Contents

1 Contents 3
Flask-Security allows you to quickly add common security mechanisms to your Flask application. They include:

1. Session based authentication
2. Role 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. User registration (optional)
9. Login tracking (optional)
10. JSON/Ajax Support

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

1. Flask-Login
2. Flask-Mail
3. Flask-Principal
4. Flask-WTF
5. itsdangerous
6. passlib

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. Flask-MongoEngine
3. Flask-Peewee
4. PonyORM
1.1 Features

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

1.1.1 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 for remembering users when their session has expired.

1.1.2 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 all roles should be uniquely named. This feature is implemented using the Flask-Principal extension. If you’d like to implement more granular access control, you can refer to the Flask-Principal documentation on this topic.

1.1.3 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 an hashing algorithm in your production environment. You may also specify to use HMAC with a configured salt value in addition to the algorithm chosen. Bear in mind passlib does not assume which algorithm you will choose and may require additional libraries to be installed.
1.1.4 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.

1.1.5 Token Authentication

Token based authentication is enabled by retrieving the user auth token by performing an HTTP POST with the authentication details as JSON data against the authentication endpoint. A successful call to this endpoint will return the user’s ID and their authentication token. This token can be used in subsequent requests to protected resources. The auth token is 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 the user’s password. Thus if the user changes his or her password their existing authentication token will become invalid. A new token will need to be retrieved using the user’s new password.

1.1.6 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.

1.1.7 Password Reset/Recovery

Password reset and recovery is available for when a user forgets his or her password. Flask-Security sends an email to the user with a link to a view which they can reset their password. Once the password is reset they are automatically logged in and can use the new password from then on. Password reset links can be configured to expire after a specified amount of time.

1.1.8 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.

1.1.9 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
1.1.10 JSON/Ajax Support

Flask-Security supports JSON/Ajax requests where appropriate. Just remember that all endpoints require a CSRF token just like HTML views. More specifically JSON is supported for the following operations:

- Login requests
- Registration requests
- Change password requests
- Confirmation requests
- Forgot password requests
- Passwordless login requests

1.1.11 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.

1.2 Configuration

The following configuration values are used by Flask-Security:
## 1.2.1 Core

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SECURITY_BLUEPRINT_NAME</td>
<td>Specifies the name for the Flask-Security blueprint. Defaults to security.</td>
</tr>
<tr>
<td>SECURITY_CLI_USERS_NAME</td>
<td>Specifies the name for the command managing users. Disable by setting False.</td>
</tr>
<tr>
<td>SECURITY_CLI_ROLES_NAME</td>
<td>Specifies the name for the command managing roles. Disable by setting False.</td>
</tr>
<tr>
<td>SECURITY_URL_PREFIX</td>
<td>Specifies the URL prefix for the Flask-Security blueprint. Defaults to None.</td>
</tr>
<tr>
<td>SECURITY_SUBDOMAIN</td>
<td>Specifies the subdomain for the Flask-Security blueprint. Defaults to None.</td>
</tr>
<tr>
<td>SECURITY_FLASH_MESSAGES</td>
<td>Specifies whether or not to flash messages during security procedures.</td>
</tr>
<tr>
<td>SECURITY_I18N_DOMAIN</td>
<td>Specifies the directory containing the MO files used for translations.</td>
</tr>
<tr>
<td>SECURITY_I18N_DIRNAME</td>
<td>Specifies the directory containing the MO files used for translations.</td>
</tr>
<tr>
<td>SECURITY_PASSWORD_HASH</td>
<td>Specifies the password hash algorithm to use when hashing passwords.</td>
</tr>
<tr>
<td>SECURITY_PASSWORD_SALT</td>
<td>Specifies the HMAC salt. This is only used if the password hash type is set to something other than plain text. Defaults to None.</td>
</tr>
<tr>
<td>SECURITY_PASSWORD_SINGLE_HASH</td>
<td>Specifies that passwords should only be hashed once. By default, passwords are hashed twice, first with SECURITY_PASSWORD_SALT, and then with a random salt. May be useful for integrating with other applications. It can also be a set of scheme that should not be hashed twice. Default to a list of known schemes not working with double hashing (django_digest, plaintext). Defaults to False.</td>
</tr>
<tr>
<td>SECURITY_HASHING_SCHEMES</td>
<td>Specifies additional options to be passed to the hashing method.</td>
</tr>
<tr>
<td>SECURITY_DEPRECATED_HASHING_SCHEMES</td>
<td>Specifies the email address to send emails as. Defaults to value set to MAIL_DEFAULT_SENDER if Flask-Mail is used otherwise no-reply@localhost.</td>
</tr>
<tr>
<td>SECURITY_EMAIL_SENDER</td>
<td>Specifies the query string parameter to read when using token authentication.</td>
</tr>
<tr>
<td>SECURITY_TOKEN_AUTHENTICATION_KEY</td>
<td>Specifies the HTTP header to read when using token authentication.</td>
</tr>
<tr>
<td>SECURITY_TOKEN_MAX_AGE</td>
<td>Specifies the number of seconds before an authentication token expires.</td>
</tr>
<tr>
<td>SECURITY_DEFAULT_HTTP_AUTH_REALM</td>
<td>Specifies the default authentication realm when using basic HTTP auth.</td>
</tr>
</tbody>
</table>
### 1.2.2 URLs and Views

