Flask-Security Documentation

Release 1.7.4

Matt Wright

May 22, 2017

Contents

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 encryption
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-Script
5. Flask-WTF
6. itsdangerous
7. 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

CHAPTER 1

Contents

Features

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

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.

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.

Password Encryption

Password encryption is enabled with passlib. Passwords are stored in plain text by default but you can easily configure the encryption algorithm. You should **always use an encryption 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.

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.

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.

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 an 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.

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.

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.

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

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

Configuration

The following configuration values are used by Flask-Security:

Core

`SECURITY_BLUEPRINT_NAME`	Specifies the name for the Flask-Security blueprint. Defaults to `security`.
`SECURITY_URL_PREFIX`	Specifies the URL prefix for the Flask-Security blueprint. Defaults to `None`.
`SECURITY_FLASH_MESSAGES`	Specifies whether or not to flash messages during security procedures. Defaults to `True`.
`SECURITY_PASSWORD_HASH`	Specifies the password hash algorithm to use when encrypting and decrypting passwords. Recommended values for production systems are `bcrypt`, `sha512_crypt`, or `pbkdf2_sha512`. Defaults to `plaintext`.
`SECURITY_PASSWORD_SALT`	Specifies the HMAC salt. This is only used if the password hash type is set to something other than plain text. Defaults to `None`.
`SECURITY_EMAIL_SENDER`	Specifies the email address to send emails as. Defaults to `no-reply@localhost`.
`SECURITY_TOKEN_AUTHENTICATION_KEY`	Specifies the query sting parameter to read when using token authentication. Defaults to `auth_token`.
`SECURITY_TOKEN_AUTHENTICATION_HEADER`	Specifies the HTTP header to read when using token authentication. Defaults to `Authentication-Token`.
`SECURITY_DEFAULT_HTTP_AUTH_REALM`	Specifies the default authentication realm when using basic HTTP auth. Defaults to `Login Required`

URLs and Views

`SECURITY_LOGIN_URL`	Specifies the login URL. Defaults to `/login`.
`SECURITY_LOGOUT_URL`	Specifies the logout URL. Defaults to `/logout`.
`SECURITY_REGISTER_URL`	Specifies the register URL. Defaults to `/register`.
`SECURITY_RESET_URL`	Specifies the password reset URL. Defaults to `/reset`.
`SECURITY_CHANGE_URL`	Specifies the password change URL. Defaults to `/change`.
`SECURITY_CONFIRM_URL`	Specifies the email confirmation URL. Defaults to `/confirm`.
`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 `/`.
`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 `/`.
`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. Defaults to `None`.
`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`. Defaults to `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`. Defaults to `None`.
`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 `SECURITY_POST_LOGIN_VIEW`. Defaults to `None`.
`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`. Defaults to `None`.
`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. Defaults to `None`.

Template Paths

`SECURITY_FORGOT_PASSWORD_TEMPLATE`	Specifies the path to the template for the forgot password page. Defaults to `security/forgot_password.html`.
`SECURITY_LOGIN_USER_TEMPLATE`	Specifies the path to the template for the user login page. Defaults to `security/login_user.html`.
`SECURITY_REGISTER_USER_TEMPLATE`	Specifies the path to the template for the user registration page. Defaults to `security/register_user.html`.
`SECURITY_RESET_PASSWORD_TEMPLATE`	Specifies the path to the template for the reset password page. Defaults to `security/reset_password.html`.
`SECURITY_CHANGE_PASSWORD_TEMPLATE`	Specifies the path to the template for the change password page. Defaults to `security/change_password.html`.
`SECURITY_SEND_CONFIRMATION_TEMPLATE`	Specifies the path to the template for the resend confirmation instructions page. Defaults to `security/send_confirmation.html`.
`SECURITY_SEND_LOGIN_TEMPLATE`	Specifies the path to the template for the send login instructions page for passwordless logins. Defaults to `security/send_login.html`.

