUTH_REALM`.
- If authorization fails as the result of *@roles_required*, *@roles_accepted*, *@permissions_required*, or *@permissions_accepted*, then if the request 'wants' a JSON response, `Security.render_json()` is called with a 403 status code. If not, then if `SECURITY_UNAUTHORIZED_VIEW` is defined, the response will redirected. If `SECURITY_UNAUTHORIZED_VIEW` is not defined, then `abort(403)` is called.

All this can be easily changed by registering any or all of `Security.render_json()`, `Security.unauthn_handler()` and `Security.unauthz_handler()`.

The decision on whether to return JSON is based on:

- Was the request content-type "application/json" (e.g. request.is_json()) OR
- Is the 'best' value of the `Accept` HTTP header "application/json"

2.1.6 Redirects

Flask-Security uses redirects frequently (when using forms), and most of the redirect destinations are configurable. When Flask-Security initiates a redirect it (almost) always flashes a message that provides some context for the user. In addition, Flask-Security - both in its views and default templates, attempts to propagate any *next* query param and in fact, an existing *?next=/xx* will override most of the configuration redirect URLs.

As a complex example consider an unauthenticated user accessing a *@auth_required* endpoint, and the user has two-factor authentication set up.:

- GET("/protected") - The *default_unauthn_handler* via Flask-Login will redirect to `/login?next=/protected`
- The login form/template will pick any *?next=/xx* argument off the request URL and append it to form action.
- When the form is submitted if will do a POST("/login?next=/protected")
- Assuming correct authentication, the system will send out a 2-factor code and redirect to `/tf-verify?next=/protected`
- The two_factor_validation_form/template also pulls any *?next=/xx* and appends to the form action.
- When the *tf-validate* form is submitted it will do a POST("/tf-validate?next=/protected").
- Assuming a correct code, the user is authenticated and is redirected. That redirection first looks for a 'next' in the request.args then in request.form and finally will use the value of *SECURITY_POST_LOGIN_VIEW*. In this example it will find the `next=/protected` in the request.args and redirect to `/protected`.

2.2 WebAuthn

WebAuthn/FIDO2 is a W3C standard that defines a cryptographic protocol between a relying party (your Flask application) and an authenticator. In simple terms this allows connecting your Flask application to a variety of authenticators including dedicated hardware (e.g. YubiKey) and devices that have cryptographic capabilities (e.g. a mobile phone with fingerprint or face id).

This protocol is supported by all major browsers, however as of Spring 2022, few web application actually support it, and those that do normally just support using a WebAuthn key as an additional, optional, second factor.

Note that a WebAuthn key can possibly satisfy a complete 2-factor authentication requirement - something you have and something you are (think a mobile device with face-Id). Flask-Security supports this use case.

2.2.1 Key Concepts

While the spec is quite complex - there are 2 important concepts to know. WebAuthn keys can be classified as 'platform'/'cross-platform' and 'resident'/or not. If a key is a `platform` key that means it is tied to a particular device. If you set up a second factor WebAuthn key that is a `platform` key you will ONLY BE ABLE TO AUTHENTICATE using that device. For that reason it is a best practice to make sure there is at least one second-factor authentication method setup that is NOT platform specific (such as SMS, an authenticator app, etc.).

Flask-Security requires that when registering a WebAuthn key, the user must specify whether the key will be used for first/primary authentication or for multi-factor/second authentication.

It should be noted the the current spec REQUIRES javascript to communicate from your front-end to the browser. Flask-Security ships with the basic required JS (static/{webauthn.js,base64.js}). An application should be able to simply wire those into their templates or javascript.

2.2.2 Configuration

As with many features in Flask-Security, configuration is a combination of config variables, constructor parameters, and a sub-classable utility class. The WebAuthn spec offers a lot of flexibility in supporting a wide range of authenticators. The default configuration is:

- Allow a WebAuthn key to be used for first/primary authentication (*SECURITY_WAN_ALLOW_AS_FIRST_FACTOR* = `True`)
- Allow a WebAuthn key to be used as a multi-factor (both first and secondary) if the key supports it (*SECURITY_WAN_ALLOW_AS_MULTI_FACTOR* = `True`)
- Allow both 'first' and 'secondary' WebAuthn keys to be used for 'freshness' verification (*SECURITY_WAN_ALLOW_AS_VERIFY* = `True`)
- Allow returning WebAuthn key names to un-authenticated users (*SECURITY_WAN_ALLOW_USER_HINTS* = `True`) Please see this portion of the WebAuthn spec for security implications.

The bundled *WebauthnUtil* class implements the following defaults:

- The `AuthenticatorSelectionCriteria` is set to `CROSS_PLATFORM` for webauthn keys being registered for first/primary authentication.
- The `UserVerificationRequirement` is set to `DISCOURAGED` for keys used for secondary authentication, and `PREFERRED` for keys used for first/primary or multi-factor.

2.3 Two-factor Configurations

Two-factor authentication provides a second layer of security to any type of login, requiring extra information or a secondary device to log in, in addition to ones login credentials. The added feature includes the ability to add a secondary authentication method using either via email, sms message, or an Authenticator app such as Google, Lastpass, or Authy.

The following code sample illustrates how to get started as quickly as possible using SQLAlchemy and two-factor feature. In this example both email and an authenticator app is supported as a second factor. See below for information about SMS.

2.3.1 Basic SQLAlchemy Two-Factor Application

SQLAlchemy Install requirements

```
$ python3 -m venv pymyenv
$ . pymyenv/bin/activate
$ pip install flask-security-too[common,mfa,fsqla]
```

Two-factor Application

The following code sample illustrates how to get started as quickly as possible using SQLAlchemy:

```
import os
from flask import Flask, current_app, render_template_string
from flask_sqlalchemy import SQLAlchemy
from flask_security import Security, SQLAlchemyUserDatastore, \
    UserMixin, RoleMixin, auth_required
```

(continues on next page)

(continued from previous page)

```
from flask_mailman import Mail

# Create app
app = Flask(__name__)
app.config['DEBUG'] = True
# Generate a nice key using secrets.token_urlsafe()
app.config['SECRET_KEY'] = os.environ.get("SECRET_KEY", 'pf9Wkove4IKEAXvy-cQkeDPhv9Cb3Ag-
↪wyJILbq_dFw')
# Bcrypt is set as default SECURITY_PASSWORD_HASH, which requires a salt
# Generate a good salt using: secrets.SystemRandom().getrandbits(128)
app.config['SECURITY_PASSWORD_SALT'] = os.environ.get("SECURITY_PASSWORD_SALT",
↪'146585145368132386173505678016728509634')

app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite://'

app.config['SECURITY_TWO_FACTOR_ENABLED_METHODS'] = ['email',
  'authenticator']  # 'sms' also valid but requires an sms provider
app.config['SECURITY_TWO_FACTOR'] = True
app.config['SECURITY_TWO_FACTOR_RESCUE_MAIL'] = "put_your_mail@gmail.com"

app.config['SECURITY_TWO_FACTOR_ALWAYS_VALIDATE'] = False
app.config['SECURITY_TWO_FACTOR_LOGIN_VALIDITY'] = "1 week"

# Generate a good totp secret using: passlib.totp.generate_secret()
app.config['SECURITY_TOTP_SECRETS'] = {"1": "TjQ9Qa31VOrfEzuPy4VHQWPCTmRzCnFzMKLxXYiZu9B
↪"}
app.config['SECURITY_TOTP_ISSUER'] = "put_your_app_name"

app.config["SQLALCHEMY_ENGINE_OPTIONS"] = {
    "pool_pre_ping": True,
}
app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False

# Create database connection object
db = SQLAlchemy(app)

# Define models
roles_users = db.Table('roles_users',
    db.Column('user_id', db.Integer(), db.ForeignKey('user.id')),
    db.Column('role_id', db.Integer(), db.ForeignKey('role.id')))

class Role(db.Model, RoleMixin):
  id = db.Column(db.Integer(), primary_key=True)
  name = db.Column(db.String(80), unique=True)
  description = db.Column(db.String(255))

class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(255), unique=True)
    # Make username unique but not required.
    username = db.Column(db.String(255), unique=True, nullable=True)
    password = db.Column(db.String(255))
```

(continues on next page)

(continued from previous page)

```
    active = db.Column(db.Boolean())
    fs_uniquifier = db.Column(db.String(255), unique=True, nullable=False)
    confirmed_at = db.Column(db.DateTime())
    roles = db.relationship('Role', secondary=roles_users,
                            backref=db.backref('users', lazy='dynamic'))
    tf_phone_number = db.Column(db.String(128), nullable=True)
    tf_primary_method = db.Column(db.String(64), nullable=True)
    tf_totp_secret = db.Column(db.String(255), nullable=True)

# Setup Flask-Security
user_datastore = SQLAlchemyUserDatastore(db, User, Role)
app.security = Security(app, user_datastore)

mail = Mail(app)

# Views
@app.route('/')
@auth_required()
def home():
    return render_template_string("Hello {{ current_user.email }}")

# one time setup
with app.app_context():
    # Create a user to test with
    db.create_all()
    if not app.security.datastore.find_user(email='test@me.com'):
        app.security.datastore.create_user(email='test@me.com', password='password')
    db.session.commit()

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

2.3.2 Adding SMS

Using SMS as a second factor requires access to an SMS service provider such as “Twilio”. Flask-Security supports Twilio out of the box. For other sms service providers you will need to subclass *SmsSenderBaseClass* and register it:

```
SmsSenderFactory.senders[<service-name>] = <service-class>
```

You need to install additional packages:

```
pip install phonenumberslite twilio
```

And set additional configuration variables:

```
app.config["SECURITY_TWO_FACTOR_ENABLED_METHODS"] = ['email',
  'authenticator', 'sms']
app.config["SECURITY_SMS_SERVICE"] = "Twilio"
app.config["SECURITY_SMS_SERVICE_CONFIG"] =
  {'ACCOUNT_SID': <from twilio>, 'AUTH_TOKEN': <from twilio>, 'PHONE_NUMBER': <from␣
↪twilio>}
```

2.3.3 Theory of Operation

Note: The Two-factor feature requires that session cookies be received and sent as part of the API. This is true regardless of whether the application uses forms or JSON.

The Two-factor (2FA) API has four paths:

- Normal login once everything set up
- Changing 2FA setup
- Initial login/registration when 2FA is required
- Rescue

When using forms, the flow from one state to the next is handled by the forms themselves. When using JSON the application must of course explicitly access the appropriate endpoints. The descriptions below describe the JSON access pattern.

Normal Login

In the normal case, when the user has already setup their preferred 2FA method (e.g. email, SMS, authenticator app), then the flow starts with the authentication process using the `/login` or `/us-signin` endpoints, providing their identity and password. If 2FA is required, the response will indicate that. Then, the application must POST to the `/tf-validate` with the correct code.

Changing 2FA Setup

An authenticated user can change their 2FA configuration (primary_method, phone number, etc.). In order to prevent a user from being locked out, the new configuration must be validated before it is stored permanently. The user starts with a GET on `/tf-setup`. This will return a list of configured 2FA methods the user can choose from, and the existing configuration. This must be followed with a POST on `/tf-setup` with the new primary method (and phone number if SMS). In the case of SMS, a code will be sent to the phone/device and again use `/tf-validate` to confirm code. In the case of setting up an authenticator app, the response to the POST will contain the QRcode image as well as the required information for manual entry. Once the code has been successfully entered, the new configuration will be permanently stored.

Initial login/registration

This is basically a combination of the above two - initial POST to `/login` will return indicating that 2FA is required. The user must then POST to `/tf-setup` to setup the desired 2FA method, and finally have the user enter the code and POST to `/tf-validate`.

Rescue

Life happens - if the user doesn't have their mobile devices (SMS) or authenticator app, then they can use the `/tf-rescue` endpoint to see possible recovery options. Flask-Security supports the following:

- Have a one-time code sent to their email (if *SECURITY_TWO_FACTOR_RESCUE_EMAIL* is set to `True`).
- Send an email to the application administrators.
- Use a previously setup one-time recovery code (see *SECURITY_MULTI_FACTOR_RECOVERY_CODES*)

2.3.4 Validity

Sometimes it can be preferable to enter the 2FA code once a day/week/month, especially if a user logs in and out of a website multiple times. This allows the security of a two factor authentication but with a slightly better user experience. This can be achieved by setting *SECURITY_TWO_FACTOR_ALWAYS_VALIDATE* to `False`, and clicking the 'Remember' button on the login form. Once the two factor code is validated, a cookie is set to allow skipping the validation step. The cookie is named `tf_validity` and contains the signed token containing the user's `fs_uniquifier`. The cookie and token are both set to expire after the time delta given in *SECURITY_TWO_FACTOR_LOGIN_VALIDITY*. Note that setting `SECURITY_TWO_FACTOR_LOGIN_VALIDITY` to 0 is equivalent to `SECURITY_TWO_FACTOR_ALWAYS_VALIDATE` being `True`.

2.4 Working with Single Page Applications

Single Page Applications (spa) are a popular model for both separating user interface from application/backend code as well as providing a responsive user experience. Angular and Vue are popular Javascript frameworks for writing SPAs. An added benefit is that the UI can be developed completely independently (in a separate repo) and take advantage of the latest Javascript packing and bundling technologies that are evolving rapidly, and not make the Flask application have to deal with things like Flask-Webpack or webassets.

For the purposes of this application note - this implies:

- The user interface code is delivered by some other means than the Flask application. In particular this means that there are no opportunities to inject context/environment via a templating language.
- The user interface interacts with the backend Flask application via JSON requests and responses - not forms. The external (json/form) API is described here
- SPAs are still browser based - so they have the same security vulnerabilities as traditional html/form-based applications.
- SPAs handle all routing/redirection via code, so redirects need context.

2.4.1 Configuration

An example configuration:

```
# no forms so no concept of flashing
SECURITY_FLASH_MESSAGES = False

# Need to be able to route backend flask API calls. Use 'accounts'
# to be the Flask-Security endpoints.
SECURITY_URL_PREFIX = '/api/accounts'

```

(continues on next page)

(continued from previous page)

```
# Turn on all the great Flask-Security features
SECURITY_RECOVERABLE = True
SECURITY_TRACKABLE = True
SECURITY_CHANGEABLE = True
SECURITY_CONFIRMABLE = True
SECURITY_REGISTERABLE = True
SECURITY_UNIFIED_SIGNIN = True

# These need to be defined to handle redirects
# As defined in the API documentation - they will receive the relevant context
SECURITY_POST_CONFIRM_VIEW = "/confirmed"
SECURITY_CONFIRM_ERROR_VIEW = "/confirm-error"
SECURITY_RESET_VIEW = "/reset-password"
SECURITY_RESET_ERROR_VIEW = "/reset-password-error"
SECURITY_REDIRECT_BEHAVIOR = "spa"

# CSRF protection is critical for all session-based browser UIs

# enforce CSRF protection for session / browser - but allow token-based
# API calls to go through
SECURITY_CSRF_PROTECT_MECHANISMS = ["session", "basic"]
SECURITY_CSRF_IGNORE_UNAUTH_ENDPOINTS = True

# Send Cookie with csrf-token. This is the default for Axios and Angular.
SECURITY_CSRF_COOKIE_NAME = "XSRF-TOKEN"
WTF_CSRF_CHECK_DEFAULT = False
WTF_CSRF_TIME_LIMIT = None

# In your app
# Enable CSRF on all api endpoints.
flask_wtf.CSRFProtect(app)

# Initialize Flask-Security
user_datastore = SQLAlchemyUserDatastore(db, User, Role)
security = Security(app, user_datastore)

# Optionally define and set unauthorized callbacks
security.unauthz_handler(<your unauth handler>)
```

When in development mode, the Flask application will run by default on port 5000. The UI might want to run on port 8080. In order to test redirects you need to set:

```
SECURITY_REDIRECT_HOST = 'localhost:8080'
```

Client side authentication options

Depending on your SPA architecture and vision you can choose between cookie or token based authentication.

For both there is more documentation and some examples. In both cases, you need to understand and handle *CSRF* concerns.

2.4.2 Security Considerations

Static elements such as your UI should be served with an industrial-grade web server - such as Nginx. This is also where various security measures should be handled such as injecting standard security headers such as:

- `Strict-Transport-Security`
- `X-Frame-Options`
- `Content Security Policy`
- `X-Content-Type-Options`
- `X-XSS-Protection`
- `Referrer policy`

There are a lot of different ways to host a SPA as the javascript part itself is quit easily hosted from any static webserver. A couple of deployment options and their configurations will be describer here.

2.4.3 Nginx

When serving a SPA from a Nginx webserver the Flask backend, with Flask-Security-Too, will probably be served via Nginx's reverse proxy feature. The javascript is served from Nginx itself and all calls to a certain path will be routed to the reversed proxy. The example below routes all http requests to *"/api/"* to the Flask backend and handles all other requests directly from javascript. This has a couple of benefits as all the requests happen within the same domain so you don't have to worry about CORS problems:

```
server {
    listen       80;
    server_name  www.example.com;

    #access_log  /var/log/nginx/host.access.log  main;

    root   /usr/share/nginx/html;
    index index.html;

    location / {
        try_files $uri $uri/ /index.html;
    }

    # Location of assets folder
    location ~ ^/(static)/  {
        gzip_static on;
        gzip_types text/plain text/xml text/css text/comma-separated-values
            text/javascript application/x-javascript application/atom+xml;
        expires max;
    }
```

(continues on next page)

(continued from previous page)

```
    # redirect server error pages to the static page /50x.html
    # 400 error's will be handled from the SPA
    error_page   500 502 503 504  /50x.html;
        location = /50x.html {
    }

    # route all api requests to the flask app, served by gunicorn
    location /api/ {
        proxy_pass http://localhost:8080/api/;
    }

    # OR served via uwsgi
    location /api/ {
        include ..../uwsgi_params;
        uwsgi_pass unix:/tmp/uwsgi.sock;
        uwsgi_pass_header AUTHENTICATION-TOKEN;
    }
}
```

Note: The example doesn't include SSL setup to keep it simple and still suitable for a more complex kubernetes setup where Nginx is often used as a load balancer and another Nginx with SSL setup runs in front of it.

2.4.4 Amazon lambda gateway / Serverless

Most Flask apps can be deployed to Amazon's lambda gateway without much hassle by using Zappa. You'll get automatic horizontal scaling, seamless upgrades, automatic SSL certificate renewal and a very cheap way of hosting a backend without being responsible for any infrastructure. Depending on how you design your app you could choose to host your backend from an api specific domain: e.g. *api.example.com*. When your SPA deployment structure is capable of routing the AJAX/XHR request from your javascript app to the separate backend; use it. When you want to use the backend from another e.g. *www.example.com* you have some deal with some CORS setup as your browser will block cross-domain POST requests. There is a Flask package for that: Flask-CORS.

The setup of CORS is simple:

```
CORS(
    app,
    supports_credentials=True,  # needed for cross domain cookie support
    resources="/*",
    allow_headers="*",
    origins="https://www.example.com",
    expose_headers="Authorization,Content-Type,Authentication-Token,XSRF-TOKEN",
)
```

You can then host your javascript app from an S3 bucket, with or without Cloudfront, GH-pages or from any static webserver.

Some background material:

- Specific to S3 but easily adaptable.
- Flask-Talisman - useful if serving everything from your Flask application - also useful as a good list of things to consider.

2.5 Security Patterns

Danger: Be aware that starting in Flask 2.2.0, they recommend extensions store context information on `g` which is the application context. Prior to this many extensions (including Flask-Security and Flask-Login) stored things like user credential information on the request context. These are now stored on `g` i.e. the application context. It is imperative that applications not mistakenly push their own application context and forget to pop it - in that case Flask won't push a new application context nor will it pop it at the end of the request - thus credential information could leak from one user request to another.

2.5.1 Authentication and Authorization

Flask-Security provides a set of authentication decorators:

- *auth_required()*
- *http_auth_required()*
- *auth_token_required()*

and a set of authorization decorators:

- *roles_required()*
- *roles_accepted()*
- *permissions_required()*
- *permissions_accepted()*

In addition, Flask-Login provides @login_required. In order to take advantage of all the Flask-Security features, it is recommended to NOT use @login_required.

Also, if you annotate your endpoints with JUST an authorization decorator, you will never get a 401 response, and (for forms) you won't be redirected to your login page. In this case you will always get a 403 status code (assuming you don't override the default handlers).

While these annotations are quick and easy, it is likely that they won't completely satisfy all an application's authorization requirements. A common example might be that a user can only edit their own posts/documents. In cases like this - it is nice to have a uniform way of handling all authorization errors. A simple way to do this is to use a special exception class that you can raise either in response to Flask-Security authorization failures, or in your own code. Then use Flask's `errorhandler` to catch that exception and create the appropriate API response:

```
Class MyForbiddenException(Exception):
    def __init__(self, msg='Not permitted with your privileges', status=http.HTTPStatus.
↪FORBIDDEN):
        self.info = {'status': status, 'msgs': [msg]}

_security = app.extensions["security"]

@_security.unauthz_handler
def my_unauthz_handler(func, params):
    raise MyForbiddenException()

@app.errorhandler(MyForbiddenException)
def my_exception(ex):
    return flask.jsonify(ex.info), ex.info['status']
```

(continues on next page)

(continued from previous page)