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SECURITY_LOGIN_URL</td>
<td>Specifies the login URL. Defaults to /login.</td>
</tr>
<tr>
<td>SECURITY_LOGOUT_URL</td>
<td>Specifies the logout URL. Defaults to /logout.</td>
</tr>
<tr>
<td>SECURITY_REGISTER_URL</td>
<td>Specifies the register URL. Defaults to /register.</td>
</tr>
<tr>
<td>SECURITY_RESET_URL</td>
<td>Specifies the password reset URL. Defaults to /reset.</td>
</tr>
<tr>
<td>SECURITY_CHANGE_URL</td>
<td>Specifies the password change URL. Defaults to /change.</td>
</tr>
<tr>
<td>SECURITY_CONFIRM_URL</td>
<td>Specifies the email confirmation URL. Defaults to /confirm.</td>
</tr>
<tr>
<td>SECURITY_POST_LOGIN_VIEW</td>
<td>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 / .</td>
</tr>
<tr>
<td>SECURITY_POST_LOGOUT_VIEW</td>
<td>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 / .</td>
</tr>
<tr>
<td>SECURITY_CONFIRM_ERROR_VIEW</td>
<td>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. Defaults to None.</td>
</tr>
<tr>
<td>SECURITY_POST_REGISTER_VIEW</td>
<td>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. Defaults to None.</td>
</tr>
<tr>
<td>SECURITY_POST_CONFIRM_VIEW</td>
<td>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. Defaults to None.</td>
</tr>
<tr>
<td>SECURITY_POST_RESET_VIEW</td>
<td>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 SECURITY_POST_LOGIN_VIEW. Defaults to None.</td>
</tr>
<tr>
<td>SECURITY_POST_CHANGE_VIEW</td>
<td>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. Defaults to None.</td>
</tr>
<tr>
<td>SECURITY_UNAUTHORIZED_VIEW</td>
<td>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. Defaults to None.</td>
</tr>
</tbody>
</table>
1.2.3 Template Paths

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SECURITY_FORGOT_PASSWORD_TEMPLATE</td>
<td>Specifies the path to the template for the forgot password page. Defaults to security/forgot_password.html.</td>
</tr>
<tr>
<td>SECURITY_LOGIN_USER_TEMPLATE</td>
<td>Specifies the path to the template for the user login page. Defaults to security/login_user.html.</td>
</tr>
<tr>
<td>SECURITY_REGISTER_USER_TEMPLATE</td>
<td>Specifies the path to the template for the user registration page. Defaults to security/register_user.html.</td>
</tr>
<tr>
<td>SECURITY_RESET_PASSWORD_TEMPLATE</td>
<td>Specifies the path to the template for the reset password page. Defaults to security/reset_password.html.</td>
</tr>
<tr>
<td>SECURITY_CHANGE_PASSWORD_TEMPLATE</td>
<td>Specifies the path to the template for the change password page. Defaults to security/change_password.html.</td>
</tr>
<tr>
<td>SECURITY_SEND_CONFIRMATION_TEMPLATE</td>
<td>Specifies the path to the template for the resend confirmation instructions page. Defaults to security/send_confirmation.html.</td>
</tr>
<tr>
<td>SECURITY_SEND_LOGIN_TEMPLATE</td>
<td>Specifies the path to the template for the send login instructions page for passwordless logins. Defaults to security/send_login.html.</td>
</tr>
</tbody>
</table>

1.2.4 Feature Flags

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SECURITY_CONFIRMABLE</td>
<td>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. The URL for this endpoint is specified by the SECURITY_CONFIRM_URL configuration option. Defaults to False.</td>
</tr>
<tr>
<td>SECURITY_REGISTERABLE</td>
<td>Specifies if Flask-Security should create a user registration endpoint. The URL for this endpoint is specified by the SECURITY_REGISTER_URL configuration option. Defaults to False.</td>
</tr>
<tr>
<td>SECURITY_RECOVERABLE</td>
<td>Specifies if Flask-Security should create a password reset/recover endpoint. The URL for this endpoint is specified by the SECURITY_RESET_URL configuration option. Defaults to False.</td>
</tr>
<tr>
<td>SECURITY_TRACKABLE</td>
<td>Specifies if Flask-Security should track basic user login statistics. If set to True, ensure your models have the required fields/attributes. Be sure to use ProxyFix if you are using a proxy. Defaults to False.</td>
</tr>
<tr>
<td>SECURITY_PASSWORDLESS</td>
<td>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 experimental and should be used with caution. Defaults to False.</td>
</tr>
<tr>
<td>SECURITY_CHANGEABLE</td>
<td>Specifies if Flask-Security should enable the change password endpoint. The URL for this endpoint is specified by the SECURITY_CHANGE_URL configuration option. Defaults to False.</td>
</tr>
</tbody>
</table>
1.2.5 Email

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SECURITY_EMAIL_SUBJECT_REGISTER</td>
<td>Sets the subject for the confirmation email. Defaults to Welcome.</td>
</tr>
<tr>
<td>SECURITY_EMAIL_SUBJECT_PASSWORDLESS</td>
<td>Sets the subject for the passwordless feature. Defaults to Login instructions.</td>
</tr>
<tr>
<td>SECURITY_EMAIL_SUBJECT_PASSWORD_NOTICE</td>
<td>Sets the subject for the password notice. Defaults to Your password has been reset.</td>
</tr>
<tr>
<td>SECURITY_EMAIL_SUBJECT_PASSWORD_RESET</td>
<td>Sets the subject for the password reset email. Defaults to Password reset instructions.</td>
</tr>
<tr>
<td>SECURITY_EMAIL_SUBJECT_PASSWORD_CHANGE_NOTICE</td>
<td>Sets the subject for the password change notice. Defaults to Your password has been changed.</td>
</tr>
<tr>
<td>SECURITY_EMAIL_SUBJECT_CONFIRM</td>
<td>Sets the subject for the email confirmation message. Defaults to Please confirm your email.</td>
</tr>
<tr>
<td>SECURITY_EMAIL_PLAINTEXT</td>
<td>Sends email as plaintext using *.txt template. Defaults to True.</td>
</tr>
<tr>
<td>SECURITY_EMAIL_HTML</td>
<td>Sends email as HTML using *.html template. Defaults to True.</td>
</tr>
</tbody>
</table>
## 1.2.6 Miscellaneous

