Flask-User Documentation

Release v0.6

Ling Thio and contributors

Apr 19, 2018

Contents

1 Secure and Reliable **3**

2 Well documented **5**

3 Fully customizable, yet Ready to use **7**

4 Comes with translations **9**

5 Requirements **11**

6 Alternatives **13**

7 Table of Contents **15**

7.1 Design Goals . . . 15
7.2 Limitations . . . 16
7.3 Installation . . . 18
7.4 User DataModels . . . 18
7.5 Basic App . . . 21
7.6 Flask-User-starter-app . . . 25
7.7 Authorization . . . 25
7.8 Roles Required App . . . 28
7.9 Base templates . . . 32
7.10 Customization . . . 33
7.11 Signals (event hooking) . . . 44
7.12 Recipes . . . 45
7.13 Internationalization . . . 47
7.14 F.A.Q. . . . 50
7.15 Flask-User API . . . 51

8 Revision History **59**

9 Acknowledgements **63**

10 Contributors **65**

11 Alternative Flask extensions **67**

Customizable User Authentication & Management

Attention

IMPORTANT: Flask-User v1.0 is under development and breaks backward compatibility with Flask-User v0.6.

To avoid disruption, please take the time to pin your Flask-User version in your `requirements.txt`. For example: `Flask-User==0.6.21`

So, you're writing a Flask web application and would like to authenticate your users.
You start with a simple **Login** page, but soon enough you'll need to handle:

- **Registrations** and **Email Confirmations**
- **Change Usernames**, **Change Passwords**, and **Forgotten Passwords**

And wouldn't it be nice to also offer:

- **Added security**
- **Increased reliability**
- **Role-based Authorization**
- **Internationalization**
- **Support for multiple emails per user**

Flask-User offers these features and more.

CHAPTER 1

Secure and Reliable

- **Secure** – Built on top of widely deployed Passlib, PyCrypto, ItsDangerous.
- **Reliable** – Code coverage of over 95%
- **Available** – Tested on Python 2.6, 2.7, 3.3, 3.4, 3.5 and 3.6

CHAPTER 2

Well documented

- Flask-User v0.6 documentation
- Flask-User v0.5 documentation

CHAPTER 3

Fully customizable, yet Ready to use

- **Largely configurable** – Through configuration settings
- **Fully customizable** – Through customizable functions and email templates
- **Ready to use** – Through sensible defaults
- Supports **SQL Databases** – Through SQLAlchemy
- **Event hooking** – Through signals

CHAPTER 4

Comes with translations

Chinese, Dutch, English, Farsi, Finnish, French, German, Italian, Russian, Spanish, Swedish, and Turkish

CHAPTER 5

Requirements

Flask-User requires the following Python packages:

- Flask 0.9+
- Flask-Babel 0.9+
- Flask-Login 0.3+
- Flask-Mail 0.9+
- Flask-SQLAlchemy 1.0+
- Flask-WTF 0.9+
- passlib 1.6+
- pycryptodome
- speaklater 1.3+

Optionally:

- blinker 1.3+ – for Event Notification
- Flask-Sendmail – for sending emails via `sendmail`
- py-bcript 0.4+ – for fast bcrypt encryption

CHAPTER 6

Alternatives

- Flask-Login
- Flask-Security

CHAPTER 7

Table of Contents

7.1 Design Goals

7.1.1 Reliable

We understand that you are looking for an easy yet reliable way to manage your users. We've run our code through automated tests from the very beginning and we're proud to consistently achieve code coverage of over 90%.

7.1.2 Secure

Passwords are hashed using **bcrypt** by default and can be customized to any set of hashing algorithms that **passlib** supports.

Tokens are encrypted using **AES**, signed using the **itsdangerous** package, and expire after a configurable period of time.

7.1.3 Fully Customizable

We offer as much customization as possible through the use of configuration settings. The remainder is customizable by writing and configuring your own custom functions. See *Customization*.

7.1.4 Ready to use

Installing is as easy as: `pip install flask-user`. See *Installation*.

Through the use of **sensible defaults**, our fully customizable package is also ready-to-use. The *Basic App* requires only a dozen lines of additional code and all the default web forms and email templates could be used in production as-is.