```

@app.route('/doc/<int:doc_id>', methods=['PATCH'])
@auth_required('token', 'session')
def doc_patch(doc_id):
    doc = fetch_doc(doc_id)
    if not current_user.has_role('admin') and doc.owner != current_user:
        raise MyForbiddenException(msg='You can only update docs you own')
```

A note about Basic Auth

Basic Auth is supported in Flask-Security, using the @http_auth_required() decorator. If a request for an endpoint protected with @http_auth_required is received, and the request doesn't contain the appropriate HTTP Headers, a 401 is returned along with the required WWW-Authenticate header. In this case there won't be a usable session cookie returned so all future requests will also require credentials to be sent. Effectively the caller is temporarily 'logged in' at the beginning of each request and 'logged out' again at the end of the request. Most (all?) browsers intercept this response and pop up a login dialog box and remember, for the site, the entered credentials. This effectively bypasses any of the normal Flask-Security login forms. By default, the Flask-Security endpoints that require the caller be authenticated do NOT support `basic` - however the *SECURITY_API_ENABLED_METHODS* can be used to override this.

Freshness

A common pattern for browser-based sites is to use sessions to manage identity. This is usually implemented using session cookies. These cookies expire once the session (browser tab) is closed. This is very convenient, and keeps the users from having to constantly re-authenticate. The downside is that sessions can easily be open for days or weeks. This adds to the security risk that some bad-actor or XSS gets control of the browser and then can do anything the user can. To mitigate that, operations that change fundamental identity characteristics (such as email, password, etc.) can be protected by requiring a 'fresh' or recent authentication. Flask-Security supports this with the following:

- *auth_required()* takes parameters that define how recent the authentication must have happened. In addition a grace period can be specified so that multiple step operations don't require re-authentication in the middle.
- A default *Security.reauthn_handler()* that is called when a request fails the recent authentication check.
- *SECURITY_VERIFY_URL* and *SECURITY_US_VERIFY_URL* endpoints that request the user to re-authenticate.
- `VerifyForm` and `UsVerifyForm` forms that can be extended.

Flask-Security itself uses this as part of securing the *Unified Sign In*, *Two-factor Authentication*, and *WebAuthn* setup endpoints.

2.5.2 Open Redirect Exposure

Flask-Security, accepts a `next=xx` parameter (either as a query param OR in the POSTed form) which it will use when completing an operation which results in a redirection. If a malicious user/ application can inject an arbitrary `next` parameter which redirects to an external location, this results in a security vulnerability called an *open redirect*. The following endpoints accept a `next` parameter:

```
- .login ("/login")
- .logout ("/logout")
- .register ("/register")
- .verify ("/verify")
- .two_factor_token_validation ("/tf-validate")
```

(continues on next page)

(continued from previous page)

```
- .wan_verify_response ("/wan-verify")
- .wan_signin_response ("/wan-signin")
- .us_signin ("/us-signin")
- .us_verify ("/us-verify")
```

Flask-Security attempts to verify that redirects are always relative. FS uses the standard Python library urlsplit() to parse the URL and verify that the `netloc` hasn't been altered. However, many browsers actually accept URLs that should be considered relative and perform various stripping and conversions that can cause them to be interpreted as absolute. A trivial example of this is:

/login?next=%20///github.com

This will pass the urlsplit() test that it is relative - but many browsers will simply strip off the space and interpret it as an absolute URL!

Prior to Werkzeug 2.1, Werkzeug set the response configuration variable `autocorrect_location_header = True` which forced the response *Location* header to always be an absolute path - thus effectively squashing any open redirect possibility. However since 2.1 it is now *False*.

Flask Security offers 2 mitigations for this via the *SECURITY_REDIRECT_VALIDATE_MODE* and *SECURITY_REDIRECT_VALIDATE_RE* configuration variables.

- The first mode - *"absolute"*, which is the default, is to once again set Werkzeug's `autocorrect_location_header` to `True`. Please note that this is set JUST for Flask-Security's blueprint - not all requests.
- With the second mode - *"regex"* - FS uses a regular expression to validate all `next` parameters to make sure they will be interpreted as *relative*. Be aware that the default regular expression is based on in-the-field testing and it is quite possible that there are other crafted relative URLs that could escape detection.

SECURITY_REDIRECT_VALIDATE_MODE actually takes a list - so both mechanisms can be specified.

2.5.3 Password Validation and Complexity

There is a large body of references (and endless discussions) around how to get users to create good passwords. The OWASP Authenication cheatsheet is a useful place to start. Flask-Security has a default password validator that:

- Checks for minimum and maximum length (minimum is configurable via *SECURITY_PASSWORD_LENGTH_MIN*). The default is 8 characters as defined by NIST.
- If *SECURITY_PASSWORD_CHECK_BREACHED* is set, will use the API for haveibeenpwned to check if the password is on a list of breached passwords. The configuration variable *SECURITY_PASSWORD_BREACHED_COUNT* can be used to set the minimum allowable 'breaches'.
- If *SECURITY_PASSWORD_COMPLEXITY_CHECKER* is set to `zxcvbn` and the package zxcvbn is installed, it will check the password for complexity.

Be aware that `zxcvbn` is not actively being maintained, and has localization issues.

In addition to validation, unicode passwords should be normalized as specified by NIST requirement: 5.1.1.2 Memorized Secret Verifiers. Normalization can be disabled by setting the *SECURITY_PASSWORD_NORMALIZE_FORM* to `None`. Validation and normalization is encapsulated in *PasswordUtil*. This can be overridden by passing your class at app initialization time. The *PasswordUtil.validate()* is passed additional kwargs to allow custom validators more flexibility. A custom validator can still call the underlying methods where appropriate:

flask_security.password_length_validator(), *flask_security.password_complexity_validator()*, and *flask_security.password_breached_validator()*.

2.5.4 Generic Responses - Avoiding User Enumeration

How an application responds to API requests that contain identity or authentication information can give would-be attackers insight into active users on the system. OWASP has a great cheat-sheet describing this and useful ways to avoid it. Flask-Security supports this by setting the *SECURITY_RETURN_GENERIC_RESPONSES* configuration to `True`. As documented in the cheat-sheet - this does come with some usability concerns. The following endpoints are affected:

- *SECURITY_REGISTER_URL* - The same response will be returned whether the email (or username) is already in the system or not. JSON requests will ALWAYS return 200. If *SECURITY_CONFIRMABLE* is set (it should be!), the *SECURITY_MSG_CONFIRM_REGISTRATION* message will be flashed for both new and existing email addresses. Detailed errors will still be returned for things like insufficient password complexity, etc.. In the case of trying to register an existing email, an email will be sent to that email address explaining that they are already registered and displaying the associated username (if any) and provide a hint on how to reset their password if they forgot it. In the case of a new email but an already registered username, an email will be sent saying that the user must try registering again with a different username.
- *SECURITY_LOGIN_URL* - For any errors (unknown username, inactive account, bad password) the *SECURITY_MSG_GENERIC_AUTHN_FAILED* message will be returned.
- *SECURITY_RESET_URL* - In all cases the *SECURITY_MSG_PASSWORD_RESET_REQUEST* message will be flashed. For JSON a 200 will always be returned (whether an email was sent or not). `Note`: If the application overrides the form and adds an additional field (e.g. *captcha*) and that field has a validation error, a normal form error response will be returned (and JSON will return a 400).
- *SECURITY_CONFIRM_URL* - In all cases the *SECURITY_MSG_CONFIRMATION_REQUEST* message will be flashed. For JSON a 200 will always be returned (whether an email was sent or not). `Note`: If the application overrides the form and adds an additional field (e.g. *captcha*) and that field has a validation error, a normal form error response will be returned (and JSON will return a 400).
- *SECURITY_US_SIGNIN_SEND_CODE_URL* - The *SECURITY_MSG_GENERIC_US_SIGNIN* message will be flashed in all cases - whether a selected method is setup for the user or not.
- *SECURITY_US_SIGNIN_URL* - For any errors (unknown username, inactive account, bad passcode) the *SECURITY_MSG_GENERIC_AUTHN_FAILED* message will be returned.
- *SECURITY_US_VERIFY_LINK_URL* - For any errors (unknown username, inactive account, bad passcode) the *SECURITY_MSG_GENERIC_AUTHN_FAILED* message will be returned.

In the case of an application using a `username` as an identity it should be noted that it is possible for a bad-actor to enumerate usernames, albeit slowly, by parsing emails.

Note also that *SECURITY_REQUIRES_CONFIRMATION_ERROR_VIEW* is ignored in these cases. If your application is using WebAuthn, be sure to set *SECURITY_WAN_ALLOW_USER_HINTS* to `False`.

2.5.5 CSRF

By default, Flask-Security, via Flask-WTForms protects all form based POSTS from CSRF attacks using well vetted per-session hidden-form-field csrf-tokens.

Any web application that relies on session cookies for authentication must have CSRF protection. For more details please read this OWASP CSRF cheatsheet. A couple important take-aways - first - it isn't about forms versus JSON - it is about how the API is authenticated (session cookies versus authentication token). Second there is the concern about 'login CSRF' - is protection needed prior to authentication (yes if you have a really secure/popular site).

Flask-Security strives to support various options for both its endpoints (e.g. `/login`) and the application endpoints (protected with Flask-Security decorators such as *`auth_required()`*).

If your application just uses forms that are derived from `Flask-WTF::Flaskform` - you are done.

CSRF: Single-Page-Applications and AJAX/XHR

If you are thinking about using authentication tokens in your browser-based UI - read this article on how and where to store authentication tokens. While the article is talking about JWT it applies to Flask-Security tokens as well.

In general, it is considered more secure (and easier) to use sessions for browser based UI, and tokens for service to service and scripts.

For SPA, and especially those that aren't served via your flask application, there are difficulties with actually retrieving and using a CSRF token. There are 2 normal ways to do this:

- Have the csrf-token available via a JSON GET request that can be attached as a header in every mutating request.
- Have a cookie that can be read via javascript whose value is the csrf-token that can be attached as a header in every mutating request.

Flask-Security supports both solutions.

Explicit fetch and send of csrf-token

The current session CSRF token is returned on every JSON GET request (to a Flask-Security endpoint) as `response['csrf_token`]`. For web applications that ARE served via flask, it is even easier to get the csrf-token - https://flask-wtf.readthedocs.io/en/1.0.x/csrf/ gives some useful tips.

Armed with the csrf-token, the UI must include that in every mutating operation. Be careful NOT to include the csrf-token in non-mutating requests (such as GETs). If your application uses GET to actually modify state - please stop.

An example using axios

```
# This will fetch the csrf-token. Note that we do a GET on the login endpoint
# which will get us the csrf-token even though we aren't yet logged in.
# Note further the 'data: null' and explicit Content-Type header - these are
# critical, otherwise Flask-Security will return the login form.
axios.get('/login',{data: null, headers: {'Content-Type': 'application/json'}}).
↪then(function (resp) {
  csrf_token = resp.data['response']['csrf_token']
})


# This will add the token header to each outgoing mutating request.
axios.interceptors.request.use(function (config) {
```

(continues on next page)

(continued from previous page)

```
  if (["post", "delete", "patch", "put"].includes(config["method"])) {
    if (csrf_token !== '') {
      config.headers["X-CSRF-Token"] = csrf_token
    }
  }
  return config;
}, function (error) {
  // Do something with request error
  return Promise.reject(error);
});
```

Note that we use the header name `X-CSRF-Token` as that is one of the default headers configured in Flask-WTF (*WTF_CSRF_HEADERS*)

To protect your application's endpoints (that presumably are not using Flask forms), you need to enable CSRF as described in the FlaskWTF documentation:

```
flask_wtf.CSRFProtect(app)
```

This will turn on CSRF protection on ALL endpoints, including Flask-Security. This protection differs slightly from the default that is part of FlaskForm in that it will first look at the request body and see if it can find a form field that contains the csrf-token, and if it can't, it will check if the request has a header that is listed in *WTF_CSRF_HEADERS* and use that. Be aware that if you enable this it will ONLY work if you send the session cookie on each request.

Note: It is IMPORTANT that you initialize/call `CSRFProtect` PRIOR to initializing Flask_Security.

Using a Cookie

You can instruct Flask-Security to send a cookie that contains the csrf token. This can be very convenient since various javascript AJAX packages are pre-configured to extract the contents of a cookie and send it on every mutating request as an HTTP header. axios for example has a default configuration that it will look for a cookie named `XSRF-TOKEN` and will send the contents of that back in an HTTP header called `X-XSRF-Token`. This means that if you use that package you don't need to make any changes to your UI and just need the following configuration:

```
# Have cookie sent
app.config["SECURITY_CSRF_COOKIE_NAME"] = "XSRF-TOKEN"

# Don't have csrf tokens expire (they are invalid after logout)
app.config["WTF_CSRF_TIME_LIMIT"] = None

# You can't get the cookie until you are logged in.
app.config["SECURITY_CSRF_IGNORE_UNAUTH_ENDPOINTS"] = True

# Enable CSRF protection
flask_wtf.CSRFProtect(app)
```

Angular's httpClient also supports this.

For React based projects you are free to choose your http client (*fetch* is bundled by default). Retrieving the token is easy:

```
fetch(url, {
  credentials: 'include',
  mode: 'cors',
  headers: {
    'Accept': 'application/json',
    'X-XSRF-TOKEN': getCookieValue('XSRF-TOKEN')
  }
});
```

Sending the token on every mutating request is something that you should implement yourself. As an example an API call to an API endpoint that does CSRF validation:

```
function addUser(details) {
  return fetch('https://api.example.com/user', {
    mode: 'cors',
    method: 'POST',
    credentials: 'include',
    body: JSON.stringify(details),
    headers: {
      'Content-Type': 'application/json',
      'Accept': 'application/json',
      'X-XSRF-TOKEN': getCookieValue('XSRF-TOKEN')
    }
  }).then(response => {
    return response.json().then(data => {
      if (response.ok) {
        return data;
      } else {
        return Promise.reject({status: response.status, data});
      }
    });
  });
}
```

When you have axios setup correctly, this is a lot easier:

```
function addUser(details) {
  return axios.post('https://api.example.com/user', details);
}
```

CSRF: Enable protection for session auth, but not token auth

As mentioned above, CSRF is critical for any mutating operation where the authentication credentials are 'invisibly' sent - such as a session cookie - from a browser. But if your endpoint a) can only be authenticated with an attached token or b) can be called either via session OR token; it is often desirable not to force token API users to deal with CSRF. To solve this, we need to keep CSRFProtect from checking the csrf-token early in the request and instead defer that decision to later decorators/code. Flask-Security's authentication decorators (*auth_required()*, *auth_token_required()*, and *http_auth_required()* all support calling csrf protection based on configuration:

```
# Disable pre-request CSRF
app.config[WTF_CSRF_CHECK_DEFAULT] = False

```

(continues on next page)

(continued from previous page)

```
# Check csrf for session and http auth (but not token)
app.config[SECURITY_CSRF_PROTECT_MECHANISMS] = ["session", "basic"]

# Enable CSRF protection
flask_wtf.CSRFProtect(app)

@app.route("/")
@auth_required("token", "session")
def home_page():
```

With this configuration, CSRF won't be required if the caller uses an authentication token, but if it uses the session cookie it will.

CSRF: Pro-Tips

1) Be aware that for CSRF to work, callers MUST send the session cookie. So for pure API (token based), and no session cookie - there is no way to support 'login CSRF'. So your app must set *SECURITY_CSRF_IGNORE_UNAUTH_ENDPOINTS* (or clients must use CSRF/session cookie for logging in then once they have an authentication token, no further need for cookie).
2) If you enable CSRFProtect(app) and you want to support non-form based JSON requests, then you must include the CSRF token in the header (e.g. X-CSRF-Token)
3) You must enable CSRFProtect(app) if you want to accept the CSRF token in the request header.
4) Annotate each of your endpoints with a @auth_required decorator (and don't rely on just a @role_required or @login_required decorator) so that Flask-Security gets control at the appropriate place.
5) If you can't use a decorator, Flask-Security exposes the underlying method *flask_security.handle_csrf()*.
6) Consider starting by setting *SECURITY_CSRF_IGNORE_UNAUTH_ENDPOINTS* to True. Your application likely doesn't need 'login CSRF' protection, and it is frustrating to not even be able to login via API!
7) If you have unauthenticated endpoints that you want to protect with CSRF then use the *flask_security.unauth_csrf()* decorator.

CHAPTER

THREE

API

3.1 API

The external (json/form) API is described here

3.1.1 Core

class flask_security.**Security**(*app=None*, *datastore=None*, *register_blueprint=True*, *login_form=<class 'flask_security.forms.LoginForm'>*, *verify_form=<class 'flask_security.forms.VerifyForm'>*, *confirm_register_form=<class 'flask_security.forms.ConfirmRegisterForm'>*, *register_form=<class 'flask_security.forms.RegisterForm'>*, *forgot_password_form=<class 'flask_security.forms.ForgotPasswordForm'>*, *reset_password_form=<class 'flask_security.forms.ResetPasswordForm'>*, *change_password_form=<class 'flask_security.forms.ChangePasswordForm'>*, *send_confirmation_form=<class 'flask_security.forms.SendConfirmationForm'>*, *passwordless_login_form=<class 'flask_security.forms.PasswordlessLoginForm'>*, *two_factor_verify_code_form=<class 'flask_security.forms.TwoFactorVerifyCodeForm'>*, *two_factor_setup_form=<class 'flask_security.forms.TwoFactorSetupForm'>*, *two_factor_rescue_form=<class 'flask_security.forms.TwoFactorRescueForm'>*, *two_factor_select_form=<class 'flask_security.tf_plugin.TwoFactorSelectForm'>*, *mf_recovery_codes_form=<class 'flask_security.recovery_codes.MfRecoveryCodesForm'>*, *mf_recovery_form=<class 'flask_security.recovery_codes.MfRecoveryForm'>*, *us_signin_form=<class 'flask_security.unified_signin.UnifiedSigninForm'>*, *us_setup_form=<class 'flask_security.unified_signin.UnifiedSigninSetupForm'>*, *us_setup_validate_form=<class 'flask_security.unified_signin.UnifiedSigninSetupValidateForm'>*, *us_verify_form=<class 'flask_security.unified_signin.UnifiedVerifyForm'>*, *wan_register_form=<class 'flask_security.webauthn.WebAuthnRegisterForm'>*, *wan_register_response_form=<class 'flask_security.webauthn.WebAuthnRegisterResponseForm'>*, *wan_signin_form=<class 'flask_security.webauthn.WebAuthnSigninForm'>*, *wan_signin_response_form=<class 'flask_security.webauthn.WebAuthnSigninResponseForm'>*, *wan_delete_form=<class 'flask_security.webauthn.WebAuthnDeleteForm'>*, *wan_verify_form=<class 'flask_security.webauthn.WebAuthnVerifyForm'>*, *anonymous_user=None*, *mail_util_cls=<class 'flask_security.mail_util.MailUtil'>*, *password_util_cls=<class 'flask_security.password_util.PasswordUtil'>*, *phone_util_cls=<class 'flask_security.phone_util.PhoneUtil'>*, *render_template=<function default_render_template>*, *totp_cls=<class 'flask_security.totp.Totp'>*, *username_util_cls=<class 'flask_security.username_util.UsernameUtil'>*, *webauthn_util_cls=<class 'flask_security.webauthn_util.WebauthnUtil'>*, *mf_recovery_codes_util_cls=<class 'flask_security.recovery_codes.MfRecoveryCodesUtil'>*, *oauth=None*, ***kwargs*)

The `Security` class initializes the Flask-Security extension.

Parameters

- **app** (*flask.Flask* | *None*) – The application.
- **datastore** (`UserDatastore` | *None*) – An instance of a user datastore.
- **register_blueprint** (*bool*) – to register the Security blueprint or not.

- **login_form** (*Type*[LoginForm]) – set form for the login view
- **verify_form** (*Type*[VerifyForm]) – set form for re-authentication due to freshness check
- **register_form** (*Type*[RegisterForm]) – set form for the register view when *SECURITY_CONFIRMABLE* is false
- **confirm_register_form** (*Type*[ConfirmRegisterForm]) – set form for the register view when *SECURITY_CONFIRMABLE* is true
- **forgot_password_form** (*Type*[ForgotPasswordForm]) – set form for the forgot password view
- **reset_password_form** (*Type*[ResetPasswordForm]) – set form for the reset password view
- **change_password_form** (*Type*[ChangePasswordForm]) – set form for the change password view
- **send_confirmation_form** (*Type*[SendConfirmationForm]) – set form for the send confirmation view
- **passwordless_login_form** (*Type*[PasswordlessLoginForm]) – set form for the passwordless login view
- **two_factor_setup_form** (*Type*[TwoFactorSetupForm]) – set form for the 2FA setup view
- **two_factor_verify_code_form** (*Type*[TwoFactorVerifyCodeForm]) – set form the the 2FA verify code view
- **two_factor_rescue_form** (*Type*[TwoFactorRescueForm]) – set form for the 2FA rescue view
- **two_factor_select_form** (*Type*[TwoFactorSelectForm]) – set form for selecting between active 2FA methods
- **mf_recovery_codes_form** (*Type*[MfRecoveryCodesForm]) – set form for retrieving and setting recovery codes
- **mf_recovery_form** (*Type*[MfRecoveryForm]) – set form for multi factor recovery
- **us_signin_form** (*Type*[UnifiedSigninForm]) – set form for the unified sign in view
- **us_setup_form** (*Type*[UnifiedSigninSetupForm]) – set form for the unified sign in setup view
- **us_setup_validate_form** (*Type*[UnifiedSigninSetupValidateForm]) – set form for the unified sign in setup validate view
- **us_verify_form** (*Type*[UnifiedVerifyForm]) – set form for re-authenticating due to freshness check
- **wan_register_form** (*Type*[WebAuthnRegisterForm]) – set form for registering a webauthn security key
- **wan_register_response_form** (*Type*[WebAuthnRegisterResponseForm]) – set form for registering a webauthn security key
- **wan_signin_form** (*Type*[WebAuthnSigninForm]) – set form for authenticating with a webauthn security key
- **wan_signin_response_form** (*Type*[WebAuthnSigninResponseForm]) – set form for authenticating with a webauthn

- **wan_delete_form** (*Type*[WebAuthnDeleteForm]) – set form for deleting a webauthn security key
- **wan_verify_form** (*Type*[WebAuthnVerifyForm]) – set form for using a webauthn key to verify authenticity
- **anonymous_user** (*Type[flask_login.AnonymousUserMixin] | None*) – class to use for anonymous user
- **mail_util_cls** (*Type*[MailUtil]) – Class to use for sending emails. Defaults to *MailUtil*
- **password_util_cls** (*Type*[PasswordUtil]) – Class to use for password normalization/validation. Defaults to *PasswordUtil*
- **phone_util_cls** (*Type*[PhoneUtil]) – Class to use for phone number utilities. Defaults to *PhoneUtil*
- **render_template** (*Callable[[...], str]*) – function to use to render templates. The default is Flask's render_template() function.
- **totp_cls** (*Type*[Totp]) – Class to use as TOTP factory. Defaults to *Totp*
- **username_util_cls** (*Type*[UsernameUtil]) – Class to use for normalizing and validating usernames. Defaults to *UsernameUtil*
- **webauthn_util_cls** (*Type*[WebauthnUtil]) – Class to use for customizing WebAuthn registration and signin. Defaults to *WebauthnUtil*
- **mf_recovery_codes_util_cls** (*Type*[MfRecoveryCodesUtil]) – Class for generating, checking, encrypting and decrypting recovery codes. Defaults to *MfRecoveryCodesUtil*
- **oauth** (*OAuth | None*) – An instance of authlib.integrations.flask_client.OAuth
- **kwargs** (*Any*) –

Tip: Be sure that all your configuration values have been set PRIOR to instantiating this class. Some configuration values are set as attributes on the instance and therefore won't track any changes.

New in version 3.4.0: `verify_form` added as part of freshness/re-authentication

New in version 3.4.0: `us_signin_form`, `us_setup_form`, `us_setup_validate_form`, and `us_verify_form` added as part of the *Unified Sign In* feature.

New in version 3.4.0: `totp_cls` added to enable applications to implement replay protection - see *Totp*.

New in version 3.4.0: `phone_util_cls` added to allow different phone number parsing implementations - see *PhoneUtil*

New in version 4.0.0: `mail_util_cls` added to isolate mailing handling. `password_util_cls` added to encapsulate password validation/normalization.

New in version 4.1.0: `username_util_cls` added to encapsulate username handling.

New in version 5.0.0: `wan_register_form`, `wan_register_response_form`, `webauthn_signin_form`, `wan_signin_response_form`, `webauthn_delete_form`, `webauthn_verify_form`, `tf_select_form`.

New in version 5.0.0: `WebauthnUtil` class.

New in version 5.0.0: Added support for multi-factor recovery codes `mf_recovery_codes_form`, `mf_recovery_form`.

New in version 5.1.0: `mf_recovery_codes_util_cls`, `oauth`

Deprecated since version 4.0.0: `send_mail` and `send_mail_task`. Replaced with `mail_util_cls`. `two_factor_verify_password_form` removed. `password_validator` removed in favor of the new `password_util_cls`.

Deprecated since version 5.0.0: Passing in a LoginManager instance. Removed in 5.1.0

Deprecated since version 5.0.0: json_encoder_cls is no longer honored since Flask 2.2 has deprecated it.

Deprecated since version 5.3.1: Passing in an anonymous_user class.

`init_app`(*app*, *datastore=None*, *register_blueprint=None*, ***kwargs*)

Initializes the Flask-Security extension for the specified application and datastore implementation.

Parameters

- **app** (*flask.Flask*) – The application.
- **datastore** (`UserDatastore` | *None*) – An instance of a user datastore.
- **register_blueprint** (*bool* | *None*) – to register the Security blueprint or not.
- **kwargs** (*Any*) – Can be used to override/initialize any of the constructor attributes.

Return type

None

If you create the Security instance with both an 'app' and 'datastore' you shouldn't call this - it will be called as part of the constructor.

`reauthn_handler`(*cb*)

Callback when endpoint required a fresh authentication. This is called by *auth_required()*.

Parameters

cb (*Callable[[timedelta, timedelta], ResponseValue]*) – Callback function with signature (within, grace)

within
timedelta that endpoint required fresh authentication within.

grace
timedelta of grace period that endpoint allowed.

Return type

None

Should return a Response or something Flask can create a Response from. Can raise an exception if it is handled as part of `flask.errorhandler(<exception>)`

The default implementation will return a 401 response if the request was JSON, otherwise will redirect to *SECURITY_US_VERIFY_URL* (if *SECURITY_UNIFIED_SIGNIN* is enabled) else to *SECURITY_VERIFY_URL*. If both of those are None it sends an `abort(401)`.

See *flask_security.auth_required()* for details about freshness checking.

New in version 3.4.0.

`render_json`(*cb*)

Callback to render response payload as JSON.

Parameters

cb (*Callable[[Dict[str, Any], int, Dict[str, str] | None, User | None], ResponseValue]*) – Callback function with signature (payload, code, headers=None, user=None)

payload
A dict. Please see the formal API spec for details.

code
Http status code

headers
Headers object

user
the UserDatastore object (or None). Note that this is usually the same as current_user - but not always.

Return type
None

The default implementation simply returns:

```
headers["Content-Type"] = "application/json"
payload = dict(meta=dict(code=code), response=payload)
return make_response(jsonify(payload), code, headers)
```

Important: Note that this has nothing to do with how the response is serialized. That is controlled by Flask and starting with Flask 2.2 that is managed by sub-classing Flask::JSONProvider. Flask-Security does this to add serializing lazy-strings.

This can be used by applications to unify all their JSON API responses. This is called in a request context and should return a Response or something Flask can create a Response from.

New in version 3.3.0.

set_form_info(*name*, *form_info*)
Set form instantiation info.

Parameters

- **name** (*str*) – Name of form.
- **form_info** (FormInfo) – see *FormInfo*

Return type
None

Advanced

Forms (which are all FlaskForms) are instantiated at the start of each request. Normally this is done as part of a view by simply calling the form class constructor - Flask-WTForms handles filling it in from various request attributes.

The form classes themselves can be extended (e.g. to add or change fields) and the derived class can be set at *Security* constructor time, *init_app* time, or using this method.

This default implementation is suitable for most applications.

Some application might want to control the instantiation of forms, for example to be able to inject additional validation services. Using this method, a callable *instantiator* can be set that Flask-Security will call to return a properly instantiated form.

> **Danger:** Do not perform any validation as part of instantiation - many views have a bunch of logic PRIOR to calling the form validator.
>
> New in version 5.1.0.

---

unauthn_handler(*cb*)

Callback for failed authentication. This is called by *auth_required()*, *auth_token_required()* or *http_auth_required()* if authentication fails.

Parameters

cb (*Callable[[List[str], Dict[str, str] | None], ResponseValue]*) – Callback function with signature (mechanisms, headers=None)

mechanisms
List of which authentication mechanisms were tried

headers
dict of headers to return

Return type
None

Should return a Response or something Flask can create a Response from. Can raise an exception if it is handled as part of `flask.errorhandler(<exception>)`

The default implementation will return a 401 response if the request was JSON, otherwise lets `flask_login.login_manager.unauthorized()` handle redirects.

New in version 3.3.0.

unauthz_handler(*cb*)

Callback for failed authorization. This is called by the *roles_required()*, *roles_accepted()*, *permissions_required()*, or *permissions_accepted()* if a role or permission is missing.

Parameters

cb (*Callable[[str, List[str] | None], ResponseValue]*) – Callback function with signature (func, params)

func_name
the decorator function name (e.g. 'roles_required')

params
list of what (if any) was passed to the decorator.

Return type
None

Should return a Response or something Flask can create a Response from. Can raise an exception if it is handled as part of flask.errorhandler(<exception>)

With the passed parameters the application could deliver a concise error message.

New in version 3.3.0.

Changed in version 5.1.0: Pass in the function name, not the function!

want_json(*fn*)

Function that returns True if response should be JSON (based on the request)

Parameters
`fn` (*Callable[[flask.Request], bool]*) – Function with the following signature (request)

request
Werkzeug/Flask request

Return type
None

The default implementation returns True if either the Content-Type is "application/json" or the best Accept header value is "application/json".

New in version 3.3.0.

`flask_security.`**`current_user`**

A proxy for the current user.

3.1.2 Protecting Views

`flask_security.`**`anonymous_user_required`**(*f*)

Decorator which requires that caller NOT be logged in. If a logged in user accesses an endpoint protected with this decorator they will be redirected to the *SECURITY_POST_LOGIN_VIEW*. If the caller requests a JSON response, a 400 will be returned.

Changed in version 3.3.0: Support for JSON response was added.

Parameters
`f` (*Callable[[...], Any]*) –

Return type
Callable[[. . .], *Any*]

`flask_security.`**`http_auth_required`**(*realm*)

Decorator that protects endpoints using Basic HTTP authentication.

Parameters
`realm` (*Any*) – optional realm name

Return type
Callable[[. . .], *Any*]

If authentication fails, then a 401 with the 'WWW-Authenticate' header set will be returned.

Once authenticated, if so configured, CSRF protection will be tested.

`flask_security.`**`auth_token_required`**(*fn*)

Decorator that protects endpoints using token authentication. The token should be added to the request by the client by using a query string variable with a name equal to the configuration value of *SECURITY_TOKEN_AUTHENTICATION_KEY* or in a request header named that of the configuration value of *SECURITY_TOKEN_AUTHENTICATION_HEADER*

Once authenticated, if so configured, CSRF protection will be tested.

Parameters
`fn` (*Callable[[...], Any]*) –

Return type
Callable[[. . .], *Any*]

`flask_security.`**`auth_required`**(**auth_methods*, *within=-1*, *grace=None*)

Decorator that protects endpoints through multiple mechanisms. Example:

```
@app.route('/dashboard')
@auth_required('token', 'session')
def dashboard():
    return 'Dashboard'
```

Parameters

- **`auth_methods`** (*`str | Callable[[], List[str]] | None`*) – Specified mechanisms (token, basic, session). If not specified then all current available mechanisms (except "basic") will be tried. A callable can also be passed (useful if you need app/request context). The callable must return a list.
- **`within`** (*`int | float | Callable[[], timedelta]`*) – Add 'freshness' check to authentication. Is either an int specifying # of minutes, or a callable that returns a timedelta. For timedeltas, timedelta.total_seconds() is used for the calculations:
  - If > 0, then the caller must have authenticated within the time specified (as measured using the session cookie).
  - If 0 and not within the grace period (see below) the caller will always be redirected to re-authenticate.
  - If < 0 (the default) no freshness check is performed.

  Note that Basic Auth, by definition, is always 'fresh' and will never result in a redirect/error.
- **`grace`** (*`int | float | Callable[[], timedelta] | None`*) – Add a grace period for freshness checks. As above, either an int or a callable returning a timedelta. If not specified then *`SECURITY_FRESHNESS_GRACE_PERIOD`* is used. The grace period allows callers to complete the required operations w/o being prompted again. See *`flask_security.check_and_update_authn_fresh()`* for details.

Return type

Callable[[. . .], *Any*]

Note that regardless of order specified - they will be tried in the following order: token, session, basic.

The first mechanism that succeeds is used, following that, depending on configuration, CSRF protection will be tested.

On authentication failure *.Security.unauthorized_callback* (deprecated) or *`Security.unauthn_handler()`* will be called.

As a side effect, upon successful authentication, the request global

`fs_authn_via` will be set to the method ("basic", "token", "session")

Note: If "basic" is specified in addition to other methods, then if authentication fails, a 401 with the "WWW-Authenticate" header will be returned - rather than being redirected to the login view.

Changed in version 3.3.0: If `auth_methods` isn't specified, then all will be tried. Authentication mechanisms will always be tried in order of `token`, `session`, `basic` regardless of how they are specified in the `auth_methods` parameter.

Changed in version 3.4.0: Added `within` and `grace` parameters to enforce a freshness check.

Changed in version 3.4.4: If `auth_methods` isn't specified try all mechanisms EXCEPT `basic`.

Changed in version 4.0.0: auth_methods can be passed as a callable.

`flask_security.`**`login_required`**`(`*`func`*`)`

If you decorate a view with this, it will ensure that the current user is logged in and authenticated before calling the actual view. (If they are not, it calls the `LoginManager.unauthorized` callback.) For example:

```
@app.route('/post')
@login_required
def post():
    pass
```

If there are only certain times you need to require that your user is logged in, you can do so with:

```
if not current_user.is_authenticated:
    return current_app.login_manager.unauthorized()
```

. . . which is essentially the code that this function adds to your views.

It can be convenient to globally turn off authentication when unit testing. To enable this, if the application configuration variable *LOGIN_DISABLED* is set to *True*, this decorator will be ignored.

Note: Per W3 guidelines for CORS preflight requests, HTTP `OPTIONS` requests are exempt from login checks.

Parameters
func (*function*) – The view function to decorate.

`flask_security.`**`roles_required`**`(`*`*roles`*`)`

Decorator which specifies that a user must have all the specified roles. Example:

```
@app.route('/dashboard')
@roles_required('admin', 'editor')
def dashboard():
    return 'Dashboard'
```

The current user must have both the *admin* role and *editor* role in order to view the page.

Parameters
roles (*str*) – The required roles.

Return type
Callable[[. . .], *Any*]

`flask_security.`**`roles_accepted`**`(`*`*roles`*`)`

Decorator which specifies that a user must have at least one of the specified roles. Example:

```
@app.route('/create_post')
@roles_accepted('editor', 'author')
def create_post():
    return 'Create Post'
```

The current user must have either the *editor* role or *author* role in order to view the page.

Parameters
roles (*str*) – The possible roles.

Return type
Callable[[. . .], *Any*]

flask_security.**permissions_required**(**fsperms*)

Decorator which specifies that a user must have all the specified permissions. Example:

```
@app.route('/dashboard')
@permissions_required('admin-write', 'editor-write')
def dashboard():
    return 'Dashboard'
```

The current user must have BOTH permissions (via the roles it has) to view the page.

N.B. Don't confuse these permissions with flask-principle Permission()!

Parameters
fsperms (*str*) – The required permissions.

Return type
Callable[[. . .], *Any*]

New in version 3.3.0.

flask_security.**permissions_accepted**(**fsperms*)

Decorator which specifies that a user must have at least one of the specified permissions. Example:

```
@app.route('/create_post')
@permissions_accepted('editor-write', 'author-wrote')
def create_post():
    return 'Create Post'
```

The current user must have one of the permissions (via the roles it has) to view the page.

N.B. Don't confuse these permissions with flask-principle Permission()!

Parameters
fsperms (*str*) – The possible permissions.

Return type
Callable[[. . .], *Any*]

New in version 3.3.0.

flask_security.**unauth_csrf**(*fall_through=False*)

Decorator for endpoints that don't need authentication but do want CSRF checks (available via Header rather than just form). This is required when setting *WTF_CSRF_CHECK_DEFAULT* = **False** since in that case, without this decorator, the form validation will attempt to do the CSRF check, and that will fail since the csrf-token is in the header (for pure JSON requests).

This decorator does nothing unless Flask-WTF::CSRFProtect has been initialized.

This decorator does nothing if *WTF_CSRF_ENABLED* == **False**.

This decorator will always require CSRF if the caller is authenticated.

This decorator will suppress CSRF if caller isn't authenticated and has set the *SECURITY_CSRF_IGNORE_UNAUTH_ENDPOINTS* config variable.

Parameters
fall_through (*bool*) – if set to True, then if CSRF fails here - simply keep going. This is appropriate if underlying view is form based and once the form is instantiated, the csrf_token will be available. Note that this can mask some errors such as 'The CSRF session token is

missing.' meaning that the caller didn't send a session cookie and instead the caller might get a 'The CSRF token is missing.' error.

Return type
Callable[[...], *Any*]

New in version 3.3.0.

`flask_security.handle_csrf(method)`

Invoke CSRF protection based on authentication method.

Usually this is called as part of a decorator, but if that isn't appropriate, endpoint code can call this directly.

If CSRF protection is appropriate, this will call flask_wtf::protect() which will raise a ValidationError on CSRF failure.

This routine does nothing if any of these are true:

1) *WTF_CSRF_ENABLED* is set to False
2) the Flask-WTF CSRF module hasn't been initialized
3) csrfProtect already checked and accepted the token

If the passed in method is not in *SECURITY_CSRF_PROTECT_MECHANISMS* then not only will no CSRF code be run, but a flag in the current context `fs_ignore_csrf` will be set so that downstream code knows to ignore any CSRF checks.

New in version 3.3.0.

Parameters
method (*str | None*) –

Return type
None

3.1.3 User Object Helpers

`class flask_security.UserMixin`

Mixin for *User* model definitions

`calc_username()`

Come up with the best 'username' based on how the app is configured (via *SECURITY_USER_IDENTITY_ATTRIBUTES*). Returns the first non-null match (and converts to string). In theory this should NEVER be the empty string unless the user record isn't actually valid.

New in version 3.4.0.

Return type
str

`get_auth_token()`

Constructs the user's authentication token.

Raises
`ValueError` – If `fs_token_uniquifier` is part of model but not set.

Return type
str | bytes

Optionally use a separate uniquifier so that changing password doesn't invalidate auth tokens.

This data MUST be securely signed using the `remember_token_serializer`

Changed in version 4.0.0: If user model has `fs_token_uniquifier` - use that (raise ValueError if not set). Otherwise fallback to using `fs_uniquifier`.

`get_id()`

Returns the user identification attribute. 'Alternative-token' for Flask-Login. This is always `fs_uniquifier`.

New in version 3.4.0.

Return type
str

`get_redirect_qparams`(*existing=None*)

Return user info that will be added to redirect query params.

Parameters
`existing` (`Dict[str, Any] | None`) – A dict that will be updated.

Returns
A dict whose keys will be query params and values will be query values.

Return type
Dict[str, *Any*]

The returned dict will always have an 'identity' key/value. If the User Model contains 'email', an 'email' key/value will added. All keys provided in 'existing' will also be merged in.

New in version 3.2.0.

Changed in version 4.0.0: Add 'identity' using UserMixin.calc_username() - email is optional.

`get_security_payload()`

Serialize user object as response payload. Override this to return any/all of the user object in JSON responses. Return a dict.

Return type
Dict[str, *Any*]

`has_permission`(*permission*)

Returns *True* if user has this permission (via a role it has).

Parameters
`permission` (`str`) – permission string name

Return type
bool

New in version 3.3.0.

`has_role`(*role*)

Returns *True* if the user identifies with the specified role.

Parameters
`role` (`str | Role`) – A role name or *Role* instance

Return type
bool

property is_active: bool

Returns *True* if the user is active.

tf_send_security_token(*method*, ***kwargs*)

Generate and send the security code for two-factor.

Parameters

- **method** (*str*) – The method in which the code will be sent
- **kwargs** (*Any*) – Opaque parameters that are subject to change at any time

Returns

None if successful, error message if not.

Return type

str | None

This is a wrapper around *tf_send_security_token()* that can be overridden to manage any errors.

New in version 3.4.0.

us_send_security_token(*method*, ***kwargs*)

Generate and send the security code for unified sign in.

Parameters

- **method** (*str*) – The method in which the code will be sent
- **kwargs** (*Any*) – Opaque parameters that are subject to change at any time

Returns

None if successful, error message if not.

Return type

str | None

This is a wrapper around *us_send_security_token()* that can be overridden to manage any errors.

New in version 3.4.0.

verify_and_update_password(*password*)

Returns `True` if the password is valid for the specified user.

Additionally, the hashed password in the database is updated if the hashing algorithm happens to have changed.

N.B. you MUST call DB commit if you are using a session-based datastore (such as SqlAlchemy) since the user instance might have been altered (i.e. `app.security.datastore.commit()`). This is usually handled in the view.

Parameters

password (*str*) – A plaintext password to verify

Return type

bool

New in version 3.2.0.

verify_auth_token(*data*)

Perform additional verification of contents of auth token. Prior to this being called the token has been validated (via signing) and has not expired.

Parameters

data (*str | bytes*) – the data as formulated by *get_auth_token()*

Return type
bool

New in version 3.3.0.

Changed in version 4.0.0: If user model has `fs_token_uniquifier` - use that otherwise use `fs_uniquifier`.

`class flask_security.`**`RoleMixin`**

Mixin for *Role* model definitions

`get_permissions``()`

Return set of permissions associated with role.

New in version 3.3.0.

Return type
set

`class flask_security.`**`WebAuthnMixin`**

`get_user_mapping``()`

Return the filter needed by find_user() to get the user associated with this webauthn credential. Note that this probably has to be overridden using mongoengine.

New in version 5.0.0.

Return type
Dict[str, *Any*]

`class flask_security.`**`AnonymousUser`**

AnonymousUser definition

`has_role``(`**args*`)`

Returns *False*

3.1.4 Datastores

`class flask_security.`**`UserDatastore`**`(`*user_model*, *role_model*, *webauthn_model=None*`)`

Abstracted user datastore.

Parameters

- **`user_model`** (*Type*`[User]`) – A user model class definition
- **`role_model`** (*Type*`[Role]`) – A role model class definition
- **`webauthn_model`** (*Type*`[WebAuthn] | None`) – A model used to store webauthn registrations

Important: For mutating operations, the user/role will be added to the datastore (by calling self.put(<object>). If the datastore is session based (such as for SQLAlchemyDatastore) it is up to caller to actually commit the transaction by calling datastore.commit().

Note: You must implement get_user_mapping in your WebAuthn model if your User model doesn't have a primary key Column called 'id' `activate_user`(*user*)

Activates a specified user. Returns *True* if a change was made.

Parameters

`user` (`User`) – The user to activate

Return type

bool

`add_permissions_to_role`(*role*, *permissions*)

Add one or more permissions to role.

Parameters

- `role` (`Role | str`) – The role to modify. Can be a Role object or string role name
- `permissions` (`set | list | tuple | str`) – a set, list, tuple or comma separated string.

Returns

True if permissions added, False if role doesn't exist.

Return type

bool

Caller must commit to DB.

New in version 4.0.0.

`add_role_to_user`(*user*, *role*)

Adds a role to a user.

Parameters

- `user` (`User`) – The user to manipulate.
- `role` (`Role | str`) – The role to add to the user. Can be a Role object or string role name

Returns

True is role was added, False if role already existed.

Return type

bool

`create_role`(***kwargs*)

Creates and returns a new role from the given parameters. Supported params (depending on RoleModel):

Parameters

- `name` – Role name
- `permissions` – a list, set, tuple or comma separated string. These are user-defined strings that correspond to args used with @permissions_required()

  New in version 3.3.0.
- `kwargs` (*Any*) –

Return type

Role

`create_user`(***kwargs*)

Creates and returns a new user from the given parameters.

Parameters

- **email** – required.
- **password** – Hashed password.
- **roles** – list of roles to be added to user. Can be Role objects or strings
- **kwargs** (*Any*) –

Return type

User

Any other element of the User data model may be supplied as well.

Note: No normalization is done on email - it is assumed the caller has already done that.

Best practice is:

```
try:
    enorm = app.security._mail_util.validate(email)
except ValueError:
```

Note: The roles kwparam is modified as part of the call - it will, if necessary, be converted from names to role instances.

Danger: Be aware that whatever *password* is passed in will be stored directly in the DB. Do NOT pass in a plaintext password! Best practice is to pass in `hash_password(plaintext_password)`.

Furthermore, no validation nor normalization is done on the password (e.g for minimum length).

Best practice is:

```
pbad, pnorm = app.security._password_util.validate(password, True)
```

Look for *pbad* being None. Pass the normalized password *pnorm* to this method.

The new user's `active` property will be set to `True` unless explicitly set to `False` in *kwargs* (e.g. active = False)

`create_webauthn`(*user*, *credential_id*, *public_key*, *name*, *sign_count*, *usage*, *device_type*, *backup_state*, *transports=None*, *extensions=None*, ***kwargs*)

Create a new webauthn registration record. Note that we need to find webauthn records per user as well as find a user from a given webauthn (credential_id) record.

Parameters

- **user** (`User`) –
- **credential_id** (*bytes*) –
- **public_key** (*bytes*) –
- **name** (*str*) –

- **sign_count** (*int*) –
- **usage** (*str*) –
- **device_type** (*str*) –
- **backup_state** (*bool*) –
- **transports** (*List[str] | None*) –
- **extensions** (*str | None*) –
- **kwargs** (*Any*) –

Return type
None

`deactivate_user`(*user*)

Deactivates a specified user. Returns *True* if a change was made.

This will immediately disallow access to all endpoints that require authentication either via session or tokens. The user will not be able to log in again.

Parameters
user (`User`) – The user to deactivate

Return type
bool

`delete_user`(*user*)

Deletes the specified user.

Parameters
user (`User`) – The user to delete

Return type
None

`delete_webauthn`(*webauthn*)

Parameters
webauthn (`WebAuthn`) –

Return type
None

`find_or_create_role`(*name*, ***kwargs*)

Returns a role matching the given name or creates it with any additionally provided parameters.

Parameters

- **name** (*str*) –
- **kwargs** (*Any*) –

Return type
Role

`find_role`(*role*)

Returns a role matching the provided name.

Parameters
role (*str*) –

Return type
Role | None

find_user(***kwargs*)

Returns a user matching the provided parameters. Besides keyword arguments used to filter the results, 'case_insensitive' can be passed (defaults to False)

Parameters
kwargs (*Any*) –

Return type
User | None

find_user_from_webauthn(*webauthn*)

Returns user associated with this webauthn credential

Parameters
webauthn (WebAuthn) –

Return type
User | None

find_webauthn(*credential_id*)

Returns a credential matching the id.

Parameters
credential_id (*bytes*) –

Return type
WebAuthn | None

mf_delete_recovery_code(*user*, *idx*)

Delete a single recovery code. Recovery codes are single-use - so delete after using!

Return True if code found and deleted, False otherwise.

Parameters

- **user** (User) –
- **idx** (*int*) –

Return type
bool

mf_set_recovery_codes(*user*, *rcs*)

Set MF recovery codes into user record. Any existing codes will be erased.

Parameters

- **user** (User) –
- **rcs** (*List[str] | None*) –

Return type
None

remove_permissions_from_role(*role*, *permissions*)

Remove one or more permissions from a role.

Parameters

- **role** (Role | *str*) – The role to modify. Can be a Role object or string role name

- **permissions** (*set | list | tuple | str*) – a set, list, tuple or a comma separated string.

Returns
True if permissions removed, False if role doesn't exist.

Return type
bool

Caller must commit to DB.

New in version 4.0.0.

remove_role_from_user(*user*, *role*)

Removes a role from a user.

Parameters

- **user** (User) – The user to manipulate. Can be an User object or email
- **role** (Role | *str*) – The role to remove from the user. Can be a Role object or string role name

Returns
True if role was removed, False if role doesn't exist or user didn't have role.

Return type
bool

reset_user_access(*user*)

Use this method to reset user authentication methods in the case of compromise. This will:

- reset fs_uniquifier - which causes session cookie, remember cookie, auth tokens to be unusable
- reset fs_token_uniquifier (if present) - cause auth tokens to be unusable
- remove all unified signin TOTP secrets so those can't be used
- remove all two-factor secrets so those can't be used
- remove all registered webauthn credentials
- remove all one-time recovery codes
- will NOT affect password

Note that if using unified sign in and allow 'email' as a way to receive a code; this will also get reset. If the user registered w/o a password then they likely will have no way to authenticate.

Note - this method isn't used directly by Flask-Security - it is provided as a helper for an application's administrative needs.

Remember to call commit on DB if needed.

New in version 3.4.1.

Changed in version 5.0.0: Added webauthn and recovery codes reset.

Parameters
user (User) –

Return type
None

`set_token_uniquifier`(*user*, *uniquifier=None*)

Set user's auth token identity key. This will immediately render outstanding auth tokens invalid.

Parameters

- **user** (`User`) – User to modify
- **uniquifier** (*str | None*) – Unique value - if none then uuid.uuid4().hex is used

Return type

None

This method is a no-op if the user model doesn't contain the attribute `fs_token_uniquifier`

New in version 4.0.0.

`set_uniquifier`(*user*, *uniquifier=None*)

Set user's Flask-Security identity key. This will immediately render outstanding auth tokens, session cookies and remember cookies invalid.

Parameters

- **user** (`User`) – User to modify
- **uniquifier** (*str | None*) – Unique value - if none then uuid.uuid4().hex is used

Return type

None

New in version 3.3.0.

`set_webauthn_user_handle`(*user*, *user_handle=None*)

Set the value for the Relaying Party's (that's us) UserHandle (user.id) If no value is passed in, a UUID is generated.

Parameters

- **user** (`User`) –
- **user_handle** (*str | None*) –

Return type

None

`tf_reset`(*user*)

Disable two-factor auth for user.

Parameters

user (`User`) –

Return type

None

`tf_set`(*user*, *primary_method*, *totp_secret=None*, *phone=None*)

Set two-factor info into user record. This carefully only changes things if different.

If totp_secret isn't provided - existing one won't be changed. If phone isn't provided, the existing phone number won't be changed.

This could be called from an application to apiori setup a user for two factor without the user having to go through the setup process.

To get a totp_secret - use `app.security._totp_factory.generate_totp_secret()`

Parameters

- **user** (User) –
- **primary_method** (*str*) –
- **totp_secret** (*str | None*) –
- **phone** (*str | None*) –

Return type
None

toggle_active(*user*)

Toggles a user's active status. Always returns True.

Parameters
user (User) –

Return type
bool

us_get_totp_secrets(*user*)

Return totp secrets. These are json encoded in the DB.

Returns a dict with methods as keys and secrets as values.

New in version 3.4.0.

Parameters
user (User) –

Return type
Dict[str, str]

us_put_totp_secrets(*user*, *secrets*)

Save secrets. Assume to be a dict (or None) with keys as methods, and values as (encrypted) secrets.

New in version 3.4.0.

Parameters

- **user** (User) –
- **secrets** (*Dict[str, str] | None*) –

Return type
None

us_reset(*user*, *method=None*)

Disable unified sign in for user. This will disable authenticator app and SMS, and email. N.B. if user has no password they may not be able to authenticate at all.

New in version 3.4.1.

Changed in version 5.0.0: Added optional method argument to delete just a single method

Parameters

- **user** (User) –
- **method** (*str | None*) –

Return type
None

us_set(*user*, *method*, *totp_secret=None*, *phone=None*)

Set unified sign in info into user record.

If totp_secret isn't provided - existing one won't be changed. If phone isn't provided, the existing phone number won't be changed.

This could be called from an application to apiori setup a user for unified sign in without the user having to go through the setup process.

To get a totp_secret - use `app.security._totp_factory.generate_totp_secret()`

New in version 3.4.1.

Parameters

- **user** (`User`) –
- **method** (*str*) –
- **totp_secret** (*str | None*) –
- **phone** (*str | None*) –

Return type

None

webauthn_reset(*user*)

Reset access via webauthn credentials. This will DELETE all registered credentials. There doesn't appear to be any reason to change the user's fs_webauthn_user_handle.

Parameters

user (`User`) –

Return type

None

class flask_security.**SQLAlchemyUserDatastore**(*db*, *user_model*, *role_model*, *webauthn_model=None*)

Bases: *SQLAlchemyDatastore*, *UserDatastore*

A UserDatastore implementation that assumes the use of Flask-SQLAlchemy for datastore transactions.

Parameters

- **db** (*flask_sqlalchemy.SQLAlchemy*) –
- **user_model** (*Type*[`User`]) – See *Models*.
- **role_model** (*Type*[`Role`]) – See *Models*.
- **webauthn_model** (*Type*[`WebAuthn`] *| None*) – See *Models*.

class flask_security.**SQLAlchemySessionUserDatastore**(*session*, *user_model*, *role_model*, *webauthn_model=None*)

Bases: *SQLAlchemyUserDatastore*, *SQLAlchemyDatastore*

A UserDatastore implementation that directly uses SQLAlchemy's session API.

Parameters

- **session** (*sqlalchemy.orm.scoping.scoped_session*) –
- **user_model** (*Type*[`User`]) – See *Models*.
- **role_model** (*Type*[`Role`]) – See *Models*.
- **webauthn_model** (*Type*[`WebAuthn`] *| None*) – See *Models*.

`class flask_security.`**`MongoEngineUserDatastore`**`(`*db*, *user_model*, *role_model*, *webauthn_model=None*`)`

Bases: *`MongoEngineDatastore`*, *`UserDatastore`*

A UserDatastore implementation that assumes the use of MongoEngine for datastore transactions.

Parameters

- **`db`** (*`mongoengine.connection`*) –
- **`user_model`** (*`Type[User]`*) – See *Models*.
- **`role_model`** (*`Type[Role]`*) – See *Models*.
- **`webauthn_model`** (*`Type[WebAuthn] | None`*) – See *Models*.

`class flask_security.`**`PeeweeUserDatastore`**`(`*db*, *user_model*, *role_model*, *role_link*, *webauthn_model=None*`)`

Bases: *`PeeweeDatastore`*, *`UserDatastore`*

A UserDatastore implementation that assumes the use of Peewee Flask utils for datastore transactions.

`class flask_security.`**`PonyUserDatastore`**`(`*db*, *user_model*, *role_model*, *webauthn_model=None*`)`

Bases: *`PonyDatastore`*, *`UserDatastore`*

A UserDatastore implementation that assumes the use of PonyORM for datastore transactions.

Code primarily from https://github.com/ET-CS but taken over after being abandoned.

Parameters

- **`db`** –
- **`user_model`** – See *Models*.
- **`role_model`** – See *Models*.
- **`webauthn_model`** – See *Models*.

`class flask_security.datastore.`**`SQLAlchemyDatastore`**`(`*db*`)`

Internal class implementing DataStore interface.

`class flask_security.datastore.`**`MongoEngineDatastore`**`(`*db*`)`

Internal class implementing DataStore interface.

`class flask_security.datastore.`**`PeeweeDatastore`**`(`*db*`)`

Internal class implementing DataStore interface.

`class flask_security.datastore.`**`PonyDatastore`**`(`*db*`)`

Internal class implementing DataStore interface.

`class` **`User`**

The User model. This must be provided by the application. See *Models*.

`class` **`Role`**

The Role model. This must be provided by the application. See *Models*.

`class` **`WebAuthn`**

The WebAuthn model. This must be provided by the application. See *Models*.

3.1.5 Utils

`flask_security.`**`lookup_identity`**`(`*identity*`)`

Lookup identity in DB. This loops through, in order, SECURITY_USER_IDENTITY_ATTRIBUTES, and first calls the mapper function to validate/normalize. Then the db.find_user is called on the specified user model attribute.

`flask_security.`**`login_user`**`(`*user*, *remember=None*, *authn_via=None*`)`

Perform the login routine.

If *SECURITY_TRACKABLE* is used, make sure you commit changes after this request (i.e. `app.security.datastore.commit()`).

Parameters

- **`user`** (`User`) – The user to login
- **`remember`** (*`bool | None`*) – Flag specifying if the remember cookie should be set. If `None` use value of SECURITY_DEFAULT_REMEMBER_ME
- **`authn_via`** (*`List[str] | None`*) – A list of strings denoting which mechanism(s) the user authenticated with. These should be one or more of ["password", "sms", "authenticator", "email"] or other 'auto-login' mechanisms.

Returns

True if user successfully logged in.

Return type

bool

`flask_security.`**`logout_user`**`()`

Logs out the current user.

This will also clean up the remember me cookie if it exists.

This sends an `identity_changed` signal to note that the current identity is now the *AnonymousIdentity*

Return type

None

`flask_security.`**`check_and_update_authn_fresh`**`(`*within*, *grace*, *method=None*`)`

Check if user authenticated within specified time and update grace period.

Parameters

- **`within`** (*`timedelta`*) – A timedelta specifying the maximum time in the past that the caller authenticated that is still considered 'fresh'.
- **`grace`** (*`timedelta`*) – A timedelta that, if the current session is considered 'fresh' will set a grace period for which freshness won't be checked. The intent here is that the caller shouldn't get part-way though a set of operations and suddenly be required to authenticate again.
- **`method`** (*`str | None`*) – Optional - if set and == "basic" then will always return True. (since basic-auth sends username/password on every request)

Return type

bool

If within.total_seconds() is negative, will always return True (always 'fresh'). This effectively just disables this entire mechanism.

If “fs_gexp” is in the session and the current timestamp is less than that, return True and extend grace time (i.e. set fs_gexp to current time + grace).

If not within the grace period, and within.total_seconds() is 0, return False (not fresh).

Be aware that for this to work, sessions and therefore session cookies must be functioning and being sent as part of the request. If the required state isn’t in the session cookie then return False (not ‘fresh’).

Warning: Be sure the caller is already authenticated PRIOR to calling this method.

New in version 3.4.0.

Changed in version 4.0.0: Added *method* parameter.

`flask_security.`**`get_hmac`**`(`*password*`)`

Returns a Base64 encoded HMAC+SHA512 of the password signed with the salt specified by *SECURITY_PASSWORD_SALT*.

Parameters
`password` (*`str | bytes`*) – The password to sign

Return type
bytes

`flask_security.`**`get_request_attr`**`(`*name*`)`

Retrieve a request local attribute.

Current public attributes are:

fs_authn_via
will be set to the authentication mechanism (session, token, basic) that the current request was authenticated with.

Returns None if attribute doesn’t exist.

New in version 4.0.0.

Changed in version 4.1.5: Use ‘g’ rather than request_ctx stack which is going away post Flask 2.2

Parameters
`name` (*`str`*) –

Return type
Any

`flask_security.`**`verify_password`**`(`*password*, *password_hash*`)`

Returns `True` if the password matches the supplied hash.

Parameters

- **`password`** (*`str | bytes`*) – A plaintext password to verify
- **`password_hash`** (*`str | bytes`*) – The expected hash value of the password (usually from your database)

Return type
bool

Note: Make sure that the password passed in has already been normalized.

flask_security.**verify_and_update_password**(*password*, *user*)

Returns `True` if the password is valid for the specified user.

Additionally, the hashed password in the database is updated if the hashing algorithm happens to have changed.

N.B. you MUST call DB commit if you are using a session-based datastore (such as SqlAlchemy) since the user instance might have been altered (i.e. `app.security.datastore.commit()`). This is usually handled in the view.

Parameters

- **password** (*str* | *bytes*) – A plaintext password to verify
- **user** (`User`) – The user to verify against

Return type
bool

Tip: This should not be called directly - rather use *UserMixin.verify_and_update_password()*

flask_security.**hash_password**(*password*)

Hash the specified plaintext password.

Unless the hash algorithm (as specified by *SECURITY_PASSWORD_HASH*) is listed in the configuration variable *SECURITY_PASSWORD_SINGLE_HASH*, perform a double hash - first create an HMAC from the plaintext password and the value of *SECURITY_PASSWORD_SALT*, then use the configured hashing algorithm. This satisfies OWASP/ASVS section 2.4.5: 'provide additional iteration of a key derivation'.

New in version 2.0.2.

Parameters
password (*str* | *bytes*) – The plaintext password to hash

Return type
Any

flask_security.**admin_change_password**(*user*, *new_passwd*, *notify=True*)

Administratively change a user's password. Note that this will immediately render the user's existing sessions (and possibly authentication tokens) invalid.

It is up to the caller to inform the user of their new password by some out-of-band means.

Parameters

- **user** (`User`) – The user object to change
- **new_passwd** (*str*) – The new plain-text password to assign to the user.
- **notify** (*bool*) – If True and SECURITY_SEND_PASSWORD_CHANGE_EMAIL is True send the 'change_notice' email to the user.

Return type
None

flask_security.**uia_phone_mapper**(*identity*)

Used to match identity as a phone number. This is a simple proxy to *PhoneUtil*

See *SECURITY_USER_IDENTITY_ATTRIBUTES*.

New in version 3.4.0.

Parameters
identity (*str*) –

Return type
str | None

`flask_security.`**`uia_email_mapper`**(*identity*)

Used to match identity as an email.

Returns
Normalized email or None if not valid email.

Parameters
`identity` (*`str`*) –

Return type
str | None

See *`SECURITY_USER_IDENTITY_ATTRIBUTES`*.

New in version 3.4.0.

`flask_security.`**`uia_username_mapper`**(*identity*)

Used to match identity as a username. This is a simple proxy to *`UsernameUtil`*

See *`SECURITY_USER_IDENTITY_ATTRIBUTES`*.

New in version 4.1.0.

Parameters
`identity` (*`str`*) –

Return type
str | None

`flask_security.`**`url_for_security`**(*endpoint*, ***values*)

Return a URL for the security blueprint

Parameters

- **`endpoint`** (*`str`*) – the endpoint of the URL (name of the function)
- **`values`** (*`Any`*) – the variable arguments of the URL rule
- **`_external`** – if set to *True*, an absolute URL is generated. Server address can be changed via *SERVER_NAME* configuration variable which defaults to *localhost*.
- **`_anchor`** – if provided this is added as anchor to the URL.
- **`_method`** – if provided this explicitly specifies an HTTP method.

Return type
str

`flask_security.`**`send_mail`**(*subject*, *recipient*, *template*, ***context*)

Send an email.

Parameters

- **`subject`** – Email subject
- **`recipient`** – Email recipient
- **`template`** – The name of the email template
- **`context`** – The context to render the template with

This formats the email and passes it off to *`MailUtil`* to actually send the message.

flask_security.**check_and_get_token_status**(*token*, *serializer_name*, *within*)

Get the status of a token and return data.

Parameters

- **token** (*str*) – The token to check
- **serializer_name** (*str*) – The name of the serializer. Can be one of the following: `confirm`, `login`, `reset`, `us_setup` `remember`, `two_factor_validity`, `wan`
- **within** (*timedelta*) – max age - passed as a timedelta

Returns
a tuple of (expired, invalid, data)

Return type
Tuple[bool, bool, *Any*]

New in version 3.4.0.

flask_security.**get_url**(*endpoint_or_url*, *qparams=None*)

Returns a URL if a valid endpoint is found. Otherwise, returns the provided value.

Parameters

- **endpoint_or_url** (*str*) – The endpoint name or URL to default to
- **qparams** (*Dict[str, str] | None*) – additional query params to add to end of url

Returns
URL

Return type
str

flask_security.**password_length_validator**(*password*)

Test password for length.

Parameters
password (*str*) – Plain text password to check

Returns
`None` if password conforms to length requirements, a list of error/suggestions if not.

Return type
List[str] | None

New in version 3.4.0.

flask_security.**password_complexity_validator**(*password*, *is_register*, ***kwargs*)

Test password for complexity.

Currently just supports 'zxcvbn'.

Parameters

- **password** (*str*) – Plain text password to check
- **is_register** (*bool*) – if True then kwargs are arbitrary additional info. (e.g. info from a registration form). If False, must be a SINGLE key "user" that corresponds to the current_user. All string values will be extracted and sent to the complexity checker.
- **kwargs** (*Any*) –

Returns
`None` if password is complex enough, a list of error/suggestions if not. Be aware that zxcvbn does not (easily) provide a way to localize messages.

Return type
List[str] | None

New in version 3.4.0.

`flask_security.`**`password_breached_validator`**(*password*)

Check if password on breached list. Does nothing unless *`SECURITY_PASSWORD_CHECK_BREACHED`* is set. If password is found on the breached list, return an error if the count is greater than or equal to *`SECURITY_PASSWORD_BREACHED_COUNT`*. Uses `pwned()`.

Parameters
`password` (*`str`*) – Plain text password to check

Returns
`None` if password passes breached tests, else a list of error messages.

Return type
List[str] | None

New in version 3.4.0.

`flask_security.`**`pwned`**(*password*)

Check password against pwnedpasswords API using k-Anonymity. https://haveibeenpwned.com/API/v3

Returns
Count of password in DB (0 means hasn't been compromised)

Parameters
`password` (*`str`*) –

Return type
int

Can raise HTTPError

New in version 3.4.0.

`flask_security.`**`transform_url`**(*url*, *qparams=None*, ***kwargs*)

Modify url

Parameters

- **`url`** (*`str`*) – url to transform (can be relative)
- **`qparams`** (*`Dict[str, str] | None`*) – additional query params to add to end of url
- **`kwargs`** (*`str`*) – pieces of URL to modify - e.g. netloc=localhost:8000

Returns
Modified URL

Return type
str

New in version 3.2.0.

`flask_security.`**`unique_identity_attribute`**(*form*, *field*)

A validator that checks the field data against all configured SECURITY_USER_IDENTITY_ATTRIBUTES. This can be used as part of registration.

Be aware that the "mapper" function likely also normalizes the input in addition to validating it.

Parameters

- `form` –
- `field` –

Returns

Nothing; if field data corresponds to an existing User, ValidationError is raised.

`flask_security.`**`us_send_security_token`**(*user*, *method*, *totp_secret*, *phone_number*, *send_magic_link=False*)

Generate and send the security code.

Parameters

- **`user`** – The user to send the code to
- **`method`** – The method in which the code will be sent
- **`totp_secret`** – the unique shared secret of the user
- **`phone_number`** – If 'sms' phone number to send to
- **`send_magic_link`** – If true a magic link that can be clicked on will be sent. This shouldn't be sent during a setup.

There is no return value - it is assumed that exceptions are thrown by underlying methods that callers can catch.

Flask-Security code should NOT call this directly - call *`UserMixin.us_send_security_token()`*

New in version 3.4.0.

`flask_security.`**`tf_send_security_token`**(*user*, *method*, *totp_secret*, *phone_number*)

Sends the security token via email/sms for the specified user.

Parameters

- **`user`** – The user to send the code to
- **`method`** – The method in which the code will be sent ('email' or 'sms', or 'authenticator') at the moment
- **`totp_secret`** – a unique shared secret of the user
- **`phone_number`** – If 'sms' phone number to send to

There is no return value - it is assumed that exceptions are thrown by underlying methods that callers can catch.

Flask-Security code should NOT call this directly - call *`UserMixin.tf_send_security_token()`*

`class flask_security.`**`AsaList`**

SQL-like DBs don't have a List type - so do that here by converting to a comma separate string. For SQLAlchemy-based datastores, this can be used as:

```
Column(MutableList.as_mutable(AsaList()), nullable=True)
```

`class flask_security.`**`SmsSenderBaseClass`**

`abstract` **`send_sms`**(*from_number*, *to_number*, *msg*)

Abstract method for sending sms messages

New in version 3.2.0.

Parameters

- **`from_number`** (*`str`*) –

- **to_number** (*str*) –
- **msg** (*str*) –

Return type
None

`class flask_security.`**`SmsSenderFactory`**

`classmethod` **`createSender`**(*name*, **args*, ***kwargs*)

Initialize an SMS sender.

Parameters
name – Name as registered in SmsSenderFactory:senders (e.g. 'Twilio')

New in version 3.2.0.

`class flask_security.`**`OAuthGlue`**(*app*, *oauthapp=None*)

Provide the necessary glue between the Flask-Security login process and authlib oauth client code.

There are some builtin providers which can be used or not - configured via *SECURITY_OAUTH_BUILTIN_PROVIDERS*. Any other provider can be registered using app.security.oauthglue.register_provider().

See Flask OAuth Client

New in version 5.1.0.

Parameters

- **app** (*flask.Flask*) –
- **oauthapp** (*OAuth | None*) –

`register_provider`(*name*, *registration_info*, *fetch_identity_cb*)

Add a provider to the list.

Parameters

- **name** (*str*) – Name of provider. This is used as part of the *SECURITY_OAUTH_START_URL*.
- **registration_info** (*Dict[str, Any] | None*) – Sent directly to authlib. Set this to None if you already have registered the provider directly with OAuth.
- **fetch_identity_cb** (*Callable[[OAuth, str], Tuple[str, Any]]*) – This callback is called when the oauth redirect happens. It must take the response from the provider and return a tuple of <user_model_field_name, value> - which will be used to look up the user in the datastore.

Return type
None

The provider can be registered with OAuth here or already be done by the application. If you register directly with OAuth make sure to use the same *name*.

3.1.6 Extendable Classes

Each of the following classes can be extended and passed in as part of Security() instantiation.

class flask_security.**PhoneUtil**(*app*)

Provide parsing and validation for user inputted phone numbers. Subclass this to use a different underlying phone number parsing library.

To provide your own implementation, pass in the class as `phone_util_cls` at init time. Your class will be instantiated once as part of Flask-Security initialization.

New in version 3.4.0.

Changed in version 4.0.0: __init__ takes app argument, and is instantiated at Flask-Security initialization time rather than at first request.

Parameters
app (*flask.Flask*) –

__init__(*app*)

Instantiate class.

Parameters
app (*flask.Flask*) – The Flask application being initialized.

get_canonical_form(*input_data*)

Validate and return a canonical form to be stored in DB and compared against. Returns `None` if input isn't a valid phone number.

Parameters
input_data (*str*) –

Return type
str | None

validate_phone_number(*input_data*)

Return `None` if a valid phone number else the `PHONE_INVALID` error message.

Parameters
input_data (*str*) –

Return type
str | None

class flask_security.**MailUtil**(*app*)

Utility class providing methods for validating, normalizing and sending emails.

This default class uses the email_validator package to handle validation and normalization, and the flask_mailman package (if initialized) to send emails.

To provide your own implementation, pass in the class as `mail_util_cls` at init time. Your class will be instantiated once as part of app initialization.

New in version 4.0.0.

Parameters
app (*flask.Flask*) –

__init__(*app*)

Instantiate class.

Parameters
app (*flask.Flask*) – The Flask application being initialized.

normalize(*email*)

Given an input email - return a normalized version or raises ValueError if field value isn't syntactically valid.

This is called for forms that use email as an identity to be looked up.

Must be called in app context and uses *SECURITY_EMAIL_VALIDATOR_ARGS* config variable to pass any relevant arguments to email_validator.validate_email() method.

This defaults to NOT checking for deliverability (i.e. DNS checks).

Will throw email_validator.EmailNotValidError (ValueError) if email isn't syntactically valid.

Parameters
email (*str*) –

Return type
str

send_mail(*template*, *subject*, *recipient*, *sender*, *body*, *html*, ***kwargs*)

Send an email via the Flask-Mailman or Flask-Mail or other mail extension.

Parameters

- **template** (*str*) – the Template name. The message has already been rendered however this might be useful to differentiate why the email is being sent.
- **subject** (*str*) – Email subject
- **recipient** (*str*) – Email recipient
- **sender** (*str | tuple*) – who to send email as (see *SECURITY_EMAIL_SENDER*)
- **body** (*str*) – the rendered body (text)
- **html** (*str | None*) – the rendered body (html)
- **kwargs** (*Any*) – the entire context

Return type
None

It is possible that sender is a lazy_string for localization (unlikely but..) so we cast to str() here to force localization.

validate(*email*)

Validate the given email. If valid, the normalized version is returned. This is used by forms/views that require an email that likely can have an actual email sent to it.

Must be called in app context and uses *SECURITY_EMAIL_VALIDATOR_ARGS* config variable to pass any relevant arguments to email_validator.validate_email() method.

ValueError is thrown if not valid.

Parameters
email (*str*) –

Return type
str

class flask_security.**PasswordUtil**(*app*)

Utility class providing methods for validating and normalizing passwords.

To provide your own implementation, pass in the class as `password_util_cls` at init time. Your class will be instantiated once as part of app initialization.

New in version 4.0.0.

Parameters
app (*flask.Flask*) –

`__init__`(*app*)

Instantiate class.

Parameters
app (*flask.Flask*) – The Flask application being initialized.

`normalize`(*password*)

Given an input password - return a normalized version (using Python's unicodedata.normalize()). Must be called in app context and uses *SECURITY_PASSWORD_NORMALIZE_FORM* config variable.

Parameters
password (*str*) –

Return type
str

`validate`(*password*, *is_register*, ***kwargs*)

Password validation. Called in app/request context.

If is_register is True then kwargs will be the contents of the register form. If is_register is False, then there is a single kwarg "user" which has the current user data model.

The password is first normalized then validated. Return value is a tuple ([msgs], normalized_password)

Parameters

- **password** (*str*) –
- **is_register** (*bool*) –
- **kwargs** (*Any*) –

Return type
Tuple[*List* | None, str]

`class flask_security.`**`MfRecoveryCodesUtil`**(*app*)

Handle creation, checking, encrypting and decrypting recovery codes. Since these are rarely used - keep them encrypted until needed - yes if someone gets access to memory they can find the key…

Parameters
app (*flask.Flask*) –

`__init__`(*app*)

Parameters
app (*flask.Flask*) –

`class flask_security.`**`UsernameUtil`**(*app*)

Utility class providing methods for validating and normalizing usernames.

To provide your own implementation, pass in the class as `username_util_cls` at init time. Your class will be instantiated once as part of app initialization.

New in version 4.1.0.

Parameters
app (*flask.Flask*) –

`__init__`(*app*)

Instantiate class.

Parameters
`app` (*`flask.Flask`*) – The Flask application being initialized.

`check_username`(*username*)

Given a username - check for allowable character categories. This is broken out so applications can easily override this method only.

By default allow letters and numbers (using unicodedata.category).

Returns None if allowed, error message if not allowed.

Parameters
`username` (*`str`*) –

Return type
str | None

`normalize`(*username*)

Given an input username - return a clean (using bleach) and normalized (using Python's unicodedata.normalize()) version. Must be called in app context and uses *`SECURITY_USERNAME_NORMALIZE_FORM`* config variable.

Parameters
`username` (*`str`*) –

Return type
str

`validate`(*username*)

Username validation. Called in app/request context.

The username is first validated then normalized. Input is restricted/validated via a call to check_username. Return value is a tuple (msg, normalized_username). msg will be None if properly validated.

It is important that None be returned if data is an empty string since otherwise DBs will complain since the field is unique/nullable.

Parameters
`username` (*`str`*) –

Return type
Tuple[str | None, str | None]

class `flask_security.WebauthnUtil`(*app*)

Utility class allowing an application to fine-tune various Relying Party attributes.

To provide your own implementation, pass in the class as `webauthn_util_cls` at init time. Your class will be instantiated once as part of app initialization.

New in version 5.0.0.

Parameters
`app` (*`flask.Flask`*) –

`__init__`(*app*)

Instantiate class.

Parameters
`app` (*`flask.Flask`*) – The Flask application being initialized.

`authentication_options`(*user*, *usage*, *existing_options*)

Parameters

- **user** (`User | None`) – User object - could be used to configure on a per-user basis. However this can be null.
- **usage** (`List[str]`) – Either "first" or "secondary" (webauthn is being used as a second factor for authentication)
- **existing_options** (`Dict[str, Any]`) – Currently filled in authentication options.

Return type
Dict[str, *Any*]

Return a dict that will be sent in to
py-webauthn generate_authentication_options

`authenticator_selection`(*user*, *usage*)

Parameters

- **user** (`User`) – User object - could be used to configure on a per-user basis.
- **usage** (`str`) – Either "first" or "secondary" (webauthn is being used as a second factor for authentication

Return type
AuthenticatorSelectionCriteria

Part of the registration ceremony is providing information about what kind of authenticators the app is interested in. See: https://www.w3.org/TR/2021/REC-webauthn-2-20210408/#dictionary-authenticatorSelection

The main options are:

- whether you want a ResidentKey (discoverable)
- Attachment - platform or cross-platform
- Does the key have to provide user-verification

Note:
If the key isn't resident then it isn't discoverable which means that the user won't be able to use that key unless they identify themselves (use the key as a second factor OR type in their identity). If they are forced to type in their identity PRIOR to being authenticated, then there is the possibility that the app will leak username information.

Parameters

- **user** (`User`) –
- **usage** (`str`) –

Return type
AuthenticatorSelectionCriteria

`registration_options`(*user*, *usage*, *existing_options*)

Parameters

- **user** (`User`) – User object - could be used to configure on a per-user basis.

- **usage** (*str*) – Either "first" or "secondary" (webauthn is being used as a second factor for authentication)
- **existing_options** (*Dict[str, Any]*) – Currently filled in registration options.

Return type
Dict[str, *Any*]

Return a dict that will be sent in to py-webauthn generate_registration_options

user_verification(*user*, *usage*)

As part of signin - do we want/need user verification. This is called from /wan-signin and /wan-verify

Parameters

- **user** (User | *None*) – User object - could be used to configure on a per-user basis. Note that this may not be set on initial wan-signin.
- **usage** (*List[str]*) – List of "first", "secondary" (webauthn is being used as a second factor for authentication). Note that in the `verify/reauthentication` case this list is derived from *SECURITY_WAN_ALLOW_AS_VERIFY*

Return type
UserVerificationRequirement

class flask_security.**Totp**(*secrets*, *issuer*)

Encapsulate usage of Passlib TOTP functionality.

Flask-Security doesn't implement any replay-attack protection out of the box as suggested by: https://passlib.readthedocs.io/en/stable/narr/totp-tutorial.html#match-verify

Subclass this and implement the get/set last_counter methods. Your subclass can be registered at Flask-Security creation/initialization time.

New in version 3.4.0.

Parameters

- **secrets** (*Dict[str | int, str]*) –
- **issuer** (*str*) –

generate_qrcode(*username*, *totp*)

Generate QRcode
Using username, totp, generate the actual QRcode image. This method can be overridden to fine-tune how the image is created - such as size, color etc.

It must return a string suitable for use in an <img src=xx> tag.

New in version 4.0.0.

Parameters

- **username** (*str*) –
- **totp** (*str*) –

Return type
str

get_last_counter(*user*)

Implement this to fetch stored last_counter from cache.

Parameters
user (User) – User model

Returns
last_counter as stored in set_last_counter()

Return type
TotpMatch | None

`set_last_counter`(*user*, *tmatch*)
Implement this to cache last_counter.

Parameters

- **user** (`User`) – User model
- **tmatch** (*TotpMatch*) – a TotpMatch as returned from totp.verify()

Return type
None

3.1.7 Forms

`class flask_security.`**`ChangePasswordForm`**(**args*, ***kwargs*)
The default change password form

`class flask_security.`**`ConfirmRegisterForm`**(**args*, ***kwargs*)
This form is used for registering when 'confirmable' is set. The only difference between this and the other RegisterForm is that this one doesn't require re-typing in the password…

We want to support OWASP best-practice around mitigating user enumeration. To that end we run through the entire validation regardless - this allows us to still return important bad-password messages. In the case of an existing email or username - we set form.existing_xx so that the view can decide how to match responses (e.g. json responses always return 200).

`class flask_security.`**`ForgotPasswordForm`**(**args*, ***kwargs*)
The default forgot password form

`class flask_security.`**`LoginForm`**(**args*, ***kwargs*)
The default login form

`class flask_security.`**`MfRecoveryCodesForm`**(**args*, ***kwargs*)
Generate and fetch recovery codes

`class flask_security.`**`MfRecoveryForm`**(**args*, ***kwargs*)
Accept recovery code for second factor authentication

`class flask_security.`**`PasswordlessLoginForm`**(**args*, ***kwargs*)
The passwordless login form

`class flask_security.`**`RegisterForm`**(**args*, ***kwargs*)

`class flask_security.`**`ResetPasswordForm`**(**args*, ***kwargs*)
The default reset password form

`class flask_security.`**`SendConfirmationForm`**(**args*, ***kwargs*)
The default send confirmation form

`class flask_security.`**`TwoFactorVerifyCodeForm`**(**args*, ***kwargs*)
The Two-factor token validation form

class flask_security.**TwoFactorSetupForm**(**args*, ***kwargs*)

The Two-factor token validation form

class flask_security.**TwoFactorSelectForm**(**args*, ***kwargs*)

class flask_security.**TwoFactorRescueForm**(**args*, ***kwargs*)

The Two-factor Rescue validation form

class flask_security.**UnifiedSigninForm**(**args*, ***kwargs*)

A unified login form For either identity/password or request and enter code.

class flask_security.**UnifiedSigninSetupForm**(**args*, ***kwargs*)

Setup form

class flask_security.**UnifiedSigninSetupValidateForm**(**args*, ***kwargs*)

The unified sign in setup validation form

class flask_security.**UnifiedVerifyForm**(**args*, ***kwargs*)

Verify authentication. This is for freshness 'reauthentication' required.

class flask_security.**VerifyForm**(**args*, ***kwargs*)

The verify authentication form

class flask_security.**WebAuthnRegisterForm**(**args*, ***kwargs*)

class flask_security.**WebAuthnRegisterResponseForm**(**args*, ***kwargs*)

class flask_security.**WebAuthnSigninForm**(**args*, ***kwargs*)

class flask_security.**WebAuthnSigninResponseForm**(**args*, ***kwargs*)

This form is used both for signin (primary/first or secondary) and verify.

class flask_security.**WebAuthnDeleteForm**(**args*, ***kwargs*)

class flask_security.**WebAuthnVerifyForm**(**args*, ***kwargs*)

class flask_security.**Form**(**args*, ***kwargs*)

class flask_security.**FormInfo**(*instantiator=<function _default_form_instantiator>*, *cls=None*)

Each view form has a name - assigned by Flask-Security. As part of every request, the form is instantiated using (usually) request.form or request.json. The default instantiator simply uses the class constructor - however applications can provide their OWN instantiator which can do pretty much anything as long as it returns an instantiated form. The 'cls' argument is optional since the instantiator COULD be form specific.

The instantiator callable will always be called from a flask request context and receive the following arguments:

```
(name, form_cls_name (optional), **kwargs)
```

kwargs will always have *formdata* and often will have *meta*. All kwargs must be passed to the underlying form constructor.

See *flask_security.Security.set_form_info()*

New in version 5.1.0.

Parameters

- **instantiator** (*Callable[[...], Form]*) –
- **cls** (*Type[Form] | None*) –

3.1.8 Signals

See the Flask documentation on signals for information on how to use these signals in your code.

Tip: Remember to add `**extra_args` to your signature so that if we add additional parameters in the future your code doesn't break.

See the documentation for the signals provided by the Flask-Login and Flask-Principal extensions. In addition to those signals, Flask-Security sends the following signals.

`user_authenticated`

Sent when a user successfully authenticates. In addition to the app (which is the sender), it is passed *user*, and *authn_via* arguments. The *authn_via* argument specifies how the user authenticated - it will be a list with possible values of `password`, `sms`, `authenticator`, `email`, `confirm`, `reset`, `register`.

New in version 3.4.0.

`user_registered`

Sent when a user registers on the site. In addition to the app (which is the sender), it is passed *user*, *confirm_token* (deprecated), *confirmation_token* and *form_data* arguments. *form_data* is a dictionary representation of registration form's content received with the registration request.

`user_not_registered`

Sent when a user attempts to register, but is already registered. This is ONLY sent when *SECURITY_RETURN_GENERIC_RESPONSES* is enabled. It is passed the following arguments:

- *user* - The existing user model
- *existing_email* - True if attempting to register an existing email
- *existing_username*- True if attempting to register an existing username
- *form_data* - the entire contents of the posted request form

New in version 5.0.0.

`user_confirmed`

Sent when a user is confirmed. In addition to the app (which is the sender), it is passed a *user* argument.

`confirm_instructions_sent`

Sent when a user requests confirmation instructions. In addition to the app (which is the sender), it is passed a *user* and *confirmation_token* arguments.

`login_instructions_sent`

Sent when passwordless login is used and user logs in. In addition to the app (which is the sender), it is passed *user* and *login_token* arguments.

`password_reset`

Sent when a user completes a password reset. In addition to the app (which is the sender), it is passed a *user* argument.

`password_changed`

Sent when a user completes a password change. In addition to the app (which is the sender), it is passed a *user* argument.

`reset_password_instructions_sent`

Sent when a user requests a password reset. In addition to the app (which is the sender), it is passed *user*, *token* (deprecated), and *reset_token* arguments.

`tf_code_confirmed`

Sent when a user performs two-factor authentication login on the site. In addition to the app (which is the sender), it is passed *user* and *method* arguments.

New in version 3.3.0.

`tf_profile_changed`

Sent when two-factor is used and user logs in. In addition to the app (which is the sender), it is passed *user* and *method* arguments.

New in version 3.3.0.

`tf_disabled`

Sent when two-factor is disabled. In addition to the app (which is the sender), it is passed *user* argument.

New in version 3.3.0.

`tf_security_token_sent`

Sent when a two factor security/access code is sent. In addition to the app (which is the sender), it is passed *user*, *method*, *login_token* and *token* (deprecated) arguments.

New in version 3.3.0.

`us_security_token_sent`

Sent when a unified sign in access code is sent. In addition to the app (which is the sender), it is passed *user*, *method*, *token* (deprecated), *login_token*, *phone_number*, and *send_magic_link* arguments.

New in version 3.4.0.

`us_profile_changed`

Sent when user completes changing their unified sign in profile. In addition to the app (which is the sender), it is passed *user*, *methods*, and *delete* arguments. *delete* will be set to `True` if the user removed a sign in option.

New in version 3.4.0.

Changed in version 5.0.0: Added delete argument and changed *method* to *methods* which is now a list.

`wan_registered`

Sent when a WebAuthn credential was successfully created. In addition to the app (which is the sender), it is passed *user* and *name* arguments.

New in version 5.0.0.

`wan_deleted`

Sent when a WebAuthn credential was deleted. In addition to the app (which is the sender), it is passed *user* and *name* arguments.

New in version 5.0.0.

CHAPTER

FOUR

ADDITIONAL NOTES

4.1 Contributing

Contributions are welcome. If you would like add features or fix bugs, please review the information below.

One source of history or ideas are the bug reports. There you can find ideas for requested features, or the remains of rejected ideas.

If you have a 'big idea' - please file an issue first so it can be discussed prior to you spending a lot of time developing. New features need to be generally useful - if your feature has limited applicability, consider making a small change that ENABLES your feature, rather than trying to get the entire feature into Flask-Security.

4.1.1 Checklist

- All new code and bug fixes need unit tests
- If you change/add to the external API be sure to update docs/openapi.yaml
- Additions to configuration variables and/or messages must be documented
- Make sure any new public API methods have good docstrings, are picked up by the api.rst document, and are exposed in __init__.py if appropriate.
- Add appropriate info to CHANGES.rst

4.1.2 Getting the code

The code is hosted on a GitHub repo at https://github.com/Flask-Middleware/flask-security. To get a working environment, follow these steps:

1. (Optional, but recommended) Create a Python 3.6 (or greater) virtualenv to work in, and activate it.
2. **Fork the repo Flask-Security**
   (look for the "Fork" button).
3. Clone your fork locally:

```
$ git clone https://github.com/<your-username>/flask-security
```

4. Change directory to flask_security:

```
$ cd flask_security
```

5. Install the requirements:

```
$ pip install -r requirements/dev.txt
```

6. Install pre-commit hooks:

```
$ pre-commit install
```

7. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

8. Develop the Feature/Bug Fix and edit
9. Write Tests for your code in:

```
tests/
```

10. When done, verify unit tests, syntax etc. all pass:

```
$ pip install -r requirements/tests.txt
$ sphinx-build docs docs/_build/html
$ tox -e compile_catalog
$ pytest tests
$ pre-commit run --all-files
```

11. Use tox:

```
$ tox  # run everything CI does
$ tox -e py38-low  # make sure works with older dependencies
$ tox -e style  # run pre-commit/style checks
```

12. When the tests are successful, commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

13. Submit a pull request through the GitHub website.
14. Be sure that the CI tests and coverage checks pass.

4.1.3 Updating the Swagger API document

When making changes to the external API, you need to update the openapi.yaml formal specification. To do this - install the swagger editor locally:

```
$ npm -g install swagger-editor-dist http-server
```

Then in a browser navigate to:

```
file:///usr/local/lib/node_modules/swagger-editor-dist/index.html#
```

Edit - it is a WYSIWYG editor and will show you errors. Once you save (as yaml) you need to look at what it will render as:

```
$ sphinx-build docs docs/_build/html
$ http-server -p 8081
```

Then in your browser navigate to:

```
http://localhost:8081/docs/_build/html/index.html
or
http://localhost:8081/docs/_build/html/_static/openapi_view.html
```

Please note that changing `openapi.yaml` won't re-trigger a docs build - so you might have to manually delete `docs/_build`.

4.1.4 Updating Translations

If you change any translatable strings (such as new messages, modified forms, etc.) you need to re-generate the translations:

```
$ tox -e extract_messages
$ tox -e update_catalog
$ tox -e compile_catalog
```

4.1.5 Testing

Unit tests are critical since Flask-Security is a piece of middleware. They also help other contributors understand any subtleties in the code and edge conditions that need to be handled.

Datastore

By default the unit tests use an in-memory sqlite DB to test datastores (except for MongoDatastore which uses mongomock). While this is sufficient for most changes, changes to the datastore layer require testing against a real DB (the CI tests test against postgres). It is easy to run the unit tests against a real DB instance. First of course install and start the DB locally then:

```
# For postgres
pytest --realdburl postgresql://<user>@localhost/
# For mysql
pytest --realdburl "mysql+pymysql://root:<password>@localhost/"
# For mongodb
pytest --realmongodburl "localhost"
```

Views

Much of Flask-Security is concerned with form-based views. These can be difficult to test especially translations etc. In the tests directory is a stand-alone Flask application `view_scaffold.py` that can be run and you can point your browser to it and walk through the various views.

4.2 Flask-Security Changelog

Here you can see the full list of changes between each Flask-Security release.

4.2.1 Version 5.3.3

Released December 29, 2023

Fixes

- (#893) Once again work on open-redirect vulnerability - this time due to newer Werkzeug. Addresses: CVE-2023-49438

4.2.2 Version 5.3.2

Released October 23, 2023

Fixes

- (#859) Update Quickstart to show how to properly handle SQLAlchemy connections.
- (#861) Auth Token not returned from /tf-validate. (thanks lilz-egoto)
- (#864) Fix for latest email_validator deprecation - bump minimum to 2.0.0
- (#865) Deprecate passing in the anonymous_user class (sent to Flask-Login).

4.2.3 Version 5.3.1

Released October 14, 2023

Please Note:

- If your application uses webauthn you must use pydantic < 2.0 until the issue with user_handle is resolved.
- If you want to use the latest Flask (3.0.0) you need to have Flask-Login changes - those aren't currently released - use the 'main' branch.

Fixes

- (#847) Compatability with Flask 3.0 (wangsha)
- (#829) Revert change in 5.3.0 that added a Referrer-Policy header.
- (#826) Fix error in quickstart (codycollier)
- (#835) Update Armenian translations (amkrtchyan-tmp)
- (#831) Update German translations. (sr-verde)
- (#853) Fix 'next' propagation when passed as form.next (thanks cariaso)

4.2.4 Version 5.3.0

Released July 27, 2023

This is a minor version bump due to some small backwards incompatible changes to WebAuthn, recoverability (/reset), confirmation (/confirm) and the two factor validity feature.

Fixes

- (#807) Webauthn Updates to handling of transport.
- (#809) Fix MongoDB support by eliminating dependency on flask-mongoengine. Improve MongoDB quickstart.
- (#801) Fix Quickstart for SQLAlchemy with scoped session.
- (#806) Login no longer, by default, checks for email deliverability.
- (#791) Token authentication is no longer accepted on endpoints which only allow 'session' as authentication-method. (N247S)
- (#814) /reset and /confirm and GENERIC_RESPONSES and additional form args don't mix.
- (#281) Reset password can be exploited and other OWASP improvements.
- (#817) Confirmation can be exploited and other OWASP improvements.
- (#819) Convert to pyproject.toml, build, remove setup.py/.cfg.
- (#823) the tf_validity feature now ONLY sets a cookie - and the token is no longer returned as part of a JSON response.
- (#825) Fix login/unified signin templates to properly send CSRF token. Add more tests.
- (#826) Improve Social Oauth example code.

Backwards Compatibility Concerns

- To align with the W3C WebAuthn Level2 and 3 spec - transports are now part of the registration response. This has been changed BOTH in the server code (using webauthn data structures) as well as the sample javascript code. If an application has their own javascript front end code - it might need to be changed.
- The tf_validity feature *SECURITY_TWO_FACTOR_ALWAYS_VALIDATE* used to set a cookie if the request was form based, and return the token as part of a JSON response. Now, this feature is ONLY cookie based and the token is no longer returned as part of any response.
- Reset password was changed to adhere to OWASP recommendations and reduce possible exploitation:
  - A new email (with new token) is no longer sent upon expired token. Users must restart the reset password process.
  - The user is no longer automatically logged in upon successful password reset. For backwards compatibility *SECURITY_AUTO_LOGIN_AFTER_RESET* can be set to `True`. Note that this compatibility feature is deprecated and will be removed in a future release.
  - Identity information (identity, email) is no longer sent as part of the URL redirect query params.
  - The SECURITY_MSG_PASSWORD_RESET_EXPIRED message no longer contains the user's identity/email.
  - The default for *SECURITY_RESET_PASSWORD_WITHIN* has been changed from *5 days* to *1 days*.

  - The response to GET /reset/<token> sets the HTTP header *Referrer-Policy* to *no-referrer* as suggested by OWASP. *PLEASE NOTE: this was backed out in 5.3.1*

- Confirm email was changed to adhere to OWASP recommendations and reduce possible exploitation:
  - A new email (with new token) is no longer sent upon expired token. Users must restart the confirmation process.
  - Identity information (identity, email) is no longer sent as part of the URL redirect query params.
  - The `SECURITY_AUTO_LOGIN_AFTER_CONFIRM` configuration variable now defaults to `False` - meaning after a successful email confirmation, the user must still sign in using the usual mechanisms. This is to align better with OWASP best practices. Setting it to `True` will restore prior behavior.
  - The SECURITY_MSG_CONFIRMATION_EXPIRED message no longer contains the user's identity/email.
  - The response to GET /reset/<token> sets the HTTP header *Referrer-Policy* to *no-referrer* as suggested by OWASP. *PLEASE NOTE: this was backed out in 5.3.1*

4.2.5 Version 5.2.0

Released May 6, 2023

Note: Due to rapid deprecation and removal of APIs from the Pallets team, maintaining the testing of back versions of various packages is taking too much time and effort. In this release only current versions of the various dependent packages are being tested.

Fixes

- (#764) Remove old Werkzeug compatibility check.
- (#777) Compatibility with Quart.
- (#780) Remove dependence on pkg_resources / setuptools (use importlib_resources package)
- (#792) Fix tests to work with latest Werkzeug/Flask. Update requirements_low to match current releases.
- (#792) Drop support for Python 3.7

Known Issues

- Flask-mongoengine hasn't released in a while and currently will not work with latest Flask and Flask-Security-Too (this is due to the JSONEncoder being deprecated and removed).

Backwards Compatibility Concerns

- The removal of pkg_resources required changing the config variable `SECURITY_I18N_DIRNAME`. If your application modified or extended this configuration variable, a small change will be required.

4.2.6 Version 5.1.2

Released March 12, 2023

Fixes

- (#771) Hungarian translations not working.
- (#769) Fix documentation for send_mail. (gg)
- (#768) Fix for latest mongoengine and mongomock.
- (#766) Fix inappropriate use of &thinsp& in French translations. (maxdup)
- (#773) Improve documentation around subclassing forms.

4.2.7 Version 5.1.1

Released March 1, 2023

Fixes

- (#740) Fix 2 Flask apps in same thread with USERNAME_ENABLE set. There was a too aggressive config check.
- (#739) Update Russian translations. (ademaro)
- (#743) Run all templates through a linter. (ademaro)
- (#757) Fix json/flask backwards compatibility hack.
- (#759) Fix quickstarts - make sure they run using *flask run*
- (#755) Fix unified signup when two-factor not enabled. (sebdroid)
- (#763) Add dependency on setuptools (pkg_resources). (hroncok)

4.2.8 Version 5.1.0

Released January 23, 2023

Features

- (#667) Expose form instantiation. See *Controlling Form Instantiation*.
- (#693) Option to encrypt recovery codes.
- (#716) Support for authentication via ‘social’ oauth.
- (#721) Support for Python 3.11

Fixes

- (#678) Fixes for Flask-SQLAlchemy 3.0.0. (jrast)
- (#680) Fixes for sqlalchemy 2.0.0 (jrast)
- (#697) Webauthn and Unified signin features now properly take into account blueprint prefixes.
- (#699) Properly propagate *?next=/xx* - the verify, webauthn, and unified signin endpoints, that had multiple redirects, needed fixes.
- (#696) Add Hungarian translations. (xQwexx)
- (#701) Two factor redirects ignored url_prefix. Added a *SECURITY_TWO_FACTOR_ERROR_VIEW* configuration option.
- (#704) Add configurations for static folder/URL and make sure templates reference blueprint relative static folder.
- (#709) Make (some) templates look better by using single quotes instead of double quotes.
- (#690) Send entire context to MailUtil::send_mail (patrickyan)
- (#728) Support for Flask-Babel 3.0.0
- (#692) Add configuration option *SECURITY_TWO_FACTOR_POST_SETUP_VIEW* which is redirected to upon successful change of a two factor method.
- (#733) The ability to pass in a LoginManager instance which was deprecated in 5.0 has been removed.
- (#732) If *SECURITY_USERNAME_REQUIRED* was `True` then users couldn't login with just an email.
- (#734) If *SECURITY_USERNAME_ENABLE* is set, bleach is a requirement.
- (#736) The unauthz_handler now takes a function name, not the function!

Backwards Compatibility Concerns

- Each form class used to be set as an attribute on the Security object. With the new form instantiation model, they no longer are.
- After a successful update/change of a two-factor method, the user was redirected to *SECURITY_POST_LOGIN_VIEW*. Now it redirects to *SECURITY_TWO_FACTOR_POST_SETUP_VIEW* which defaults to *".two_factor_setup"*.
- The *Security.unauthz_handler()* now takes a function name - not the function - which never made sense.

4.2.9 Version 5.0.2

Released September 23, 2022

Fixes

- (#673) Role permissions backwards compatibility bug. For SQL based datastores that use Flask-Security's models.fsqla_vx - there should be NO issues. If you declare your own models - please see the 5.0.0 releases notes for required change.

4.2.10 Version 5.0.1

Released September 6, 2022

Fixes

- (#662) Fix Change Password regression. (tysonholub)

4.2.11 Version 5.0.0

Released August 27, 2022

PLEASE READ CHANGE NOTES CAREFULLY - THERE ARE LIKELY REQUIRED CHANGES YOU WILL HAVE TO MAKE.

Features

- (#475) Support for WebAuthn.
- (#479) Support Two-factor recovery codes.
- (#585) Provide option to prevent user enumeration (i.e. Generic Responses).
- (#532) Support for Python 3.10.
- (#657, #655) Support for Flask >= 2.2.
- (#540) Improve Templates in support of JS required by WebAuthn.
- (#608) Add Icelandic translations. (ofurkusi)
- (#650) Update German translations. (sr-verde)
- (#256) Add custom HTML attributes to improve user experience. This changed LoginForm quite a bit - please see backwards compatability concerns below. The default LoginForm and template should be the same as before.
- (#638) The JSON errors response has been unified. Please see backwards compatibility concerns below.
- Updated all-inclusive data models (fsqla_v3). Add fields necessary for the new WebAuthn and Two-Factor recovery codes features. Changed *us_phone_number* to be unique (but not required). Changed *password* to be nullable.

Deprecations

- (#568) Deprecate the old passwordless feature in favor of Unified Signin.
- (#568) Deprecate replacing login_manager so we can possibly vendor that in in the future.
- (#654) The previously deprecated methods RoleMixin.add_permissions and RoleMixin.remove_permissions have been removed.
- (#657) The ability to pass in a json_encoder_cls as part of initialization has been removed since Flask 2.2 has deprecated and replaced that functionality.
- (#655) Flask has deprecated @before_first_request. This was used mostly in examples/quickstart. These have been changed to use app.app_context() prior to running the app. Flask-Security itself used it in 2 places - to populate _ in jinja globals if Babel wasn't initialized and to perform various configuration sanity checks w.r.t. WTF CSRF. All Flask-Security templates have been converted to use *_fsdomain* rather than `_` so Flask-Security no longer sets `_` into jinja2 globals. The configuration checks have been moved to the end of Security::init_app() - so it is now imperative that *FlaskWTF::CSRFProtect()* be called PRIOR to initializing Flask-Security.
- encrypt_password method has been removed. It has been deprecated since 2.0.2
- get_token_status has been deprecated.

Fixes

- (#591) Make the required zxcvbn complexity score configurable. (mephi42)
- (#531) Get rid of Flask-Mail. Flask-Mailman is now the default preferred email package. Flask-Mail is still supported so there should be no backwards compatability issues.
- (#597) A delete option has been added to us-setup (form and view).
- (#625) Improve username support - the LoginForm now has a separate field for username if `SECURITY_USERNAME_ENABLE` is True, and properly displays input fields only if the associated field is an identity attribute (as specified by *`SECURITY_USER_IDENTITY_ATTRIBUTES`*).
- (#627) Improve empty password handling. Prior, an unguessable password was set into the user record when a user registered without a password - now, the DB user model has been changed to allow nullable passwords. This provides a better user experience since Flask-Security now knows if a user has an empty password or not. Since registering without a password is not a mainstream feature, a new configuration variable *`SECURITY_PASSWORD_REQUIRED`* has been added (defaults to `True`).
- (#479) A new configuration option *`SECURITY_TWO_FACTOR_RESCUE_EMAIL`* has been added that allows disabling that feature - defaults to backwards compatible `True`
- (#658) us_phone_number needs to be validated to be unique.

Backward Compatibility Concerns

For unified signin:

- The redirect after a successful us-setup used to redirect to `SECURITY_US_POST_SETUP_VIEW` or `SECURITY_POST_LOGIN_VIEW` (which would default to '/'). Now it just redirects to `SECURITY_US_POST_SETUP_VIEW` which defaults back to the `/us-setup` view.
- The ability to authenticate using a one-time email link was automatically setup by the system for all users. "email" now behaves like the other unified sign in methods and must be explicitly set up - with the exception that if a user registers WITHOUT a password, the system will setup the one-time email link option - since otherwise the user would never be able to authenticate.

- `/us-signin/send-code` didn't used to check if the user account required confirmation it just sent a code and the `/us-signin` endpoint did the confirmation check. Now `send-code` does the confirmation check and won't send a code unless the user is confirmed.
- In `us-verify` the 'code_methods' item now lists just active/setup methods that generate a code not ALL possible methods that generate a code.
- `SECURITY_US_VERIFY_SEND_CODE_URL` and `SECURITY_US_SIGNIN_SEND_CODE_URL` endpoints are now POST only.
- Empty passwords were always permitted when `SECURITY_UNIFIED_SIGNIN` was enabled - now an additional configuration variable `SECURITY_PASSWORD_REQUIRED` must be set to False.
- `SECURITY_US_VERIFY_SEND_CODE_URL` and `SECURITY_US_SIGNIN_SEND_CODE_URL` used to send `code_sent` to the template. Now they flash the `SECURITY_MSG_CODE_HAS_BEEN_SENT` message.
- With the addition of being able to delete a previously setup up sign in method, the signal *us_profile_changed* arguments have changed. *method* is now *methods* and is a list, and a new argument *delete* is True if a sign in option was deleted.

Login:

- Since the beginning of time, the flask-security login form has accepted any input in the 'email' field, and used that to check if it corresponds to any field in `SECURITY_USER_IDENTITY_ATTRIBUTES`. This has always been problematic and confusing - and with the addition of HTML attributes for various form fields - having a field with multiple possible inputs is no longer a viable user experience. This is no longer supported, and the LoginForm now declares the `email` field to be of type `EmailField` which requires a valid (after normalization) email address. The most common usage of this legacy feature was to allow an email or username - Flask-Security now has core support for a `username` option - see *SECURITY_USERNAME_ENABLE*. Please see *Customizing the Login Form* for an example of how to replicate the legacy behavior.
- Some error messages have changed - `USER_DOES_NOT_EXIST` is now returned for any identity error including an empty value.

Other:

- A very old piece of code in registrable, would immediately commit to the DB when a new user was created. It is now consistent with all other views, and has the caller responsible for committing the transaction - usually by setting up a flask `after_this_request` action. This could affect an application that captured the registration signal and stored the `user` object for later use - this user object would likely be invalid after the request is finished.
- Some fields have custom HTML attributes attached to them (e.g. autocomplete, type, etc). These are stored as part of the form in the `render_kw` attribute. This could cause some confusion if an app had its own templates and set different attributes.
- The keys for "/tf-rescue" select options have changed to be more 'action' oriented:
  - *lost_device* -> *email*
  - *no_mail_access* -> *help*
- JSON error responses. **THIS IS A BREAKING CHANGE**. In earlier releases, the JSON error response could have either a *error* key which was for rare cases where there was a single non-form related error, or an *errors* key which was a a dict as defined by WTForms. Now, the *errors* key will contain a list of (localized) messages - both non-form related as well as any form related. The key *field_errors* will contain the dict as specified by WTForms. Please note that starting with WTForms 3.0 form-level errors are supported and show up in the dict with the field name/key of "none". There are no changes to non-error related JSON responses.
- Permissions **THIS IS A BREAKING CHANGE**. The Role Model now stores permissions as a list, and requires that the underlying DB ORM map that to a supported DB type. For SQLAlchemy, this is mapped to a comma separated string (as before). For SQLAlchemy DBs the underlying Column type (UnicodeText) didn't change so

no data migration should be required. However, the ORM Column type did change and requires the following change to your model:

```
from flask_security import AsaList
from sqlalchemy.ext.mutable import MutableList
class Role(Base, RoleMixin):
    ...
    permissions = Column(MutableList.as_mutable(AsaList()), nullable=True)
    ...
```

If your application makes use of Flask-Security's models.fsqla_vX classes - no changes are required. For Mongo, a ListField can be directly used.

- CSRF - As mentioned above, it is now required that *FlaskWTF::CSRFProtect()*, if used, must be called PRIOR to initializing Flask-Security.
- json_encoder_cls - As mentioned above - Flask-Security initialization no longer accepts overriding the json_encoder class. If this is required, update to Flask >=2.2 and implement Flask's JSONProvider interface.

For templates:

- Pretty much every template was modified to replace <p> with <div class=xx> to make styling possible and to make more complex forms more readable.
- Many forms had places where things weren't properly localizable - that has (hopefully) been fixed.
- The `us_setup.html` template was modified to add ability to delete an existing set up method.

DB Migration

To use the new WebAuthn feature a new table and two new columns in the User model are required. To ease updates - Flask-Security will automatically create a fs_webauthn_user_handle upon first use for existing users. If you are using Alembic the schema migration is easy:

```
op.add_column('user', sa.Column('fs_webauthn_user_handle', sa.String(length=64),␣
↪nullable=True, unique=True))
```

If you want to allow for empty passwords as part of registration then set *SECURITY_PASSWORD_REQUIRED* to `False`. In addition you need to change your DB schema to allow the `password` field to be nullable.

4.2.12 Version 4.1.5

Released July 28, 2022

Fixes

- (#644) Fix test and other failures with newer Flask-Login/Werkzeug versions.

4.2.13 Version 4.1.4

Released April 19, 2022

Fixes

- (#594) Fix test failures with newer Flask versions.

4.2.14 Version 4.1.3

Released March 2, 2022

Fixes

- (#581) Fix bug when attempting to disable register_blueprint. (halali)
- (#539) Fix example documentation re: generating localized messages. (kazuhei2)
- (#546) Make roles joinedload compatible with SQLAlchemy 2.0. (keats)
- (#586) Ship py.typed as part of package.
- (#580) Improve documentation around use of bleach and include in common install extra.

4.2.15 Version 4.1.2

Released September 22, 2021

Fixes

- (#526) default_reauthn_handler doesn't honor SECURITY_URL_PREFIX
- (#528) Improve German translations (sr-verde)
- (#527) Fix two-factor sample code (djpnewton)

4.2.16 Version 4.1.1

Released September 10, 2021

Fixes

- (#518) Fix corner case where Security object was being reused in tests.
- (#512) If USERNAME_ENABLE is set, change LoginForm field from EmailField to StringField. Also - dynamically add fields to Login and Registration forms rather than always having them - this made the RegistrationForm much simpler.
- (#516) Improved username feature handling solved issue of always requiring bleach.
- (#513) Improve documentation of default username validation.

4.2.17 Version 4.1.0

Released July 23, 2021

Features

- (#474) Add public API and CLI command to change a user's password.
- (#140) Add type hints. Please note that many of the packages that flask-security depends on aren't typed yet - so there are likely errors in some of the types.
- (#466) Add first-class support for using username for signing in.

Fixes

- (#483) 4.0 doesn't accept 3.4 authentication tokens. (kuba-lilz)
- (#490) Flask-Mail sender name can be a tuple. (hrishikeshrt)
- (#486) Possible open redirect vulnerability.
- (#478) Improve/update German translation. (sr-verde)
- (#488) Improve handling of Babel packages.
- (#496) Documentation improvements, distribution extras, fix single message override.
- (#497) Improve cookie handling and default `samesite` to `Strict`.

Backwards Compatibility Concerns

- (#488) In 4.0.0, with the addition of Flask-Babel support, Flask-Security enforced that if it could import either Flask-Babel or Flask-BabelEx, that those modules had been initialized as proper Flask extensions. Prior to 4.0.0, just Flask-BabelEx was supported - and that didn't require any explicit initialization. Flask-Babel DOES require explicit initialization. However for some applications that don't completely control their environment (such as system pre-installed versions of python) this caused applications that didn't even want translation services to fail on startup. With this release, Flask-Security still attempts to import one or the other package - however if those modules are NOT initialized, Flask-Security will simply ignore them and no translations will occur.
- (#497) The CSRF_COOKIE and TWO_FACTOR_VALIDITY cookie had their defaults changed to set `samesite=Strict`. This follows the Flask-Security goal of making things more secure out-of-the-box.
- (#140) Type hinting. For the most part this of course has no runtime effects. However, this required a fairly major overhaul of how Flask-Security is initialized in order to provide valid types for the many constructor attributes. There are no known compatability concerns - however initialization used to convert all arguments into kwargs then add those as attributes and merge with application constants. That no longer happens and it is possible that some corner cases don't behave precisely as they did before.

4.2.18 Version 4.0.1

Released April 2, 2021

Features

Fixes

- (#461) 4.0 doesn't accept 3.4 authentication tokens. (kuba-lilz)
- (#460) 2-fa error: Failed to send code - improved documentation and debuggability.
- (#454) 2-fa error: TypeError - fixed documentation.
- (#443) Calling create user without any arguments - fixed underlying cause of translating form errors in the CLI.
- (#442) Email validation confusion - added documentation.
- (#450) Add documentation on how to override specific error messages.
- (#439) Don't install global-scope tests. (mgorny)
- (#470) Add note about updating DB using MySQL. (jugmac00)
- (#468) Fix documentation - uia_phone_number should be uia_phone_mapper. (dvrg)
- (#457) Improve chinese translations. (zxjlm)
- (#453) Improve basque and spanish translations. (mmozos)
- (#448) Add Afrikaans translations. (lonelyvikingmichael)
- (#467) Add Blinker as explicit dependency, improve/fix celery usage docs, dont require pyqrcode unless authenticator configured, improve SMS configuration variables documentation.

4.2.19 Version 4.0.0

Released January 26, 2021

PLEASE READ CHANGE NOTES CAREFULLY - THERE ARE LIKELY REQUIRED CHANGES YOU WILL HAVE TO MAKE TO EVEN START YOUR APPLICATION WITH 4.0

Start Here

- Your UserModel must contain `fs_uniquifier`
- Either uninstall Flask-BabelEx (if you don't need translations) or add either Flask-Babel (>=2.0) or Flask-BabelEx to your dependencies AND be sure to initialize it in your app.
- Add Flask-Mail to your dependencies.
- If you have unicode emails or passwords read change notes below.

4.2.20 Version 4.0.0rc2

Released January 18, 2021

Features & Cleanup

- Removal of python 2.7 and <3.6 support
- Removal of token caching feature (a relatively new feature that had some systemic issues)
- (#328) Remove dependence on Flask-Mail and refactor.
- (#335) Remove two-factor */tf-confirm* endpoint and use generic *freshness* mechanism.
- (#336) Remove `SECURITY_BACKWARDS_COMPAT_AUTH_TOKEN_INVALID(ATE)`. In addition to not making sense - the documentation has never been correct.
- (#339) Require `fs_uniquifier` in the UserModel and stop using/referencing the UserModel primary key.
- (#349) Change `SECURITY_USER_IDENTITY_ATTRIBUTES` configuration variable semantics.
- Remove (all?) requirements around having an 'email' column in the UserModel. API change - JSON SPA redirects used to always include a query param 'email=xx'. While that is still sent (if and only if) the UserModel contains an 'email' columns, a new query param 'identity' is returned which returns the value of *`UserMixin.calc_username()`*.
- (#382) Improvements and documentation for two-factor authentication.
- (#394) Add support for email validation and normalization (see *`MailUtil`*).
- (#231) Normalize unicode passwords (see *`PasswordUtil`*).
- (#391) Option to redirect to */confirm* if user hits an endpoint that requires confirmation. New option *`SECURITY_REQUIRES_CONFIRMATION_ERROR_VIEW`* which if set and the user hits the */login*, */reset*, or */us-signin* endpoint, and they require confirmation the response will be a redirect. (SnaKyEyeS)
- (#366) Allow redirects on sub-domains. Please see *`SECURITY_REDIRECT_ALLOW_SUBDOMAINS`*. (willcroft)
- (#376) Have POST redirects default to Flask's `APPLICATION_ROOT`. Previously the default configuration was `/`. Now it first looks at Flask's *APPLICATION_ROOT* configuration and uses that (which also by default is `/`. (tysonholub)
- (#401) Add 2FA Validity Window so an application can configure how often the second factor has to be entered. (baurt)
- (#403) Add HTML5 Email input types to email fields. This has some backwards compatibility concerns outlined below. (drola)
- (#413) Add hy_AM translations. (rudolfamirjanyan)
- (#410) Add Basque and fix Spanish translations. (mmozos)
- (#408) Polish translations. (kamil559)
- (#390) Update ru_RU translations. (TitaniumHocker)

Fixed

- (#389) Fixes for translations. First - email subjects were never being translated. Second, converted all templates to use _fsdomain(xx) rather than _(xx) so that they get translated regardless of the app's domain.
- (#381) Support Flask-Babel 2.0 which has backported Domain support. Flask-Security now supports Flask-Babel (>=2.00), Flask-BabelEx, as well as no translation support. Please see backwards compatibility notes below.
- (#352) Fix issue with adding/deleting permissions - all mutating methods must be at the datastore layer so that db.put() can be called. Added `UserDatastore.add_permissions_to_role()` and `UserDatastore.remove_permissions_from_role()`. The methods *.RoleMixin.add_permissions* and *.RoleMixin.remove_permissions* have been deprecated.
- (#395) Provide ability to change table names for User and Role tables in the fsqla model.
- (#338) All sessions are invalidated when a user changes or resets their password. This is accomplished by changing the user's *fs_uniquifier*. The user is automatically re-logged in (and a new session created) after a successful change operation.
- (#418) Two-factor (and to a lesser extent unified sign in) QRcode fetching wasn't protected via CSRF. The fix makes things secure and simpler (always good); however read below for compatibility concerns. In addition, the elements that make up the QRcode (key, username, issuer) area also made available to the form and returned as part of the JSON return value - this allows for manual or other ways to initialize the authenticator app.
- (#421) GET on */login* and */change* could return the callers authentication_token. This is a security concern since GETs don't have CSRF protection. This bug was introduced in 3.3.0.

Backwards Compatibility Concerns

- (#328) Remove dependence on Flask-Mail and refactor. The `send_mail_task` and `send_mail` methods as part of Flask-Security initialization have been removed and replaced with a new *`MailUtil`* class. The utility method *`send_mail()`* can still be used. If your application didn't use either of the deprecated methods, then the only change required is to add Flask-Mail to your package requirements (since Flask-Security no longer lists it). Please see the *Emails* for updated examples.
- (#335) Convert two-factor setup flow to use the freshness feature rather than its own verify password endpoint. This COMPLETELY removes the `/tf-confirm` endpoint and associated form: `two_factor_verify_password_form`. Now, when /tf-setup is invoked, the *`flask_security.check_and_update_authn_fresh()`* is invoked, and if the current session isn't 'fresh' the caller will be redirected to a verify endpoint (either *`SECURITY_VERIFY_URL`* or *`SECURITY_US_VERIFY_URL`*). The simplest change would be to call `/verify` everywhere the application used to call `/tf-confirm`.
- (#339) Require `fs_uniquifier`. In 3.3 the `fs_uniquifier` was added in the UserModel to fix the slow authentication token issue. In 3.4 the `fs_uniquifier` was used to implement Flask-Login's *Alternative Token* feature - thus decoupling the primary key (id) from any security context. All along, there have been a few issues with applications not wanting to use the name 'id' in their model, or wanting a different type for their primary key. With this change, Flask-Security no longer interprets or uses the UserModel primary key - just the `fs_uniquifier` field. See the changes section for 3.3 for information on how to do the schema and data upgrades required to add this field. There is also an API change - the JSON response (via UserModel.get_security_payload()) returned the `user.id` field. With this change the default is an empty directory - override *`UserMixin.get_security_payload()`* to return any portion of the UserModel you need.
- (#349) *`SECURITY_USER_IDENTITY_ATTRIBUTES`* has changed syntax and semantics. It now contains the combined information from the old `SECURITY_USER_IDENTITY_ATTRIBUTES` and the newly introduced in 3.4 *`SECURITY_USER_IDENTITY_MAPPINGS`*. This enabled changing the underlying way we validate credentials in the login form and unified sign in form. In prior releases we simply tried to look up the form value as the PK of the UserModel - this often failed and then looped through the other `SECURITY_USER_IDENTITY_ATTRIBUTES`. This had a history of issues, including many applications not wanting to have a standard PK for the user model.

Now, using the mapping configuration, the UserModel attribute/column the input corresponds to is determined, then the UserModel is queried specifically for that *attribute:value* pair. If you application didn't change the variable, no modifications are required.

- (#354) The *flask_security.PhoneUtil* is now initialized as part of Flask-Security initialization rather than `@app.before_first_request` (since that broke the CLI). Since it isn't called in an application context, the *app* being initialized is passed as an argument to *__init__*.
- (#381) When using Flask-Babel (>= 2.0) it is required that the application initialize Flask-Babel (e.g. Babel(app)). Flask-BabelEx would self-initialize so it didn't matter. Flask-Security will throw a run time error upon first request if Flask-Babel OR FLask-BabelEx is installed, but not initialized. Also, Flask-Security no longer has a dependency on either Flask-Babel or Flask-BabelEx - if neither are installed, it falls back to a dummy translation. *If your application expects translation services, it must specify the appropriate dependency AND initialize it.*
- (#394) Email input is now normalized prior to being stored in the DB. Previously, it was validated, but the raw input was stored. Normalization and validation rely on the email_validator package. The *MailUtil* class provides the interface for normalization and validation - allowing all this to be customized. If you have unicode local or domain parts - existing users may have difficulties logging in. Administratively you need to read each user record, normalize the email (see *MailUtil*), and write it back.
- (#381) Passwords are now, by default, normalized using Python's unicodedata.normalize() method. The *SECURITY_PASSWORD_NORMALIZE_FORM* defaults to "NKFD". This brings Flask-Security in line with the NIST recommendations outlined in Memorized Secret Verifiers If your users have unicode passwords they may have difficulty authenticating. You can turn off this normalization or have your users reset their passwords. Password normalization and validation has been encapsulated in a new *PasswordUtil* class. This replaces the method `password_validator` introduced in 3.4.0.
- (#403) By default all forms that have an email as input now use the wtforms html5 `EmailField`. For most applications this will make the user experience slightly nicer - especially for mobile devices. Some applications use the email form field for other identity attributes (such as username). If your application does this you will probably need to subclass `LoginForm` and change the email type back to StringField.
- (#338) By default, both passwords and authentication tokens use the same attribute `fs_uniquifier` to uniquely identify the user. This means that if the user changes or resets their password, all authentication tokens also become invalid. This could be viewed as a feature or a bug. If this behavior isn't desired, add another uniquifier: `fs_token_uniquifier` to your UserModel and that will be used to generate authentication tokens.
- (#418) Fix CSRF vulnerability w.r.t. getting QRcodes. Both two-factor and unified-signup had a separate GET endpoint to fetch the QRcode when setting up an authenticator app. GETS don't have any CSRF protection. Both of those endpoints have been completely removed, and the QRcode is embedded in a successful POST of the setup form. The changes to the templates are minimal and of course if you didn't override the template - there is no compatibility concern.
- (#421) Fix CSRF vulnerability on */login* and */change* that could return the callers authentication token. Now, callers can only get the authentication token on successful POST calls.

4.2.21 Version 3.4.5

Released January 8, 2021

Security Vulnerability Fix.

Two CSRF vulnerabilities were reported: qrcode and login. This release fixes the more severe of the 2 - the */login* vulnerability. The QRcode issue has a much smaller risk profile since a) it is only for two-factor authentication using an authenticator app b) the qrcode is only available during the time the user is first setting up their authentication app. The QRcode issue has been fixed in 4.0.

Fixed

- (#421) GET on */login* and */change* could return the callers authentication_token. This is a security concern since GETs don't have CSRF protection. This bug was introduced in 3.3.0.

Backwards Compatibility Concerns

- (#421) Fix CSRF vulnerability on */login* and */change* that could return the callers authentication token. Now, callers can only get the authentication token on successful POST calls.

4.2.22 Version 3.4.4

Released July 27, 2020

Bug/regression fixes.

Fixed

- (#359) Basic Auth broken. When the unauthenticated handler was changed to provide a more uniform/consistent response - it broke using Basic Auth from a browser, since it always redirected rather than returning 401. Now, if the response headers contain `WWW-Authenticate` (which is set if `basic` @auth_required method is used), a 401 is returned. See below for backwards compatibility concerns.
- (#362) As part of figuring out issue 359 - a redirect loop was found. In release 3.3.0 code was put in to redirect to *SECURITY_POST_LOGIN_VIEW* when GET or POST was called and the caller was already authenticated. The method used would honor the request `next` query parameter. This could cause redirect loops. The pre-3.3.0 behavior of redirecting to *SECURITY_POST_LOGIN_VIEW* and ignoring the `next` parameter has been restored.
- (#347) Fix peewee. Turns out - due to lack of unit tests - peewee hasn't worked since 'permissions' were added in 3.3. Furthermore, changes in 3.4 around get_id and alternative tokens also didn't work since peewee defines its own *get_id* method.

Compatibility Concerns

In 3.3.0, *flask_security.auth_required()* was changed to add a default argument if none was given. The default include all current methods - `session`, `token`, and `basic`. However `basic` really isn't like the others and requires that we send back a `WWW-Authenticate` header if authentication fails (and return a 401 and not redirect). `basic` has been removed from the default set and must once again be explicitly requested.

4.2.23 Version 3.4.3

Released June 12, 2020

Minor fixes for a regression and a couple other minor changes

Fixed

- (#340) Fix regression where tf_phone_number was required, even if SMS wasn't configured.
- (#342) Pick up some small documentation fixes from 4.0.0.

4.2.24 Version 3.4.2

Released May 2, 2020

Only change is to move repo to the Flask-Middleware github organization.

4.2.25 Version 3.4.1

Released April 22, 2020

Fix a bunch of bugs in new unified sign in along with a couple other major issues.

Fixed

- (#298) Alternative ID feature ran afoul of postgres/psycopg2 finickiness.
- (#300) JSON 401 responses had WWW-Authenticate Header attached - that caused browsers to pop up their own login/password form. Not what applications want.
- (#280) Allow admin/api to setup TFA (and unified sign in) out of band. Please see *`UserDatastore.tf_set()`*, *`UserDatastore.tf_reset()`*, *`UserDatastore.us_set()`*, *`UserDatastore.us_reset()`* and *`UserDatastore.reset_user_access()`*.
- (#305) We used form._errors which wasn't very pythonic, and it was removed in WTForms 2.3.0.
- (#310) WTForms 2.3.0 made email_validator optional - we need it.

4.2.26 Version 3.4.0

Released March 31, 2020

Features

- (#257) Support a unified sign in feature. Please see *Unified Sign In*.
- (#265) Add phone number validation class. This is used in both unified sign in as well as two-factor when using `sms`.
- (#274) Add support for 'freshness' of caller's authentication. This permits endpoints to be additionally protected by ensuring a recent authentication.
- (#99, #195) Support pluggable password validators. Provide a default validator that offers complexity and breached support.
- (#266) Provide interface to two-factor send_token so that applications can provide error mitigation. Defaults to returning errors if can't send the verification code.
- (#247) Updated all-inclusive data models (fsqlaV2). Add fields necessary for the new unified sign in feature and changed 'username' to be unique (but not required).

- (#245) Use fs_uniquifier as the default Flask-Login 'alternative token'. Basically this means that changing the fs_uniquifier will cause outstanding auth tokens, session and remember me cookies to be invalidated. So if an account gets compromised, an admin can easily stop access. Prior to this cookies were storing the 'id' which is the user's primary key - difficult to change! (kishi85)

Fixed

- (#273) Don't allow reset password for accounts that are disabled.
- (#282) Add configuration that disallows GET for logout. Allowing GET can cause some denial of service issues. The default still allows GET for backwards compatibility. (kantorii)
- (#258) Reset password wasn't integrated into the two-factor feature and therefore two-factor auth could be bypassed.
- (#254) Allow lists and sets as underlying permissions. (pffs)
- (#251) Allow a registration form to have additional fields that aren't part of the user model that are just passed to the user_registered.send signal, where the application can perform arbitrary additional actions required during registration. (kuba-lilz)
- (#249) Add configuration to disable the 'role-joining' optimization for SQLAlchemy. (pffs)
- (#238) Fix more issues with atomically setting the new TOTP secret when setting up two-factor. (kishi85)
- (#240) Fix Quart Compatibility. (ristellise)
- (#232) CSRF Cookie not being set when using 'Remember Me' cookie to re-sign in. (kishi85)
- (#229) Two-factor enabled accounts didn't work with the Remember Me feature. (kishi85)

As part of adding unified sign in, there were many similarities with two-factor. Some refactoring was done to unify naming, configuration variables etc. It should all be backwards compatible.

- In TWO_FACTOR_ENABLED_METHODS "mail" was changed to "email". "mail" will still be honored if already stored in DB. Also "google_authenticator" is now just "authenticator".
- TWO_FACTOR_SECRET, TWO_FACTOR_URI_SERVICE_NAME, TWO_FACTOR_SMS_SERVICE, and TWO_FACTOR_SMS_SERVICE_CONFIG have all been deprecated in favor of names that are the same for two-factor and unified sign in.

Other changes with possible backwards compatibility issues:

- `/tf-setup` never did any phone number validation. Now it does.
- `two_factor_setup.html` template - the chosen_method check was changed to `email`. If you have your own custom template - be sure make that change.

4.2.27 Version 3.3.3

Released February 11, 2020

Minor changes required to work with latest released Werkzeug and Flask-Login.

4.2.28 Version 3.3.2

Released December 7, 2019

- (#215) Fixed 2FA totp secret regeneration bug (kishi85)
- (#172) Fixed 'next' redirect error in login view
- (#221) Fixed regressions in login view when already authenticated user again does a GET or POST.
- (#219) Added example code for unit testing FS protected routes.
- (#223) Integrated two-factor auth into registration and confirmation.

Thanks to kuba-lilz and kishi85 for finding and providing detailed issue reports.

In Flask-Security 3.3.0 the login view was changed to allow already authenticated users to access the view. Prior to 3.3.0, the login view was protected with @anonymous_user_required - so any access (via GET or POST) would simply redirect the user to the `POST_LOGIN_VIEW`. With the 3.3.0 changes, both GET and POST behaved oddly. GET simply returned the login template, and POST attempted to log out the current user, and log in the new user. This was problematic since this couldn't possibly work with CSRF. The old behavior has been restored, with the subtle change that older Flask-Security releases did not look at "next" in the form or request for the redirect, and now, all redirects from the login view will honor "next".

4.2.29 Version 3.3.1

Released November 16, 2019

- (#197) Add Quart compatibility (Ristellise)
- (#194) Add Python 3.8 support into CI (jdevera)
- (#196) Improve docs around Single Page Applications and React (acidjunk)
- (#201) fsqla model was added to __init__.py making Sqlalchemy a required package. That is wrong and has been removed. Applications must now explicitly import from `flask_security.models`
- (#204) Fix/improve examples and quickstart to show one MUST call hash_password() when creating users programmatically. Also show real SECRET_KEYs and PASSWORD_SALTs and how to generate them.
- (#209) Add argon2 as an allowable password hash.
- (#210) Improve integration with Flask-Admin. Actually - this PR improves localization support by adding a method `_fsdomain` to jinja2's global environment. Added documentation around localization.

4.2.30 Version 3.3.0

Released September 26, 2019

There are several default behavior changes that might break existing applications. Most have configuration variables that restore prior behavior.

If you use Authentication Tokens (rather than session cookies) you MUST make a (small) change. Please see below for details.

- (#120) Native support for Permissions as part of Roles. Endpoints can be protected via permissions that are evaluated based on role(s) that the user has.
- (#126, #93, #96) Revamp entire CSRF handling. This adds support for Single Page Applications and having CSRF protection for browser(session) authentication but ignored for token based authentication. Add extensive documentation about all the options.

- (#156) Token authentication is slow. Please see below for details on how to enable a new, fast implementation.
- (#130) Enable applications to provide their own `render_json()` method so that they can create unified API responses.
- (#121) Unauthorized callback not quite right. Split into 2 different callbacks - one for unauthorized and one for unauthenticated. Made default unauthenticated handler use Flask-Login's unauthenticated method to make everything uniform. Extensive documentation added. *.Security.unauthorized_callback* has been deprecated.
- (#120) Add complete User and Role model mixins that support all features. Modify tests and Quickstart documentation to show how to use these. Please see *Responses* for details.
- Improve documentation for `UserDatastore.create_user()` to make clear that hashed password should be passed in.
- Improve documentation for `UserDatastore` and `verify_and_update_password()` to make clear that caller must commit changes to DB if using a session based datastore.
- (#122) Clarify when to use `confirm_register_form` rather than `register_form`.
- Fix bug in 2FA that didn't commit DB after using *verify_and_update_password*.
- Fix bug(s) in UserDatastore where changes to user `active` flag weren't being added to DB.
- (#127) JSON response was failing due to LazyStrings in error response.
- (#117) Making a user inactive should stop all access immediately.
- (#134) Confirmation token can no longer be reused. Added *SECURITY_AUTO_LOGIN_AFTER_CONFIRM* option for applications that don't want the user to be automatically logged in after confirmation (defaults to True - existing behavior).
- (#159) The `/register` endpoint returned the Authentication Token even though confirmation was required. This was a huge security hole - it has been fixed.
- (#160) The 2FA totp_secret would be regenerated upon submission, making QRCode not work. (malware-watch)
- (#166) *default_render_json* uses `flask.make_response` and forces the Content-Type to JSON for generating the response (koekie)
- (#166) *SECURITY_MSG_UNAUTHENTICATED* added to the configuration.
- (#168) When using the @auth_required or @auth_token_required decorators, the token would be verified twice, and the DB would be queried twice for the user. Given how slow token verification is - this was a significant issue. That has been fixed.
- (#84) The `anonymous_user_required()` was not JSON friendly - always performing a redirect. Now, if the request 'wants' a JSON response - it will receive a 400 with an error message defined by *SECURITY_MSG_ANONYMOUS_USER_REQUIRED*.
- (#145) Improve 2FA templates to that they can be localized. (taavie)
- (#173) *SECURITY_UNAUTHORIZED_VIEW* didn't accept a url (just an endpoint). All other view configurations did. That has been fixed.

Possible compatibility issues

- (#164) In prior releases, the Authentication Token was returned as part of the JSON response to each successful call to */login*, */change*, or */reset/{token}* API call. This is not a great idea since for browser-based UIs that used JSON request/response, and used session based authentication - they would be sent this token - even though it was likely ignored. Since these tokens by default have no expiration time this exposed a needless security hole. The new default behavior is to ONLY return the Authentication Token from those APIs if the query param `include_auth_token` is added to the request. Prior behavior can be restored by setting the *SECURITY_BACKWARDS_COMPAT_AUTH_TOKEN* configuration variable.
- (#120) `RoleMixin` now has a method `get_permissions()` which is called as part each request to add Permissions to the authenticated user. It checks if the RoleModel has a property `permissions` and assumes it is a comma separated string of permissions. If your model already has such a property this will likely fail. You need to override `get_permissions()` and simply return an emtpy set.
- (#121) Changes the default (failure) behavior for views protected with @auth_required, @token_auth_required, or @http_auth_required. Before, a 401 was returned with some stock html. Now, Flask-Login.unauthorized() is called (the same as @login_required does) - which by default redirects to a login page/view. If you had provided your own *.Security.unauthorized_callback* there are no changes - that will still be called first. The old default behavior can be restored by setting *SECURITY_BACKWARDS_COMPAT_UNAUTHN* to True. Please see *Responses* for details.
- (#127) Fix for LazyStrings in json error response. The fix for this has Flask-Security registering its own JsonEncoder on its blueprint. If you registered your own JsonEncoder for your app - it will no longer be called when serializing responses to Flask-Security endpoints. You can register your JsonEncoder on Flask-Security's blueprint by sending it as *json_encoder_cls* as part of initialization. Be aware that your JsonEncoder needs to handle LazyStrings (see speaklater).
- (#84) Prior to this fix - anytime the decorator `anonymous_user_required()` failed, it caused a redirect to the post_login_view. Now, if the caller wanted a JSON response, it will return a 400.
- (#156) Faster Authentication Token introduced the following non-backwards compatible behavior change:
  - Since the old Authentication Token algorithm used the (hashed) user's password, those tokens would be invalidated whenever the user changed their password. This is not likely to be what most users expect. Since the new Authentication Token algorithm doesn't refer to the user's password, changing the user's password won't invalidate outstanding Authentication Tokens. The method `UserDatastore.set_uniquifier()` can be used by an administrator to change a user's `fs_uniquifier` - but nothing the user themselves can do to invalidate their Authentication Tokens. Setting the *SECURITY_BACKWARDS_COMPAT_AUTH_TOKEN_INVALIDATE* configuration variable will cause the user's `fs_uniquifier` to be changed when they change their password, thus restoring prior behavior.

New fast authentication token implementation

Current auth tokens are slow because they use the user's password (hashed) as a uniquifier (the user id isn't really enough since it might be reused). This requires checking the (hashed) password against what is in the token on EVERY request - however hashing is (on purpose) slow. So this can add almost a whole second to every request.

To solve this, a new attribute in the User model was added - `fs_uniquifier`. If this is present in your User model, then it will be used instead of the password for ensuring the token corresponds to the correct user. This is very fast. If that attribute is NOT present - then the behavior falls back to the existing (slow) method.

DB Migration

To use the new UserModel mixins or to add the column `user.fs_uniquifier` to speed up token authentication, a schema AND data migration needs to happen. If you are using Alembic the schema migration is easy - but you need to add `fs_uniquifier` values to all your existing data. You can add code like this to your migrations::update method:

```
# be sure to MODIFY this line to make nullable=True:
op.add_column('user', sa.Column('fs_uniquifier', sa.String(length=64), nullable=True))

# update existing rows with unique fs_uniquifier
import uuid
user_table = sa.Table('user', sa.MetaData(), sa.Column('id', sa.Integer, primary_
↪key=True),
                      sa.Column('fs_uniquifier', sa.String))
conn = op.get_bind()
for row in conn.execute(sa.select([user_table.c.id])):
    conn.execute(user_table.update().values(fs_uniquifier=uuid.uuid4().hex).where(user_
↪table.c.id == row['id']))

# finally - set nullable to false
op.alter_column('user', 'fs_uniquifier', nullable=False)

# for MySQL the previous line has to be replaced with...
# op.alter_column('user', 'fs_uniquifier', existing_type=sa.String(length=64),␣
↪nullable=False)
```

4.2.31 Version 3.2.0

Released June 26th 2019

- (#80) Support caching of authentication token (eregnier opr #839). This adds a new configuration variable *SECURITY_USE_VERIFY_PASSWORD_CACHE* which enables a cache (with configurable TTL) for authentication tokens. This is a big performance boost for those accessing Flask-Security via token as opposed to session.
- (#81) Support for JSON/Single-Page-Application. This completes support for non-form based access to Flask-Security. See PR for details. (jwag956)
- (#79 Add POST logout to enhance JSON usage (jwag956).
- (#73) Fix get_user for various DBs (jwag956). This is a more complete fix than in opr #633.
- (#78, #103) Add formal openapi API spec (jwag956).
- (#86, #94, #98, #101, #104) Add Two-factor authentication (opr #842) (baurt, jwag956).
- (#108) Fix form field label translations (jwag956)
- (#115) Fix form error message translations (upstream #801) (jwag956)
- (#87) Convert entire repo to Black (baurt)

4.2.32 Version 3.1.0

Released never

- (#53) Use Security.render_template in mails too (noirbizarre opr #487)
- (#56) Optimize DB accesses by using an SQL JOIN when retrieving a user. (nfvs opr #679)
- (#57) Add base template to security templates (grihabor opr #697)
- (#73) datastore: get user by numeric identity attribute (jirikuncar opr #633)
- (#58) bugfix: support application factory pattern (briancappello opr #703)
- (#60) Make SECURITY_PASSWORD_SINGLE_HASH a list of scheme ignoring double hash (noirbizarre opr #714)
- (#61) Allow custom login_manager to be passed in to Flask-Security (jaza opr #717)
- (#62) Docs for OAauth2-based custom login manager (jaza opr #727)
- (#63) core: make the User model check the password (mklassen opr #779)
- (#64) Customizable send_mail (abulte opr #730)
- (#68) core: fix default for UNAUTHORIZED_VIEW (jirijunkar opr #726)

These should all be backwards compatible.

Possible compatibility issues:

- #487 - prior to this, render_template() was overridable for views, but not emails. If anyone actually relied on this behavior, this has changed.
- #703 - get factory pattern working again. There was a very complex dance between Security() instantiation and init_app regarding kwargs. This has been rationalized (hopefully).
- #679 - SqlAlchemy SQL improvement. It is possible you will get the following error:

  ```
  Got exception during processing: <class 'sqlalchemy.exc.InvalidRequestError'> -
  'User.roles' does not support object population - eager loading cannot be applied.
  ```

  This is likely solvable by removing `lazy='dynamic'` from your Role definition.

Performance improvements:

- #679 - for sqlalchemy, for each request, there would be 2 DB accesses - now there is one.

Testing: For datastores operations, Sqlalchemy, peewee, pony were all tested against sqlite, postgres, and mysql real databases.

4.2.33 Version 3.0.2

Released April 30th 2019

- (opr #439) HTTP Auth respects SECURITY_USER_IDENTITY_ATTRIBUTES (pnpnpn)
- (opr #660) csrf_enabled` deprecation fix (abulte)
- (opr #671) Fix referrer loop in _get_unauthorized_view(). (nfvs)
- (opr #675) Fix AttributeError in _request_loader (sbagan)
- (opr #676) Fix timing attack on login form (cript0nauta)
- (opr #683) Close db connection after running tests (reambus)

- (opr #691) docs: add password salt to SQLAlchemy app example (KshitijKarthick)
- (opr #692) utils: fix incorrect email sender type (switowski)
- (opr #696) Fixed broken Click link (williamhatcher)
- (opr #722) Fix password recovery confirmation on deleted user (kesara)
- (opr #747) Update login_user.html (rickwest)
- (opr #748) i18n: configurable the dirname domain (escudero)
- (opr #835) adds relevant user to reset password form for validation purposes (fuhrysteve)

These are bug fixes and a couple very small additions. No change in behavior and no new functionality. 'opr#' is the original pull request from https://github.com/mattupstate/flask-security

4.2.34 Version 3.0.1

Released April 28th 2019

- Support 3.7 as part of CI
- Rebrand to this forked repo
- (#15) Build docs and translations as part of CI
- (#17) Move to msgcheck from pytest-translations
- (opr #669) Fix for Read the Docs (jirikuncar)
- (opr #710) Spanish translation (maukoquiroga)
- (opr #712) i18n: improvements of German translations (eseifert)
- (opr #713) i18n: add Portuguese (Brazilian) translation (dinorox)
- (opr #719) docs: fix anchor links and typos (kesara)
- (opr #751) i18n: fix missing space (abulte)
- (opr #762) docs: fixed proxy import (lsmith)
- (opr #767) Update customizing.rst (allanice001)
- (opr #776) i18n: add Portuguese (Portugal) translation (micael-grilo)
- (opr #791) Fix documentation for mattupstate#781 (fmerges)
- (opr #796) Chinese translations (Steinkuo)
- (opr #808) Clarify that a commit is needed after login_user (christophertull)
- (opr #823) Add Turkish translation (Admicos)
- (opr #831) Catalan translation (miceno)

These are all documentation and i18n changes - NO code changes. All except the last 3 were accepted and reviewed by the original Flask-Security team. Thanks as always to all the contributors.

4.2.35 Version 3.0.0

Released May 29th 2017

- Fixed a bug when user clicking confirmation link after confirmation and expiration causes confirmation email to resend. (see #556)
- Added support for I18N.
- Added options *SECURITY_EMAIL_PLAINTEXT* and *SECURITY_EMAIL_HTML* for sending respectively plaintext and HTML version of email.
- Fixed validation when missing login information.
- Fixed condition for token extraction from JSON body.
- Better support for universal bdist wheel.
- Added port of CLI using Click configurable using options *SECURITY_CLI_USERS_NAME* and *SECURITY_CLI_ROLES_NAME*.
- Added new configuration option *SECURITY_DATETIME_FACTORY* which can be used to force default timezone for newly created datetimes. (see mattupstate/flask-security#466)
- Better IP tracking if using Flask 0.12.
- Renamed deprecated Flask-WFT base form class.
- Added tests for custom forms configured using app config.
- Added validation and tests for next argument in logout endpoint. (see #499)
- Bumped minimal required versions of several packages.
- Extended test matric on Travis CI for minimal and released package versions.
- Added of .editorconfig and forced tests for code style.
- Fixed a security bug when validating a confirmation token, also checks if the email that the token was created with matches the user's current email.
- Replaced token loader with request loader.
- Changed trackable behavior of *login_user* when IP can not be detected from a request from 'untrackable' to *None* value.
- Use ProxyFix instead of inspecting X-Forwarded-For header.
- Fix identical problem with app as with datastore.
- Removed always-failing assertion.
- Fixed failure of init_app to set self.datastore.
- Changed to new style flask imports.
- Added proper error code when returning JSON response.
- Changed obsolete Required validator from WTForms to DataRequired. Bumped Flask-WTF to 0.13.
- Fixed missing *SECURITY_SUBDOMAIN* in config docs.
- Added cascade delete in PeeweeDatastore.
- Added notes to docs about *SECURITY_USER_IDENTITY_ATTRIBUTES*.
- Inspect value of *SECURITY_UNAUTHORIZED_VIEW*.
- Send password reset instructions if an attempt has expired.

- Added "Forgot password?" link to LoginForm description.
- Upgraded passlib, and removed bcrypt version restriction.
- Removed a duplicate line ('retype_password': 'Retype Password') in forms.py.
- Various documentation improvement.

4.2.36 Version 1.7.5

Released December 2nd 2015

- Added *SECURITY_TOKEN_MAX_AGE* configuration setting
- Fixed calls to *SQLAlchemyUserDatastore.get_user(None)* (this now returns *False* instead of raising a *TypeError*
- Fixed URL generation adding extra slashes in some cases (see GitHub #343)
- Fixed handling of trackable IP addresses when the *X-Forwarded-For* header contains multiple values
- Include WWW-Authenticate headers in *@auth_required* authentication checks
- Fixed error when *check_token* function is used with a json list
- Added support for custom *AnonymousUser* classes
- Restricted *forgot_password* endpoint to anonymous users
- Allowed unauthorized callback to be overridden
- Fixed issue where passwords cannot be reset if currently set to *None*
- Ensured that password reset tokens are invalidated after use
- Updated *is_authenticated* and *is_active* functions to support Flask-Login changes
- Various documentation improvements

4.2.37 Version 1.7.4

Released October 13th 2014

- Fixed a bug related to changing existing passwords from plaintext to hashed
- Fixed a bug in form validation that did not enforce case insensitivity
- Fixed a bug with validating redirects

4.2.38 Version 1.7.3

Released June 10th 2014

- Fixed a bug where redirection to *SECURITY_POST_LOGIN_VIEW* was not respected
- Fixed string encoding in various places to be friendly to unicode
- Now using *werkzeug.security.safe_str_cmp* to check tokens
- Removed user information from JSON output on */reset* responses
- Added Python 3.4 support

4.2.39 Version 1.7.2

Released May 6th 2014

- Updated IP tracking to check for *X-Forwarded-For* header
- Fixed a bug regarding the re-hashing of passwords with a new algorithm
- Fixed a bug regarding the *password_changed* signal.

4.2.40 Version 1.7.1

Released January 14th 2014

- Fixed a bug where passwords would fail to verify when specifying a password hash algorithm

4.2.41 Version 1.7.0

Released January 10th 2014

- Python 3.3 support!
- Dependency updates
- Fixed a bug when *SECURITY_LOGIN_WITHOUT_CONFIRMATION = True* did not allow users to log in
- Added *SECURITY_SEND_PASSWORD_RESET_NOTICE_EMAIL* configuration option to optionally send password reset notice emails
- Add documentation for *@security.send_mail_task*
- Move to *request.get_json* as *request.json* is now deprecated in Flask
- Fixed a bug when using AJAX to change a user's password
- Added documentation for select functions in the *flask_security.utils* module
- Fixed a bug in *flask_security.forms.NextFormMixin*
- Added *CHANGE_PASSWORD_TEMPLATE* configuration option to optionally specify a different change password template
- Added the ability to specify addtional fields on the user model to be used for identifying the user via the *USER_IDENTITY_ATTRIBUTES* configuration option
- An error is now shown if a user tries to change their password and the password is the same as before. The message can be customed with the *SECURITY_MSG_PASSWORD_IS_SAME* configuration option
- Fixed a bug in *MongoEngineUserDatastore* where user model would not be updated when using the *add_role_to_user* method
- Added *SECURITY_SEND_PASSWORD_CHANGE_EMAIL* configuration option to optionally disable password change email from being sent
- Fixed a bug in the *find_or_create_role* method of the PeeWee datastore
- Removed pypy tests
- Fixed some tests
- Include CHANGES and LICENSE in MANIFEST.in
- A bit of documentation cleanup

- A bit of code cleanup including removal of unnecessary utcnow call and simplification of get_max_age method

4.2.42 Version 1.6.9

Released August 20th 2013

- Fix bug in SQLAlchemy datastore's *get_user* function
- Fix bug in PeeWee datastore's *remove_role_from_user* function
- Fixed import error caused by new Flask-WTF release

4.2.43 Version 1.6.8

Released August 1st 2013

- Fixed bug with case sensitivity of email address during login
- Code cleanup regarding token_callback
- Ignore validation errors in find_user function for MongoEngineUserDatastore

4.2.44 Version 1.6.7

Released July 11th 2013

- Made password length form error message configurable
- Fixed email confirmation bug that prevented logged in users from confirming their email

4.2.45 Version 1.6.6

Released June 28th 2013

- Fixed dependency versions

4.2.46 Version 1.6.5

Released June 20th 2013

- Fixed bug in *flask.ext.security.confirmable.generate_confirmation_link*

4.2.47 Version 1.6.4

Released June 18th 2013

- Added *SECURITY_DEFAULT_REMEMBER_ME* configuration value to unify behavior between endpoints
- Fixed Flask-Login dependency problem
- Added optional *next* parameter to registration endpoint, similar to that of login

4.2.48 Version 1.6.3

Released May 8th 2013

- Fixed bug in regards to imports with latest version of MongoEngine

4.2.49 Version 1.6.2

Released April 4th 2013

- Fixed bug with http basic auth

4.2.50 Version 1.6.1

Released April 3rd 2013

- Fixed bug with signals

4.2.51 Version 1.6.0

Released March 13th 2013

- Added Flask-Pewee support
- Password hashing is now more flexible and can be changed to a different type at will
- Flask-Login messages are configurable
- AJAX requests must now send a CSRF token for security reasons
- Form messages are now configurable
- Forms can now be extended with more fields
- Added change password endpoint
- Added the user to the request context when successfully authenticated via http basic and token auth
- The Flask-Security blueprint subdomain is now configurable
- Redirects to other domains are now not allowed during requests that may redirect
- Template paths can be configured
- The welcome/register email can now optionally be sent to the user
- Passwords can now contain non-latin characters
- Fixed a bug when confirming an account but the account has been deleted

4.2.52 Version 1.5.4

Released January 6th 2013

- Fix bug in forms with *csrf_enabled* parameter not accounting attempts to login using JSON data

4.2.53 Version 1.5.3

Released December 23rd 2012

- Change dependency requirement

4.2.54 Version 1.5.2

Released December 11th 2012

- Fix a small bug in *flask_security.utils.login_user* method

4.2.55 Version 1.5.1

Released November 26th 2012

- Fixed bug with *next* form variable
- Added better documentation regarding Flask-Mail configuration
- Added ability to configure email subjects

4.2.56 Version 1.5.0

Released October 11th 2012

- Major release. Upgrading from previous versions will require a bit of work to accommodate API changes. See documentation for a list of new features and for help on how to upgrade.

4.2.57 Version 1.2.3

Released June 12th 2012

- Fixed a bug in the RoleMixin eq/ne functions

4.2.58 Version 1.2.2

Released April 27th 2012

- Fixed bug where *roles_required* and *roles_accepted* did not pass the next argument to the login view

4.2.59 Version 1.2.1

Released March 28th 2012

- Added optional user model mixin parameter for datastores
- Added CreateRoleCommand to available Flask-Script commands

4.2.60 Version 1.2.0

Released March 12th 2012

- Added configuration option *SECURITY_FLASH_MESSAGES* which can be set to a boolean value to specify if Flask-Security should flash messages or not.

4.2.61 Version 1.1.0

Initial release

Flask-Security was written by Matt Wright and various contributors.

Flask-Security-Too is an independently maintained repo:

4.3 Development Lead

- Chris Wagner <jwag956@github.com>

4.4 Maintainer

- Chris Wagner <jwag956@github.com>

4.5 Patches and Suggestions

Alexander Sukharev Alexey Poryadin Andrew J. Camenga Anthony Plunkett Artem Andreev Catherine Wise Chris Haines Christophe Simonis David Ignacio Eric Butler Eskil Heyn Olsen Iuri de Silvio Jay Goel Jiri Kuncar Joe Esposito Joe Hand Josh Purvis Kostyantyn Leschenko Luca Invernizzi Manuel Ebert Martin Maillard Paweł Krześniak Robert Clark Rodrigue Cloutier Rotem Yaari Srijan Choudhary Tristan Escalada Vadim Kotov Walt Askew John Paraskevopoulos Chris Wagner Eric Regnier Gal Stainfeld Ivan Piskunov Tyler Baur Glenn Lehman

INDEX