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SECURITY_USER_IDENTITY_ATTRIBUTES</td>
<td>Specifies which attributes of the user object can be used for login.Defaults to ['email'].</td>
</tr>
<tr>
<td>SECURITY_SEND_REGISTER_EMAIL</td>
<td>Specifies whether registration email is sent. Defaults to True.</td>
</tr>
<tr>
<td>SECURITY_SEND_PASSWORD_CHANGE_EMAIL</td>
<td>Specifies whether password change email is sent. Defaults to True.</td>
</tr>
<tr>
<td>SECURITY_SEND_PASSWORD_RESET_EMAIL</td>
<td>Specifies whether password reset email is sent. Defaults to True.</td>
</tr>
<tr>
<td>SECURITY_SEND_PASSWORD_RESET_NOTICE_EMAIL</td>
<td>Specifies whether password reset notice email is sent. Defaults to True.</td>
</tr>
<tr>
<td>SECURITY_CONFIRM_EMAIL_WITHIN</td>
<td>Specifies the amount of time a user has before their confirmation link expires. Always pluralized the time unit for this value. Defaults to 5 days.</td>
</tr>
<tr>
<td>SECURITY_RESET_PASSWORD_WITHIN</td>
<td>Specifies the amount of time a user has before their password reset link expires. Always pluralized the time unit for this value. Defaults to 5 days.</td>
</tr>
<tr>
<td>SECURITY_LOGIN_WITHIN</td>
<td>Specifies the amount of time a user has before a login link expires. This is only used when the passwordless login feature is enabled. Always pluralized the time unit for this value. Defaults to 1 days.</td>
</tr>
<tr>
<td>SECURITY_LOGIN_WITHOUT_CONFIRMATION</td>
<td>Specifies if a user may login before confirming their email when the value of SECURITY_CONFIRMABLE is set to True. Defaults to False.</td>
</tr>
<tr>
<td>SECURITY_CONFIRM_SALT</td>
<td>Specifies the salt value when generating confirmation links/tokens. Defaults to confirm-salt.</td>
</tr>
<tr>
<td>SECURITY_RESET_SALT</td>
<td>Specifies the salt value when generating password reset links/tokens. Defaults to reset-salt.</td>
</tr>
<tr>
<td>SECURITY_LOGIN_SALT</td>
<td>Specifies the salt value when generating login links/tokens. Defaults to login-salt.</td>
</tr>
<tr>
<td>SECURITY_REMEMBER_SALT</td>
<td>Specifies the salt value when generating remember tokens. Remember tokens are used instead of user ID’s as it is more secure. Defaults to remember-salt.</td>
</tr>
<tr>
<td>SECURITY_DEFAULT_REMEMBER_ME</td>
<td>Specifies the default “remember me&quot; value used when logging in a user. Defaults to False.</td>
</tr>
<tr>
<td>SECURITY_DATETIME_FACTORY</td>
<td>Specifies the default datetime factory. Defaults to datetime.datetime.utcnow.</td>
</tr>
</tbody>
</table>

## 1.2.7 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_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_FORGOT_PASSWORD
• SECURITY_MSG_INVALID_CONFIRMATION_TOKEN
• SECURITY_MSG_INVALID_EMAIL_ADDRESS
• SECURITY_MSG_INVALID_LOGIN_TOKEN
• SECURITY_MSG_INVALID_PASSWORD
• SECURITY_MSG_INVALID_REDIRECT
• SECURITY_MSG_INVALID_RESET_PASSWORD_TOKEN
• SECURITY_MSG_LOGIN
• SECURITY_MSG_LOGIN_EMAIL_SENT
• SECURITY_MSG_LOGIN_EXPIRED
• SECURITY_MSG_PASSWORDLESS_LOGIN_SUCCESSFUL
• 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_NOT_SET
• SECURITY_MSG_PASSWORD_RESET
• SECURITY_MSG_PASSWORD_RESET_EXPIRED
• SECURITY_MSG_PASSWORD_RESET_REQUEST
• SECURITY_MSG_REFRESH
• SECURITY_MSG_RETYPE_PASSWORD_MISMATCH
• SECURITY_MSG_UNAUTHORIZED
• SECURITY_MSG_USER_DOES_NOT_EXIST

### 1.3 Quick Start

- **Basic SQLAlchemy Application**
- **Basic SQLAlchemy Application with session**
- **Basic MongoEngine Application**
- **Basic Peewee Application**
- **Mail Configuration**
Proxy Configuration

1.4 Basic SQLAlchemy Application

1.4.1 SQLAlchemy Install requirements

```bash
$ mkvirtualenv <your-app-name>
$ pip install flask-security flask-sqlalchemy
```

1.4.2 SQLAlchemy Application

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

```python
from flask import Flask, render_template
from flask_sqlalchemy import SQLAlchemy
from flask_security import Security, SQLAlchemyUserDatastore, UserMixin, RoleMixin, login_required

# Create app
app = Flask(__name__)
app.config['DEBUG'] = True
app.config['SECRET_KEY'] = 'super-secret'
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite://'

# 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)
    password = db.Column(db.String(255))
    active = db.Column(db.Boolean())
    confirmed_at = db.Column(db.DateTime())
    roles = db.relationship('Role', secondary=roles_users, 
                            backref=db.backref('users', lazy='dynamic'))

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

# Create a user to test with
@app.before_first_request
def create_user():
    # (continues on next page)
```
1.5 Basic SQLAlchemy Application with session

1.5.1 SQLAlchemy Install requirements

```
$ mkvirtualenv <your-app-name>
$ pip install flask-security sqlalchemy
```

Also, you can use the extension Flask-SQLAlchemy-Session documentation.

