Flask-Security Documentation

Выпуск 1.7.4

Matt Wright

авг. 28, 2017

Оглавление

Flask-security позволяет быстро добавлять механизмы безопасности для вашего приложения. Он включает в себя:

1. Аутентификация на основе сессии
2. Управление ролями (группами пользователей)
3. Шифрование пароля
4. HTTP-аутентификацию
5. Токен аутентификацию
6. Активация учетной записи на основе токена (опционально)
7. Токен восстановления/сброса пароля (опционально)
8. Регистрация пользователей (опционально)
9. Отслеживание входа (опционально)
10. Поддержка JSON/AJAX

Многие из этих функций стали возможными благодаря интеграции с различными flask-расширениями и библиотеками. А именно:

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

Кроме того, предполагается, что вы будете использовать одну из представленных ниже библиотек, для соединения с базой данных и определения моделей. Flask-Security поддерживает следующие расширения из коробки:

1. Flask-SQLAlchemy
2. Flask-MongoEngine
3. Flask-Peewee

Глава 1

Содержание

Особенности

Flask-security позволяет быстро добавлять общие механизмы безопасности для вашего приложения. Он включает в себя:

Аутентификация на основе сессии

Аутентификация на основе сессии осуществляется исключительно за счет расширения Flask-Login. Flask-Security автоматически обрабатывает конфигурацию Flask-Login на основе нескольких собственных конфигурационных значений и использует **'Flask-Login токен'_** для запоминания пользователей, когда их сессия истекла.

Role/Identity Based Access

Flask-Security реализует очень простое управление группами пользователей из коробки. Это означает, что вы можете привязать к одному пользователю, как высокий уровень привилегий, так и несколько ролей сразу. Например, вы можете назначить следующие роли пользователям: администратор, редактор, суперпользователь, или их комбинацию. Контроль доступа на основе ролей и все их имена должны быть уникальными. Эта функция реализуется с помощью расширения Flask-Principal. Если вы хотели бы осуществлять более продвинутый контроль доступа, вы можете обратиться к Flask-Principal **'документации'_**.

Шифрование паролей

Шифрование паролей реализовано через passlib. По умолчанию, пароли хранятся в виде обычного текста, но вы можете легко настроить алгоритм шифрования. Вы должны всегда использовать алгоритм шифрования в вашем приложении, подробнее вы можете прочитать в моей статье.

HTTP-аутентификация

HTTP-аутентификация использует простой декоратор. Эта функция ожидает входящие данные аутентификации для идентификации пользователя в системе. Это означает, что имя пользователя должно быть равно адресу его электронной почты.

Токен авторизации

Проверка авторизации на основе токена включена путем извлечения токена аутентификации пользователя, путем отправки HTTP POST с деталями аутентификации. Успешный запрос вернет ID пользователя и токен аутентификации. Этот токен может быть использован в последующих запросах к защищенным ресурсам. Токен аутентификации передается в запросе через HTTP-заголовки или в GET параметрах. По умолчанию, имя HTTP-заголовка Authentication-Token, а стандартное имя GET параметра auth_token. Токены аутентификации генерируются с помощью пароля пользователя. Таким образом, если изменяется пароль пользователя, то существующий токен становится недействительным. Соответственно, новый токен будет получен уже с помощью нового пароля пользователя.

Подтверждающее письмо о регистрации

При желании вы можете потребовать, чтобы новый пользователь подтвердил, указанный при регистрации, адрес электронной почты. Flask-Security будет отправлять email-сообщение для этого пользователя, с ссылкой для подтверждения своего аккаунта. После перехода по ссылке, пользователь будет автоматически зарегистрирован. Так же, существует возможность для повторной отправки подтверждающей ссылки, если пользователи пытался активировать уже просроченный токен или при потере доступа к предыдущему email-адресу.

Сброс/восстановление пароля

Flask-Security отправляет сообщение электронной почты пользователю со ссылкой, по которой они могут сбросить свой пароль. Когда пароль будет сброшен, они автоматически войдут в систему и с того момента, смогут использовать новый пароль.

Регистрация пользователя

Flask-Security предоставляет функционал регистрации пользователей. Это очень просто, ведь для регистрации нового пользователя необходимо ввести только адрес своей электронной почты и пароль. Количество полей можно легко изменить, если ваш процесс регистрации требует больше данных.

Логгирование

Flask-Security, если настроен на эту опцию, может отслеживать основные события входа в систему. А именно:

- Дата последнего входа в систему
- Время текущего входа
- Последний IP адрес
- Текущий IP адрес
- Общее количество авторизаций

Поддержка AJAX/JSON

Flask-Security поддерживает формат json/AJAX-запросов там, где это уместно. Просто помните, что все конечные точки требуют csrf-токен. Более конкретно, json поддерживается для следующих запросов:

- 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 string 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_TOKEN_MAX_AGE`	Specifies the number of seconds before an authentication token expires. Defaults to None, meaning the token never expires.
`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. Be sure to use *ProxyFix <http://flask.pocoo.org/docs/0.10/deploying/wsgi-standalone/#proxy-setups>* if you are using a proxy. 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`.

Quick Start

- *Basic SQLAlchemy Application*
- *Basic MongoEngine Application*
- *Basic Peewee Application*
- *Mail Configuration*

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 = StringField('First Name', [Required()])
    last_name = StringField('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

`flask_security.core.current_user`
A proxy for the current user.

Protecting Views

User Object Helpers

Datastores

Utils

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.

Алфавитный указатель