7.1.5 Great Feature Set

- Login with username or email or both, Remember me, Logout
- Register, Confirm email, Resend confirmation email
- Forgot password, Change username, Change password
- Secure password hashing and token generation
- Role-based Authorization – See *Authorization*
- Internationalization – See *Internationalization*
- Signals (event hooking) – See *Signals (event hooking)*

7.1.6 Also

- Well documented
- Database ORM abstraction (SQLAlchemyAdapter provided)

7.2 Limitations

As you will soon experience, a great many things in Flask-User can be customized so it can behave exactly the way you want it to behave. But this documentation would not be complete without first discussing what its limitations are.

7.2.1 Supported Databases

Out-of-the box, Flask-User ships with a SQLAlchemyAdapter, allowing support for many SQL databases including:

- Drizzle
- Firebird
- Microsoft SQL Server
- MySQL
- Oracle
- PostgreSQL
- SQLite
- Sybase

For a full list see http://docs.sqlalchemy.org/en/rel_0_9/dialects/index.html

Flask-User does abstract DB interactions through a 'DbAdapter' class, so support for other databases is possible by writing a DbAdapter extension class.

7.2.2 Database table names

No known restrictions

7.2.3 Database column names

No known restrictions

7.2.4 Primary keys

The primary key of the User table must be an Integer and may not be a compound key.

7.2.5 Data model field names

Flask-User requires specific Data model field names, but accepts arbitrary names for the Database column names.

Required Data model field names:

```
# User authentication information
User.id
User.username               or  UserAuth.username
User.password               or  UserAuth.password
                                UserAuth.user_id

# User email information
User.email                  or  UserEmail.email
User.confirmed_at           or  UserEmail.confirmed_at
                                UserEmail.user_id

# User information
User.active

# Relationships
User.roles                  # only if @roles_required is used

# Role information
Role.name
```

SQLAlchemy offers a way to use specific Data model field names with different Database column names:

```
class User(db.Model, UserMixin)

    # Map Data model field 'email' to Database column 'email_address'
    email = db.Column('email_address', db.String(100))

    # Map Data model field 'active' to Database column 'is_active'
    active = db.Column('is_active', db.Boolean())
```

7.2.6 Flask versions

Flask-User has been tested with Flask 0.10

7.2.7 Python versions

Flask-User has been tested with Python 2.6, 2.7, 3.3, 3.4, 3.5 and 3.6

7.3 Installation

We recommend making use of virtualenv and virtualenvwrapper:

```
mkvirtualenv my_env
workon my_env
```

7.3.1 Installation Instructions

After setting up virtualenv, installation is as easy as:

```
workon my_env
pip install flask-user==0.6.15
```

7.3.2 Requirements

- Python 2.6, 2.7, 3.3+
- Flask 0.10+
- Flask-Login 0.2+
- Flask-Mail 0.9+ or Flask-Sendmail
- Flask-WTF 0.9+
- passlib 1.6+
- pycrypto 2.6+
- py-bcript 0.4+ # Recommended for speed, and only if bcrypt is used to hash passwords

When using the included SQLAlchemyAdapter, Flask-User requires:

- Flask-SQLAlchemy 1.0+ (with a driver such as MySQL-Python or PyMySQL)

Optional requirements for Event Notification:

- blinker 1.3+

Optional requirements for Internationalization:

- Flask-Babel 0.9+
- speaklater 1.3+

7.3.3 Up Next

Basic App

7.4 User DataModels

Flask-User distinguishes between the following groups of user information:

1. User Authentication information such as username and password
2. User Email information such as email address and confirmed_at

3. User information such as first_name and last_name
4. User Role information

Flask-User allows the developer to store Authentication, Email and User information in one DataModel or across several DataModels.

Flask-User requires User Role information to be stored in a Role DataModel and an UserRole association table.