Feature Flags

`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. The URL for this endpoint is specified by the `SECURITY_CONFIRM_URL` configuration option. Defaults to `False`.
`SECURITY_REGISTERABLE`	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`.
`SECURITY_RECOVERABLE`	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`.
`SECURITY_TRACKABLE`	Specifies if Flask-Security should track basic user login statistics. If set to `True`, ensure your models have the required fields/attribues. Defaults to `False`
`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 experimental and should be used with caution. Defaults to `False`.
`SECURITY_CHANGEABLE`	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`.

Email

SECURITY_EMAIL_SUBJECT_REGISTER	Sets the subject for the confirmation email. Defaults to `Welcome`
SECURITY_EMAIL_SUBJECT_PASSWORDLESS	Sets the subject for the passwordless feature. Defaults to `Login instructions`
SECURITY_EMAIL_SUBJECT_PASSWORD_NOTICE	Sets subject for the password notice. Defaults to `Your password has been reset`
SECURITY_EMAIL_SUBJECT_PASSWORD_RESET	Sets the subject for the password reset email. Defaults to `Password reset instructions`
SECURITY_EMAIL_SUBJECT_PASSWORD_CHANGE_NOTICE	Sets the subject for the password change notice. Defaults to `Your password has been changed`
SECURITY_EMAIL_SUBJECT_CONFIRM	Sets the subject for the email confirmation message. Defaults to `Please confirm your email`

Miscellaneous

SECURITY_SEND_REGISTER_EMAIL	Specifies whether registration email is sent. Defaults to `True`.
SECURITY_SEND_PASSWORD_CHANGE_EMAIL	Specifies whether password change email is sent. Defaults to `True`.
SECURITY_SEND_PASSWORD_RESET_NOTICE_EMAIL	Specifies whether password reset notice email is sent. Defaults to `True`.
SECURITY_CONFIRM_EMAIL_WITHIN	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`.
SECURITY_RESET_PASSWORD_WITHIN	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`.
SECURITY_LOGIN_WITHIN	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`.
SECURITY_LOGIN_WITHOUT_CONFIRMATION	Specifies if a user may login before confirming their email when the value of `SECURITY_CONFIRMABLE` is set to `True`. Defaults to `False`.
SECURITY_CONFIRM_SALT	Specifies the salt value when generating confirmation links/tokens. Defaults to `confirm-salt`.
SECURITY_RESET_SALT	Specifies the salt value when generating password reset links/tokens. Defaults to `reset-salt`.
SECURITY_LOGIN_SALT	Specifies the salt value when generating login links/tokens. Defaults to `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. Defaults to `remember-salt`.
SECURITY_DEFAULT_REMEMBER_ME	Specifies the default "remember me" value used when logging in a user. Defaults to `False`.

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_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`

Quick Start

Basic SQLAlchemy Application

SQLAlchemy Install requirements

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

SQLAlchemy Application

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

```
from flask import Flask, render_template
from flask.ext.sqlalchemy import SQLAlchemy
from flask.ext.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():
    db.create_all()
    user_datastore.create_user(email='matt@nobien.net', password='password')
    db.session.commit()

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

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

Basic MongoEngine Application

MongoEngine Install requirements

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

MongoEngine Application

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

```
from flask import Flask, render_template
from flask.ext.mongoengine import MongoEngine
from flask.ext.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)
    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()
```

Basic Peewee Application

Peewee Install requirements

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

Peewee Application

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

```
from flask import Flask, render_template
from flask_peewee.db import Database
from peewee import *
from flask.ext.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('/')
@login_required
def home():
    return render_template('index.html')

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

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
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.

Models

Flask-Security assumes you'll be using libraries such as SQLAlchemy, MongoEngine or Peewee 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`

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`

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.