1.5.2 SQLAlchemy Application

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

We are gonna split the application at least in three files: app.py, database.py and models.py. You can also do the models a folder and spread your tables there.

- **app.py**

```python
from flask import Flask, render_template_string
from flask_security import Security, current_user, login_required, 
SQLAlchemySessionUserDatastore
from database import db_session, init_db
from models import User, Role

# Create app
app = Flask(__name__)
app.config['DEBUG'] = True
app.config['SECRET_KEY'] = 'super-secret'
# Bcrypt is set as default SECURITY_PASSWORD_HASH, which requires a salt
app.config['SECURITY_PASSWORD_SALT'] = 'super-secret-random-salt'

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

# Create a user to test with
@app.before_first_request
```

(continues on next page)
def create_user():
    init_db()
    user_datastore.create_user(email='matt@nobien.net', password='password')
    db_session.commit()

# Views
@app.route('/', @login_required
def home():
    return render_template_string('Hello {{email}} !', email=current_user.email)
if __name__ == '__main__':
    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', 
    convert_unicode=True)
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
from sqlalchemy import create_engine
from sqlalchemy.orm import relationship, backref
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))

class User(Base, UserMixin):
1.6 Basic MongoEngine Application

1.6.1 MongoEngine Install requirements

```bash
$ mkvirtualenv <your-app-name>
$ pip install flask-security flask-mongoengine
```

1.6.2 MongoEngine Application

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

```python
from flask import Flask, render_template
from flask_mongoengine import MongoEngine
from flask_security import Security, MongoEngineUserDatastore, UserMixin, RoleMixin, login_required

# Create app
app = Flask(__name__)
app.config['DEBUG'] = True
app.config['SECRET_KEY'] = 'super-secret'

# MongoDB Config
app.config['MONGODB_DB'] = 'mydatabase'
app.config['MONGODB_HOST'] = 'localhost'
app.config['MONGODB_PORT'] = 27017

# Create database connection object
db = MongoEngine(app)

class Role(db.Document, RoleMixin):
    name = db.StringField(max_length=80, unique=True)
    description = db.StringField(max_length=255)

class User(db.Document, UserMixin):
    email = db.StringField(max_length=255)
    password = db.StringField(max_length=255)
```

(continues on next page)
active = db.BooleanField(default=True)
confirmed_at = db.DateTimeField()
roles = db.ListField(db.ReferenceField(Role), default=[]) 

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

# Create a user to test with
@app.before_first_request
def create_user():
    user_datastore.create_user(email='matt@nobien.net', password='password')

# Views
@app.route('/', @login_required
def home():
    return render_template('index.html')

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

1.7 Basic Peewee Application

1.7.1 Peewee Install requirements

$ mkvirtualenv <your-app-name>
$ pip install flask-security flask-peewee

1.7.2 Peewee Application

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

```python
from flask import Flask, render_template
from flask_peewee.db import Database
from peewee import *
from flask_security import Security, PeeweeUserDatastore, 
    UserMixin, RoleMixin, login_required

# Create app
app = Flask(__name__)
app.config['DEBUG'] = True
app.config['SECRET_KEY'] = 'super-secret'
app.config['DATABASE'] = {
    'name': 'example.db',
    'engine': 'peewee.SqliteDatabase',
}

# Create database connection object
db = Database(app)
```
class Role(db.Model, RoleMixin):
    name = CharField(unique=True)
    description = TextField(null=True)

class User(db.Model, UserMixin):
    email = TextField()
    password = TextField()
    active = BooleanField(default=True)
    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)

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

# Create a user to test with
@app.before_first_request
def create_user():
    for Model in (Role, User, UserRoles):
        Model.drop_table(fail_silently=True)
        Model.create_table(fail_silently=True)
    user_datastore.create_user(email='matt@nobien.net', password='password')

# Views
@app.route('/', methods=['GET'])
@login_required
def home():
    return render_template('index.html')

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

1.8 Mail Configuration

Flask-Security integrates with Flask-Mail to handle all email communications between user and site, so it’s important to configure Flask-Mail with your email server details so Flask-Security can talk with Flask-Mail correctly.

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_mail import Mail

# After 'Create app'
app.config['MAIL_SERVER'] = 'smtp.example.com'
app.config['MAIL_PORT'] = 465
app.config['MAIL_USE_SSL'] = True
Flask-Security Documentation, Release 3.0.0

app.config['MAIL_USERNAME'] = 'username'
app.config['MAIL_PASSWORD'] = 'password'
mail = Mail(app)

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

1.9 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:

```python
# At top of file
from werkzeug.contrib.fixers import ProxyFix

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

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

1.10 Models

Flask-Security assumes you’ll be using libraries such as SQLAlchemy, MongoEngine, Peewee or PonyORM to define a data model that includes a User and Role 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. At the bare minimum your User and Role model should include the following fields:

User
- id
- email
- password
- active

Role
- id
- name
- description

1.10.1 Additional Functionality

Depending on the application’s configuration, additional fields may need to be added to your User model.

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

**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
• current_login_at
• last_login_ip
• current_login_ip
• login_count

**Custom User Payload**

If you want a custom payload after Register or Login an user, define the method `get_security_payload` in your User model. The method must return a serializable object:

```python
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):
        return {
            'id': self.id,
            'name': self.name,
            'email': self.email
        }
```

### 1.11 Customizing Views

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.

#### 1.11.1 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/register_user.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:

```python
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
- `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

### 1.11.2 Forms

All forms can be overridden. For each form used, you can specify a replacement class. This allows you to add extra fields to the register form or override validators:
from flask_security.forms import RegisterForm
from wtforms import StringField, validators

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`, each field is passed to the user model (as kwargs) when a user is created. In the above case, the `first_name` and `last_name` fields are passed directly to the model, so the model should look like:

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))

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

- login_form: Login 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
- passwordless_login_form: Passwordless login form

1.11.3 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/change_notice.txt
- security/email/change_notice.html
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 for any links that are required in the email. If you require more values in the templates, you can specify an email context processor with the `mail_context_processor` decorator. For example:

```python
security = Security(app, user_datastore)

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

### 1.11.4 Emails with Celery

Sometimes it makes sense to send emails via a task queue, such as Celery. To delay the sending of emails, you can use the `@security.send_mail_task` decorator like so:

```python
# Setup the task
@celery.task
def send_security_email(msg):
    # Use the Flask-Mail extension instance to send the incoming `msg` parameter
    # which is an instance of `flask_mail.Message`
    mail.send(msg)

@security.send_mail_task
def delay_security_email(msg):
    send_security_email.delay(msg)
```

If factory method is going to be used for initialization, use `_SecurityState` object returned by `init_app` method to initialize Celery tasks instead of using `security.send_mail_task` directly like so:

```python
from flask import Flask
from flask_mail import Mail
from flask_security import Security, SQLAlchemyUserDatastore
from celery import Celery

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

def create_app(config):
    """Initialize Flask instance."""
    app = Flask(__name__)
    app.config.from_object(config)

    # Initialize Celery tasks
    celery.TaskBase.set_app(app)
    celery.TaskBase.set_logger(app.logger)
    celery.TaskBase.set_current_app(app)
    celery.TaskBase.set_current_task(app)
```

(continues on next page)
@celery.task
def send_flask_mail(msg):
    mail.send(msg)

mail.init_app(app)
datastore = SQLAlchemyUserDatastore(db, User, Role)
security_ctx = security.init_app(app, datastore)

# Flexible way for defining custom mail sending task.
@security_ctx.send_mail_task
def delay_flask_security_mail(msg):
    send_flask_mail.delay(msg)

# A shortcut.
security_ctx.send_mail_task(send_flask_mail.delay)

return app

Note that `flask_mail.Message` may not be serialized as an argument passed to Celery. The practical way with custom serialization may look like so:

```python
@celery.task
def send_flask_mail(**kwargs):
    mail.send(Message(**kwargs))

@security_ctx.send_mail_task
def delay_flask_security_mail(msg):
    send_flask_mail.delay(subject=msg.subject, sender=msg.sender,
                           recipients=msg.recipients, body=msg.body,
                           html=msg.html)
```

### 1.11.5 Custom send_mail method

It's also possible to completely override the `security.send_mail` method to implement your own logic. For example, you might want to use an alternative email library like *Flask-Emails*:

```python
from flask import Flask
from flask_security import Security, SQLAlchemyUserDatastore
from flask_emails import Message

security = Security()
def create_app(config):
    """Initialize Flask instance."""
    app = Flask(__name__) app.config.from_object(config)

    def custom_send_mail(subject, recipient, template, **context):
        ctx = ('security/email', template)
        message = Message(subject=subject, html=_security.render_template('%s/%s.html' % ctx, **context))
        message.send(mail_to=[recipient])

    datastore = SQLAlchemyUserDatastore(db, User, Role)
    security_ctx.send_mail = custom_send_mail

    return app
```

1.11. Customizing Views 23
Note: The above `security.send_mail_task` override will be useless if you override the entire `send_mail` method.

1.11.6 Authorization with OAuth2

Flask-Security can be set up to co-operate with Flask-OAuthlib, by implementing a custom request loader that authorizes a user based either on a Bearer token in the HTTP Authorization header, or on the Flask-Security standard authorization logic:

```python
from flask_oauthlib.provider import OAuth2Provider
from flask_security import AnonymousUser
from flask_security.core import (_user_loader as _flask_security_user_loader,
                                 _request_loader as _flask_security_request_loader)
from flask_security.utils import config_value as security_config_value

oauth = OAuth2Provider(app)

def _request_loader(request):
    """
    Load user from OAuth2 Authentication header or using Flask-Security's request loader.
    """
    user = None

    if hasattr(request, 'oauth'):
        user = request.oauth.user
    else:
        try:
            is_valid, oauth_request = oauth.verify_request(scopes=[])  # Need this try stmt in case oauthlib sometimes throws:
            if is_valid:
                user = oauth_request.user
        except AttributeError:
            pass

    if not user:
        user = _flask_security_request_loader(request)

    return user

def _get_login_manager(app, anonymous_user):
    """Prepare a login manager for Flask-Security to use."""
    login_manager = LoginManager()

    login_manager.anonymous_user = anonymous_user or AnonymousUser
    login_manager.login_view = '{0}.login'.format(security_config_value('BLUEPRINT_NAME', app=app))
    login_manager.user_loader(_flask_security_user_loader)
    login_manager.request_loader(_request_loader)