7.4.1 All-in-one User DataModel

If you'd like to store all user information in one DataModel, use the following:

```
# Define User model. Make sure to add flask_user UserMixin !!!
class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)

    # User Authentication information
    username = db.Column(db.String(50), nullable=False, unique=True)
    password = db.Column(db.String(255), nullable=False, default='')

    # User Email information
    email = db.Column(db.String(255), nullable=False, unique=True)
    confirmed_at = db.Column(db.DateTime())

    # User information
    is_enabled = db.Column(db.Boolean(), nullable=False, default=False)
    first_name = db.Column(db.String(50), nullable=False, default='')
    last_name = db.Column(db.String(50), nullable=False, default='')

    def is_active(self):
      return self.is_enabled

# Setup Flask-User
db_adapter = SQLAlchemyAdapter(db, User)        # Register the User model
user_manager = UserManager(db_adapter, app)     # Initialize Flask-User
```

7.4.2 Separated User/UserAuth DataModel

If you'd like to store User Authentication information separate from User information, use the following:

```
# Define User DataModel
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)

    # User email information
    email = db.Column(db.String(255), nullable=False, unique=True)
    confirmed_at = db.Column(db.DateTime())

    # User information
    is_enabled = db.Column(db.Boolean(), nullable=False, default=False)
    first_name = db.Column(db.String(50), nullable=False, default='')
    last_name = db.Column(db.String(50), nullable=False, default='')

    def is_active(self):
      return self.is_enabled
```

```
# Define UserAuth DataModel. Make sure to add flask_user UserMixin!!
class UserAuth(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer(), db.ForeignKey('user.id', ondelete='CASCADE'))

    # User authentication information
    username = db.Column(db.String(50), nullable=False, unique=True)
    password = db.Column(db.String(255), nullable=False, default='')

    # Relationships
    user = db.relationship('User', uselist=False, foreign_keys=user_id)

# Setup Flask-User
db_adapter = SQLAlchemyAdapter(db,  User, UserAuthClass=UserAuth)
user_manager = UserManager(db_adapter, app)
```

7.4.3 UserEmail DataModel

Separating User Email information from User information allows for support of multiple emails per user.

It can be applied to both the All-in-one User DataModel and the separated User/UserAuth DataModel

```
# Define User DataModel. Make sure to add flask_user UserMixin !!!
class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    ...
    # Relationship
    user_emails = db.relationship('UserEmail')

# Define UserEmail DataModel.
class UserEmail(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'))

    # User email information
    email = db.Column(db.String(255), nullable=False, unique=True)
    confirmed_at = db.Column(db.DateTime())
    is_primary = db.Column(db.Boolean(), nullable=False, default=False)

    # Relationship
    user = db.relationship('User', uselist=False)
```

7.4.4 User Roles DataModel

The Roles table holds the name of each role. This name will be matched to the @roles_required function decorator in a CASE SENSITIVE manner.

```
# Define the Role DataModel
class Role(db.Model):
    id = db.Column(db.Integer(), primary_key=True)
    name = db.Column(db.String(50), unique=True)
```

The UserRoles DataModel associates Users with their Roles.

It can be applied to both the All-in-one User DataModel and the separated User/UserAuth DataModel

```
# Define the User DataModel. Make sure to add flask_user UserMixin!!
class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    ...
    # Relationships
    roles = db.relationship('Role', secondary='user_roles',
            backref=db.backref('users', lazy='dynamic'))

# Define the UserRoles DataModel
class UserRoles(db.Model):
    id = db.Column(db.Integer(), primary_key=True)
    user_id = db.Column(db.Integer(), db.ForeignKey('user.id', ondelete='CASCADE'))
    role_id = db.Column(db.Integer(), db.ForeignKey('role.id', ondelete='CASCADE'))
```

7.4.5 Porting Flask-User v0.5 applications to Flask-User v0.6

For applications using the All-in-one User DataModel, no changes are required.

For applications using the separated User/UserAuth DataModel, v0.6 maintains backward compatibility, but future versions may not, and it is therefore recommended to make the following changes:

- Change `SQLAlchemyAdapter(db, User, UserProfile=UserProfile)` to `SQLAlchemyAdapter(db, UserProfile, UserAuth=User)`.
- Move the UserMixin from `class User(db.Model)` to `class UserProfile(db.Model, UserMixin)`
- Move the `roles` relationship from class User to class UserProfile.
- Move the UserRoles.user_id association from 'user.id' to 'user_profile.id'. This requires a DB schema change.
- If it's possible to rename table names, please rename User to UserAuth and UserProfile to User. This would require a DB schema change.