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*
- *security/reset_password.html*
- *security/change_password.html*
- *security/send_confirmation.html*
- *security/send_login.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

- `register_context_processor`: Register view
- `reset_password_context_processor`: Reset password view
- `change_password_context_processor`: Reset password view
- `send_confirmation_context_processor`: Send confirmation view
- `send_login_context_processor`: Send login view

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

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

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`: Reset password form
- `send_confirmation_form`: Send confirmation form
- `passwordless_login_form`: Passwordless login form

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*
- *security/email/reset_notice.txt*
- *security/email/welcome.html*
- *security/email/welcome.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 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:

```
security = Security(app, user_datastore)

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

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:

```
# 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)
```

API

Core

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

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

Parameters

- **`app`** – The application.
- **`datastore`** – An instance of a user datastore.

`init_app`(*app*, *datastore=None*, *register_blueprint=True*, *login_form=None*, *confirm_register_form=None*, *register_form=None*, *forgot_password_form=None*, *reset_password_form=None*, *change_password_form=None*, *send_confirmation_form=None*, *passwordless_login_form=None*)

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

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.

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:

```
@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.decorators.**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 **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:

```
@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. The username should be set to the user's email address.

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*

User Object Helpers

class flask_security.core.**UserMixin**

Mixin for *User* model definitions

get_auth_token()

Returns the user's authentication token.

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.

class flask_security.core.**RoleMixin**

Mixin for *Role* model definitions

class flask_security.core.**AnonymousUser**

AnonymousUser definition

has_role(**args*)

Returns *False*

Datastores

class `flask_security.datastore.`**`UserDatastore`**(*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.

class `flask_security.datastore.`**`SQLAlchemyUserDatastore`**(*db*, *user_model*, *role_model*)
A SQLAlchemy datastore implementation for Flask-Security that assumes the use of the Flask-SQLAlchemy 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.

`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.

`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.

`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.

Utils

`flask_security.utils.`**`login_user`**(*user*, *remember=None*)
Performs the login routine.

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*)
Encrypts the specified plaintext password using the configured encryption options.

Parameters **`password`** – The plaintext password to encrypt `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*)

Get the status of a token.

Parameters

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

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. It is passed a dict with the *user* and *confirm_token*, the user being logged in and the (if so configured) the confirmation token issued.

`user_confirmed`
Sent when a user is confirmed. It is passed *user*, which is the user being confirmed.

`confirm_instructions_sent`
Sent when a user requests confirmation instructions. It is passed the *user*.

`login_instructions_sent`
Sent when passwordless login is used and user logs in. It is passed a dict with the *user* and *login_token*, the user being logged in and the (if so configured) the login token issued.

`password_reset`
Sent when a user completes a password reset. It is passed the *user*.

`password_changed`
Sent when a user completes a password change. It is passed the *user*.

`reset_password_instructions_sent`
Sent when a user requests a password reset. It is passed a dict with the *user* and *token*, the user being logged in and the (if so configured) the reset token issued.

All signals are also passed a *app* keyword argument, which is the current application.

Flask-Security Changelog

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

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

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

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.

Version 1.7.1

Released January 14th 2014

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

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* configuraiton 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

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

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

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

Version 1.6.6

Released June 28th 2013

- Fixed dependency versions

Version 1.6.5

Released June 20th 2013

- Fixed bug in *flask.ext.security.confirmable.generate_confirmation_link*

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

Version 1.6.3

Released May 8th 2013

- Fixed bug in regards to imports with latest version of MongoEngine

Version 1.6.2

Released April 4th 2013

- Fixed bug with http basic auth

Version 1.6.1

Released April 3rd 2013

- Fixed bug with signals

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

Version 1.5.4

Released January 6th 2013

- Fix bug in forms with *csrf_enabled* parameter not accounting attempts to login using JSON data

Version 1.5.3

Released December 23rd 2012

- Change dependency requirement

Version 1.5.2

Released December 11th 2012

- Fix a small bug in *flask_security.utils.login_user* method

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

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.

Version 1.2.3

Released June 12th 2012

- Fixed a bug in the RoleMixin eq/ne functions

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

Version 1.2.1

Released March 28th 2012

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

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.

Version 1.1.0

Initial release

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

Development Lead

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

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 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

Index