    if security_config_value('FLASH_MESSAGES', app=app):
        (login_manager.login_message,
```

(continues on next page)
login_manager.login_message_category = (  
    security_config_value('MSG_LOGIN', app=app) 
)(login_manager.needs_refresh_message,  
login_manager.needs_refresh_message_category) = (  
    security_config_value('MSG_REFRESH', app=app) 
)  
else:  
    login_manager.login_message = None  
    login_manager.needs_refresh_message = None  

login_manager.init_app(app)  
return login_manager

security = Security(  
    app, user_datastore,  
    login_manager=_get_login_manager(app, anonymous_user=None)
)

1.12 API

1.12.1 Core

class flask_security.core.Security(app=None, datastore=None, register_blueprint=True, **kwargs)

The Security class initializes the Flask-Security extension.

Parameters

- **app** – The application.
- **datastore** – An instance of a user datastore.
- **register_blueprint** – to register the Security blueprint or not.
- **login_form** – set form for the login view
- **register_form** – set form for the register view
- **confirm_register_form** – set form for the confirm register view
- **forgot_password_form** – set form for the forgot password view
- **reset_password_form** – set form for the reset password view
- **change_password_form** – set form for the change password view
- **send_confirmation_form** – set form for the send confirmation view
- **passwordless_login_form** – set form for the passwordless login view
- **anonymous_user** – class to use for anonymous user

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

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

Parameters

- **app** – The application.
- **datastore** – An instance of a user datastore.
- **register_blueprint** – to register the Security blueprint or not.
flask_security.core.current_user
A proxy for the current user.

1.12.2 Protecting Views

flask_security.decorators.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:

```python
@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:

```python
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.decorators.roles_required(*roles)
Decorator which specifies that a user must have all the specified roles. Example:

```python
@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**

- **args** – The required roles.