7.5 Basic App

The sample code below illustrates the power of using Flask-User with sensible defaults: With just a dozen additional code statements, a basic Flask application can be transformed to offer the following features:

- Register with username and email
- Email confirmation
- Login with username or email, Logout
- Protect pages from unauthenticated access
- Change username
- Change password
- Forgot password

7.5.1 Single-file techniques

To keep the examples simple, we are using some unusual single-file techniques:
- Using class based configuration instead of file based configuration

- Using `render_template_string()` instead of `render_template()`
- Placing everything in one file

None of these techniques are recommended outside of tutorial usage.

7.5.2 Setup a development environment

These tutorials assume that you are working with virtualenv and virtualenvwrapper and that the code resides in ~/dev/example:

```
# Create virtualenv 'example'
mkvirtualenv example

# Install required Python packages in the 'example' virtualenv
workon example
pip install flask-user
pip install flask-mail

# Change working directory
mkdir -p ~dev/example
cd ~/dev/example                # or C:\dev\example on Windows
```

7.5.3 Create the basic_app.py file

Create ~/dev/example/basic_app.py with the content below.

Make sure to replace the following settings:

MAIL_USERNAME = 'email@example.com'
MAIL_PASSWORD = 'password'
MAIL_DEFAULT_SENDER = '"Sender" <noreply@example.com>'
MAIL_SERVER = 'smtp.gmail.com'
MAIL_PORT = 465
MAIL_USE_SSL = True
MAIL_USE_TLS = False

with settings that are appropriate for your SMTP server.

Highlighted lines shows the lines added to a basic Flask application.

```
1  import os
2  from flask import Flask, render_template_string
3  from flask_babel import Babel
4  from flask_mail import Mail
5  from flask_sqlalchemy import SQLAlchemy
6  from flask_user import login_required, UserManager, UserMixin, SQLAlchemyAdapter
7
8
9  # Use a Class-based config to avoid needing a 2nd file
10 # os.getenv() enables configuration through OS environment variables
11 class ConfigClass(object):
```

```
    # Flask settings
    SECRET_KEY =              os.getenv('SECRET_KEY',       'THIS IS AN INSECURE SECRET')
    SQLALCHEMY_DATABASE_URI = os.getenv('DATABASE_URL',     'sqlite:///basic_app.sqlite')
    CSRF_ENABLED = True

    # Flask-Mail settings
    MAIL_USERNAME =           os.getenv('MAIL_USERNAME',        'youremail@example.com')
    MAIL_PASSWORD =           os.getenv('MAIL_PASSWORD',        'yourpassword')
    MAIL_DEFAULT_SENDER =     os.getenv('MAIL_DEFAULT_SENDER',  '"MyApp" <noreply@example.com>')
    MAIL_SERVER =             os.getenv('MAIL_SERVER',          'smtp.gmail.com')
    MAIL_PORT =           int(os.getenv('MAIL_PORT',            '465'))
    MAIL_USE_SSL =        int(os.getenv('MAIL_USE_SSL',         True))

    # Flask-User settings
    USER_APP_NAME        = "AppName"                # Used by email templates


def create_app():
    """ Flask application factory """

    # Setup Flask app and app.config
    app = Flask(__name__)
    app.config.from_object(__name__+'.ConfigClass')

    # Initialize Flask-BabelEx
    babel = Babel(app)

    # Initialize Flask extensions
    db = SQLAlchemy(app)                            # Initialize Flask-SQLAlchemy
    mail = Mail(app)                                # Initialize Flask-Mail

    # Define the User data model.
    # Make sure to add flask_user UserMixin !!!
    class User(db.Model, UserMixin):
        id = db.Column(db.Integer, primary_key=True)

        # User authentication information
        username = db.Column(db.String(50), nullable=False, unique=True)
        password = db.Column(db.String(255), nullable=False, server_default='')

        # User email information
        email = db.Column(db.String(255), nullable=False, unique=True)
        confirmed_at = db.Column(db.DateTime())

        # User information
        active = db.Column('is_active', db.Boolean(), nullable=False, server_default='0')
        first_name = db.Column(db.String(100), nullable=False, server_default='')
        last_name = db.Column(db.String(100), nullable=False, server_default='')

    # Create all database tables
    db.create_all()

    # Setup Flask-User
```

```
    db_adapter = SQLAlchemyAdapter(db, User)        # Register the User model
    user_manager = UserManager(db_adapter, app)     # Initialize Flask-User

    # The Home page is accessible to anyone
    @app.route('/')
    def home_page():
        return render_template_string("""
            {% extends "base.html" %}
            {% block content %}
                <h2>Home page</h2>
                <p>This page can be accessed by anyone.</p><br/>
                <p><a href={{ url_for('home_page') }}>Home page</a> (anyone)</p>
                <p><a href={{ url_for('members_page') }}>Members page</a> (login required)</p>
            {% endblock %}
            """)

    # The Members page is only accessible to authenticated users
    @app.route('/members')
    @login_required                                 # Use of @login_required decorator
    def members_page():
        return render_template_string("""
            {% extends "base.html" %}
            {% block content %}
                <h2>Members page</h2>
                <p>This page can only be accessed by authenticated users.</p><br/>
                <p><a href={{ url_for('home_page') }}>Home page</a> (anyone)</p>
                <p><a href={{ url_for('members_page') }}>Members page</a> (login required)</p>
            {% endblock %}
            """)

    return app


# Start development web server
if __name__=='__main__':
    app = create_app()
    app.run(host='0.0.0.0', port=5000, debug=True)