flask_security.decorators.roles_accepted(*roles)
Decorator which specifies that a user must have at least one of the specified roles. Example:

```python
@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**

- **args** – The possible roles.

flask_security.decorators.http_auth_required(realm)
Decorator that protects endpoints using Basic HTTP authentication.

**Parameters**

- **realm** – optional realm name
flask_security.decorators.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

1.12.3 User Object Helpers

class flask_security.core.UserMixin
Mixin for User model definitions

  get_auth_token()
  Returns the user’s authentication token.

  get_security_payload()
  Serialize user object as response payload.

  has_role(role)
  Returns True if the user identifies with the specified role.

    Parameters role – A role name or Role instance

  is_active
  Returns True if the user is active.

  verify_and_update_password(password)
  Verify and update user password using configured hash.

class flask_security.core.RoleMixin
Mixin for Role model definitions

class flask_security.core.AnonymousUser
AnonymousUser definition

  has_role(*args)
  Returns False

1.12.4 Datastores

class flask_security.datastore.Datastore(user_model, role_model)
Abstracted user datastore.

  Parameters

    • user_model – A user model class definition

    • role_model – A role model class definition

  activate_user(user)
  Activates a specified user. Returns True if a change was made.

    Parameters user – The user to activate

  add_role_to_user(user, role)
  Adds a role to a user.

    Parameters

    • user – The user to manipulate

    • role – The role to add to the user
create_role(**kwargs)
Creates and returns a new role from the given parameters.

create_user(**kwargs)
Creates and returns a new user from the given parameters.

deactivate_user(user)
Deactivates a specified user. Returns True if a change was made.

Parameters
user – The user to deactivate

delete_user(user)
Deletes the specified user.

Parameters
user – The user to delete

find_or_create_role(name, **kwargs)
Returns a role matching the given name or creates it with any additionally provided parameters.

find_role(*args, **kwargs)
Returns a role matching the provided name.

find_user(*args, **kwargs)
Returns a user matching the provided parameters.

get_user(id_or_email)
Returns a user matching the specified ID or email address.

remove_role_from_user(user, role)
Removes a role from a user.

Parameters
• user – The user to manipulate
• role – The role to remove from the user

toggle_active(user)
Toggles a user’s active status. Always returns True.

activate_user(user)
Activates a specified user. Returns True if a change was made.

Parameters
user – The user to activate

add_role_to_user(user, role)
Adds a role to a user.

Parameters
• user – The user to manipulate
• role – The role to add to the user

create_role(**kwargs)
Creates and returns a new role from the given parameters.

create_user(**kwargs)
Creates and returns a new user from the given parameters.
deactivate_user (user)
Deactivates a specified user. Returns True if a change was made.

Parameters
• user – The user to deactivate

delete_user (user)
Deletes the specified user.

Parameters
• user – The user to delete

find_or_create_role (name, **kwargs)
Returns a role matching the given name or creates it with any additionally provided parameters.

find_role (role)
Returns a role matching the provided name.

find_user (**kwargs)
Returns a user matching the provided parameters.

get_user (identifier)
Returns a user matching the specified ID or email address.

remove_role_from_user (user, role)
Removes a role from a user.

Parameters
• user – The user to manipulate
• role – The role to remove from the user

toggle_active (user)
Toggles a user’s active status. Always returns True.

class flask_security.datastore.MongoEngineUserDatastore (db, user_model, role_model)
A MongoEngine datastore implementation for Flask-Security that assumes the use of the Flask-MongoEngine extension.

activate_user (user)
Activates a specified user. Returns True if a change was made.

Parameters
• user – The user to activate

add_role_to_user (user, role)
 Adds a role to a user.

Parameters
• user – The user to manipulate
• role – The role to add to the user

create_role (**kwargs)
Creates and returns a new role from the given parameters.

create_user (**kwargs)
Creates and returns a new user from the given parameters.

deactivate_user (user)
Deactivates a specified user. Returns True if a change was made.

Parameters
• user – The user to deactivate

delete_user (user)
Deletes the specified user.
Parameters `user` – The user to delete

`find_or_create_role(name, **kwargs)`
Returns a role matching the given name or creates it with any additionally provided parameters.

`find_role(role)`
Returns a role matching the provided name.

`find_user(**kwargs)`
Returns a user matching the provided parameters.

`get_user(identifier)`
Returns a user matching the specified ID or email address.

`remove_role_from_user(user, role)`
Removes a role from a user.

Parameters
- `user` – The user to manipulate
- `role` – The role to remove from the user

`toggle_active(user)`
Toggles a user’s active status. Always returns True.

`class flask_security.datastore.PeeweeUserDatastore(db, user_model, role_model, role_link)`
A PeeweeD datastore implementation for Flask-Security that assumes the use of the Flask-Peewee extension.

Parameters
- `user_model` – A user model class definition
- `role_model` – A role model class definition
- `role_link` – A model implementing the many-to-many user-role relation

`activate_user(user)`
Activates a specified user. Returns True if a change was made.

Parameters `user` – The user to activate

`add_role_to_user(user, role)`
Adds a role to a user.

Parameters
- `user` – The user to manipulate
- `role` – The role to add to the user

`create_role(**kwargs)`
Creates and returns a new role from the given parameters.

`create_user(**kwargs)`
Creates and returns a new user from the given parameters.

`deactivate_user(user)`
Deactivates a specified user. Returns True if a change was made.

Parameters `user` – The user to deactivate

`delete_user(user)`
Deletes the specified user.

Parameters `user` – The user to delete
find_or_create_role (name, **kwargs)
Returns a role matching the given name or creates it with any additionally provided parameters.

find_role (role)
Returns a role matching the provided name.

find_user (**kwargs)
Returns a user matching the provided parameters.

get_user (identifier)
Returns a user matching the specified ID or email address.

remove_role_from_user (user, role)
Removes a role from a user.

Parameters
• user – The user to manipulate
• role – The role to remove from the user

toggle_active (user)
Toggles a user’s active status. Always returns True.

class flask_security.datastore.PonyUserDatastore (db, user_model, role_model)
A Pony ORM datastore implementation for Flask-Security.

Code primarily from https://github.com/ET-CS but taken over after being abandoned.

activate_user (user)
Activates a specified user. Returns True if a change was made.

Parameters user – The user to activate

add_role_to_user (**kwargs)
Adds a role to a user.

Parameters
• user – The user to manipulate
• role – The role to add to the user

create_role (**kwargs)
Creates and returns a new role from the given parameters.

create_user (**kwargs)
Creates and returns a new user from the given parameters.

deactivate_user (user)
Deactivates a specified user. Returns True if a change was made.

Parameters user – The user to deactivate

delete_user (user)
Deletes the specified user.

Parameters user – The user to delete

find_or_create_role (name, **kwargs)
Returns a role matching the given name or creates it with any additionally provided parameters.

find_role (**kwargs)
Returns a role matching the provided name.
find_user(**kwargs)
Returns a user matching the provided parameters.

get_user(**kwargs)
Returns a user matching the specified ID or email address.

remove_role_from_user(user, role)
Removes a role from a user.

Parameters

- user – The user to manipulate
- role – The role to remove from the user

toggle_active(user)
Toggles a user’s active status. Always returns True.

1.12.5 Utils

flask_security.utils.login_user(user, remember=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 – The user to login
- remember – Flag specifying if the remember cookie should be set. Defaults to False

flask_security.utils.logout_user()
Logs out the current.

This will also clean up the remember me cookie if it exists.

flask_security.utils.get_hmac(password)
Returns a Base64 encoded HMAC+SHA512 of the password signed with the salt specified by SECURITY_PASSWORD_SALT.

Parameters

- password – The password to sign

flask_security.utils.verify_password(password, password_hash)
Returns True if the password matches the supplied hash.

Parameters

- password – A plaintext password to verify