```

7.5.4 Run the Basic App

Run the Basic App with the following command:

```
cd ~/dev/example
python basic_app.py
```

And point your browser to `http://localhost:5000`.

7.5.5 Troubleshooting

If you receive an SendEmailError message, or if the Registration form does not respond quickly then you may have specified incorrect SMTP settings.

If you receive a 'AssertionError: No sender address has been set' error, you may be using an old version of Flask-Mail which uses DEFAULT_MAIL_SENDER instead of MAIL_DEFAULT_SENDER.

If you receive a SQLAlchemy error message, delete the basic_app.sqlite file and restart the app. You may be using an old DB schema in that file.

7.5.6 Up Next

Flask-User-starter-app

7.6 Flask-User-starter-app

A more typical Flask application has modularized code organized in a directory structure.

An example of such a Flask-User application is the Flask-User-starter-app, which available on Github:

https://github.com/lingthio/Flask-User-starter-app

It can serve as a great starter app for building your next Flask-User application.

Files of interest:

- app/startup/init_app.py
- app/models/user.py
- app/templates/flask_user/*.html
- app/templates/users/user_profile_page.html

7.6.1 Up Next

Authorization

7.7 Authorization

Authorization is the process of specifying and enforcing access rights of users to resources.

Flask-User offers role based authorization through the use of function decorators:

- *@login_required*
- *@roles_required*

7.7.1 @login_required

Decorate a view function with @login_required to ensure that the user is logged in before accessing that particular page:

```
from flask_user import login_required

@route('/profile')                    # @route() must always be the outer-most␣
↪decorator
@login_required
```

```
def profile_page():
    # render the user profile page
```

Flask-User relies on Flask-Login to implement and offer the @login_required decorator along with its underlying current_user.is_authenticated() implementation.
See the Flask-Login Documentation

7.7.2 @roles_required

Decorate a view function with @roles_required to ensure that the user is logged in and has sufficient role-based access rights that particular page.

In the example below the current user is required to have the 'admin' role:

```
from flask_user import roles_required

@route('/admin/dashboard')           # @route() must always be the outer-most␣
↪decorator
@roles_required('admin')
def admin_dashboard():
    # render the admin dashboard
```

Note: Comparison of role names is case sensitive, so 'Member' will NOT match 'member'.

Multiple string arguments – the AND operation

The @roles_required decorator accepts multiple strings if the current_user is required to have **ALL** of these roles.

In the example below the current user is required to have the **ALL** of these roles:

```
@roles_required('dark', 'tall', 'handsome')
# Multiple string arguments require ALL of these roles
```

Multiple string arguments represent the 'AND' operation.

Array arguments – the OR operation

The @roles_required decorator accepts an array (or a tuple) of roles.

In the example below the current user is required to have **One or more** of these roles:

```
@roles_required(['funny', 'witty', 'hilarious'])
# Notice the usage of square brackets representing an array.
# Array arguments require at least ONE of these roles.
```

AND/OR operations

The potentially confusing syntax described above allows us to construct complex AND/OR operations.

In the example below the current user is required to have

either (the 'starving' AND the 'artist' roles)
OR (the 'starving AND the 'programmer' roles)

```
@roles_required('starving', ['artist', 'programmer'])
# Ensures that the user is ('starving' AND (an 'artist' OR a 'programmer'))
```

Note: The nesting level only goes as deep as this example shows.

7.7.3 Required Tables

For @login_required only the User model is required

For @roles_required, the database must have the following models:

- The usual User model with an additional 'roles' relationship field
- A Role model with at least one string field called 'name'
- A UserRoles association model with a 'user_id' field and a 'role_id' field

Here's a SQLAlchemy example:

```
# Define User model
class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(50), nullable=True, unique=True)
    ...
    roles = db.relationship('Role', secondary='user_roles',
            backref=db.backref('users', lazy='dynamic'))

# Define Role model
class Role(db.Model):
    id = db.Column(db.Integer(), primary_key=True)
    name = db.Column(db.String(50), unique=True)

# Define UserRoles model
class UserRoles(db.Model):
    id = db.Column(db.Integer(), primary_key=True)
    user_id = db.Column(db.Integer(), db.ForeignKey('user.id', ondelete='CASCADE'))
    role_id = db.Column(db.Integer(), db.ForeignKey('role.id', ondelete='CASCADE'))
```

Roles are defined by adding rows to the role table with a specific Role.name value.

```
# Create 'user007' user with 'secret' and 'agent' roles
user1 = User(username='user007', email='user007@example.com', is_enabled=True,
             password=user_manager.hash_password('Password1'))
role1 = Role(name='secret')
role2 = Role(name='agent')
```

Users are assigned one or more roles by adding a records to the 'user_roles' table, binding a User to one or more Roles.

```
# Bind user to two roles
user1.roles.append(role1)
user1.roles.append(role2)

# Store user and roles
```

```
db.session.add(user1)
db.session.commit()
```

7.7.4 Up Next

Roles Required App

7.8 Roles Required App

The Roles Required App builds on the features of *Basic App*:

- Register with username and email
- Email confirmation
- Login with username or email, Logout
- Protect pages from unauthenticated access
- Change username
- Change password
- Forgot password

And adds the following:

- Role-based Authorization

7.8.1 Single-file techniques

To keep the examples simple, we are using some unusual single-file techniques:
- Using class based configuration instead of file based configuration
- Using `render_template_string()` instead of `render_template()`
- Placing everything in one file

None of these techniques are recommended outside of tutorial usage.

7.8.2 Setup a development environment

These tutorials assume that you are working with virtualenv and virtualenvwrapper and that the code resides in ~/dev/example:

```
# Create virtualenv 'example'
mkvirtualenv example

# Install required Python packages in the 'example' virtualenv
workon example
pip install flask-user
pip install flask-mail

# Change working directory
```

```
mkdir -p ~dev/example
cd ~/dev/example                # or C:\dev\example on Windows
```

7.8.3 Create roles_required_app.py

Create ~/dev/example/roles_required_app.py with the content below.

Make sure to replace the following settings:

MAIL_USERNAME = 'email@example.com'
MAIL_PASSWORD = 'password'
MAIL_DEFAULT_SENDER = '"Sender" <noreply@example.com>'
MAIL_SERVER = 'smtp.gmail.com'
MAIL_PORT = 465
MAIL_USE_SSL = True
MAIL_USE_TLS = False

with settings that are appropriate for your SMTP server.