- password_hash – The expected hash value of the password (usually from your database)

flask_security.utils.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.

Parameters

- password – A plaintext password to verify
- user – The user to verify against
flask_security.utils.encrypt_password(password)
Encrypt the specified plaintext password.
It uses the configured encryption options.
Deprecated since version 2.0.2: Use hash_password() instead.

Parameters
password – The plaintext password to encrypt

flask_security.utils.hash_password(password)
Hash the specified plaintext password.
It uses the configured hashing options.
New in version 2.0.2.

Parameters
password – The plaintext password to hash

flask_security.utils.url_for_security(endpoint, **values)
Return a URL for the security blueprint

Parameters

• endpoint – the endpoint of the URL (name of the function)
• values – 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.

flask_security.utils.get_within_delta(key, app=None)
Get a timedelta object from the application configuration following the internal convention of:

<Amount of Units> <Type of Units>

Examples of valid config values:

5 days
10 minutes

Parameters

• key – The config value key without the ‘SECURITY_’ prefix
• app – Optional application to inspect. Defaults to Flask’s current_app

flask_security.utils.send_mail(subject, recipient, template, **context)
Send an email via the Flask-Mail extension.

Parameters

• subject – Email subject
• recipient – Email recipient
• template – The name of the email template
• context – The context to render the template with

flask_security.utils.get_token_status(token, serializer, max_age=None, return_data=False)
Get the status of a token.
Parameters

- **token** – The token to check
- **serializer** – The name of the serializer. Can be one of the following: confirm, login, reset
- **max_age** – The name of the max age config option. Can be one of the following: CONFIRM_EMAIL, LOGIN, RESET_PASSWORD

### 1.12.6 Signals

See the Flask documentation on signals for information on how to use these signals in your code.

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_registered**
  Sent when a user registers on the site. In addition to the app (which is the sender), it is passed `user` and `confirm_token` arguments.

- **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` argument.

- **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` and `token` arguments.

### 1.13 Flask-Security Changelog

Here you can see the full list of changes between each Flask-Security release.

#### 1.13.1 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 matrix 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 obsolette 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.
1.13.2 Version 1.7.5

Released December 2nd 2015

- Added `SECURITY_TOKEN_MAX_AGE` configuration setting
- Fixed calls to `SQLAlchemyUserDatstore.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

1.13.3 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 insensivitiy
- Fixed a bug with validating redirects

1.13.4 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

1.13.5 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.
1.13.6 Version 1.7.1

Released January 14th 2014

• Fixed a bug where passwords would fail to verify when specifying a password hash algorithm

1.13.7 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 additional 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 customized 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

1.13.8 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

1.13.9 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

1.13.10 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

1.13.11 Version 1.6.6

Released June 28th 2013
• Fixed dependency versions

1.13.12 Version 1.6.5

Released June 20th 2013
• Fixed bug in flask.ext.security.confirmable.generate_confirmation_link

1.13.13 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

1.13.14 Version 1.6.3

Released May 8th 2013
• Fixed bug in regards to imports with latest version of MongoEngine

1.13.15 Version 1.6.2

Released April 4th 2013
• Fixed bug with http basic auth
1.13.16 Version 1.6.1
Released April 3rd 2013
  • Fixed bug with signals

1.13.17 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

1.13.18 Version 1.5.4
Released January 6th 2013
  • Fix bug in forms with csrf_enabled parameter not accounting attempts to login using JSON data

1.13.19 Version 1.5.3
Released December 23rd 2012
  • Change dependency requirement

1.13.20 Version 1.5.2
Released December 11th 2012
  • Fix a small bug in flask_security.utils.login_user method
1.13.21 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

1.13.22 Version 1.5.0

Released October 11th 2012

• Major release. Upgrading from previous versions will require a bit of work to accomodate API changes. See documentation for a list of new features and for help on how to upgrade.

1.13.23 Version 1.2.3

Released June 12th 2012

• Fixed a bug in the RoleMixin `eq/ne` functions

1.13.24 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

1.13.25 Version 1.2.1

Released March 28th 2012

• Added optional user model mixin parameter for datastores
• Added CreateRoleCommand to available Flask-Script commands

1.13.26 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.

1.13.27 Version 1.1.0

Initial release

Flask-Security is written and maintained by Matt Wright and various contributors:
1.14 Development Lead

- Matt Wright <matt+github@nobien.net>

1.15 Patches and Suggestions

activate_user() (flask_security.datastore.MongoEngineUserDatastore method), 29
activate_user() (flask_security.datastore.PeeweeUserDatastore method), 30
activate_user() (flask_security.datastore.PonyUserDatastore method), 31
activate_user() (flask_security.datastore.SQLAlchemyUserDatastore method), 28
activate_user() (flask_security.datastore.UserDatastore method), 27
add_role_to_user() (flask_security.datastore.MongoEngineUserDatastore method), 29
add_role_to_user() (flask_security.datastore.PeeweeUserDatastore method), 30
add_role_to_user() (flask_security.datastore.PonyUserDatastore method), 31
add_role_to_user() (flask_security.datastore.SQLAlchemyUserDatastore method), 28
add_role_to_user() (flask_security.datastore.UserDatastore method), 27
AnonymousUser (class in flask_security.core), 27
auth_token_required() (in module flask_security.decorators), 26
confirm_instructions_sent (built-in variable), 34
create_role() (flask_security.datastore.MongoEngineUserDatastore method), 29
create_role() (flask_security.datastore.PeeweeUserDatastore method), 30
create_role() (flask_security.datastore.PonyUserDatastore method), 31
create_role() (flask_security.datastore.SQLAlchemyUserDatastore method), 28
create_role() (flask_security.datastore.UserDatastore method), 27
create_user() (flask_security.datastore.MongoEngineUserDatastore method), 29
create_user() (flask_security.datastore.PeeweeUserDatastore method), 30
create_user() (flask_security.datastore.PonyUserDatastore method), 31
create_user() (flask_security.datastore.SQLAlchemyUserDatastore method), 28
create_user() (flask_security.datastore.UserDatastore method), 27
deactivate_user() (flask_security.datastore.MongoEngineUserDatastore method), 29
deactivate_user() (flask_security.datastore.PeeweeUserDatastore method), 30
deactivate_user() (flask_security.datastore.PonyUserDatastore method), 31
deactivate_user() (flask_security.datastore.SQLAlchemyUserDatastore method), 28
deactivate_user() (flask_security.datastore.UserDatastore method), 28
delete_user() (flask_security.datastore.MongoEngineUserDatastore method), 29
delete_user() (flask_security.datastore.PeeweeUserDatastore method), 30
delete_user() (flask_security.datastore.PonyUserDatastore method), 31
delete_user() (flask_security.datastore.SQLAlchemyUserDatastore method), 29
delete_user() (flask_security.datastore.UserDatastore method), 28
decrypt_password() (in module flask_security.utils), 32
find_or_create_role() (flask_security.datastore.MongoEngineUserDatastore method), 30
find_or_create_role() (flask_security.datastore.PeeweeUserDatastore method), 30
find_or_create_user() (flask_security.datastore.MongoEngineUserDatastore method), 30
find_or_create_user() (flask_security.datastore.PeeweeUserDatastore method), 30
toggle_active() (flask_security.datastore.PonyUserDatastore method), 32

toggle_active() (flask_security.datastore.SQLAlchemyUserDatastore method), 29

toggle_active() (flask_security.datastore.UserDatastore method), 28

U

url_for_security() (in module flask_security.utils), 33

user_confirmed (built-in variable), 34

user_registered (built-in variable), 34

UserDatastore (class in flask_security.datastore), 27

UserMixin (class in flask_security.core), 27

V

verify_and_update_password()
   (flask_security.core.UserMixin method), 27

verify_and_update_password() (in module flask_security.utils), 32

verify_password() (in module flask_security.utils), 32