Highlighted lines shows the lines added to the Basic App.

```
import os
from flask import Flask, render_template_string, request
from flask_mail import Mail
from flask_sqlalchemy import SQLAlchemy
from flask_user import login_required, SQLAlchemyAdapter, UserManager, UserMixin
from flask_user import roles_required


# Use a Class-based config to avoid needing a 2nd file
# os.getenv() enables configuration through OS environment variables
class ConfigClass(object):
    # Flask settings
    SECRET_KEY =              os.getenv('SECRET_KEY',       'THIS IS AN INSECURE␣
↪SECRET')
    SQLALCHEMY_DATABASE_URI = os.getenv('DATABASE_URL',     'sqlite:///single_file_
↪app.sqlite')
    CSRF_ENABLED = True

    # Flask-Mail settings
    MAIL_USERNAME =           os.getenv('MAIL_USERNAME',        'email@example.com')
    MAIL_PASSWORD =           os.getenv('MAIL_PASSWORD',        'password')
    MAIL_DEFAULT_SENDER =     os.getenv('MAIL_DEFAULT_SENDER',  '"MyApp"
↪<noreply@example.com>')
    MAIL_SERVER =             os.getenv('MAIL_SERVER',          'smtp.gmail.com')
    MAIL_PORT =           int(os.getenv('MAIL_PORT',            '465'))
    MAIL_USE_SSL =        int(os.getenv('MAIL_USE_SSL',         True))

    # Flask-User settings
    USER_APP_NAME        = "AppName"                # Used by email templates


def create_app(test_config=None):                  # For automated tests
    # Setup Flask and read config from ConfigClass defined above
```

```
    app = Flask(__name__)
    app.config.from_object(__name__+'.ConfigClass')

    # Load local_settings.py if file exists         # For automated tests
    try: app.config.from_object('local_settings')
    except: pass

    # Load optional test_config                     # For automated tests
    if test_config:
        app.config.update(test_config)

    # Initialize Flask extensions
    mail = Mail(app)                                # Initialize Flask-Mail
    db = SQLAlchemy(app)                            # Initialize Flask-SQLAlchemy

    # Define the User data model. Make sure to add flask_user UserMixin!!
    class User(db.Model, UserMixin):
        id = db.Column(db.Integer, primary_key=True)

        # User authentication information
        username = db.Column(db.String(50), nullable=False, unique=True)
        password = db.Column(db.String(255), nullable=False, server_default='')

        # User email information
        email = db.Column(db.String(255), nullable=False, unique=True)
        confirmed_at = db.Column(db.DateTime())

        # User information
        active = db.Column('is_active', db.Boolean(), nullable=False, server_default=
↪'0')
        first_name = db.Column(db.String(100), nullable=False, server_default='')
        last_name = db.Column(db.String(100), nullable=False, server_default='')

        # Relationships
        roles = db.relationship('Role', secondary='user_roles',
                backref=db.backref('users', lazy='dynamic'))

    # Define the Role data model
    class Role(db.Model):
        id = db.Column(db.Integer(), primary_key=True)
        name = db.Column(db.String(50), unique=True)

    # Define the UserRoles data model
    class UserRoles(db.Model):
        id = db.Column(db.Integer(), primary_key=True)
        user_id = db.Column(db.Integer(), db.ForeignKey('user.id', ondelete='CASCADE
↪'))
        role_id = db.Column(db.Integer(), db.ForeignKey('role.id', ondelete='CASCADE
↪'))

    # Reset all the database tables
    db.create_all()

    # Setup Flask-User
    db_adapter = SQLAlchemyAdapter(db,  User)
    user_manager = UserManager(db_adapter, app)

    # Create 'user007' user with 'secret' and 'agent' roles
```

```
    if not User.query.filter(User.username=='user007').first():
        user1 = User(username='user007', email='user007@example.com', active=True,
                password=user_manager.hash_password('Password1'))
        user1.roles.append(Role(name='secret'))
        user1.roles.append(Role(name='agent'))
        db.session.add(user1)
        db.session.commit()

    # The Home page is accessible to anyone
    @app.route('/')
    def home_page():
        return render_template_string("""
            {% extends "base.html" %}
            {% block content %}
                <h2>Home page</h2>
                <p>This page can be accessed by anyone.</p><br/>
                <p><a href={{ url_for('home_page') }}>Home page</a> (anyone)</p>
                <p><a href={{ url_for('members_page') }}>Members page</a> (login required)</p>
                <p><a href={{ url_for('special_page') }}>Special page</a> (login with username 'user007' and password 'Password1')</p>
            {% endblock %}
            """)

    # The Members page is only accessible to authenticated users
    @app.route('/members')
    @login_required                                 # Use of @login_required decorator
    def members_page():
        return render_template_string("""
            {% extends "base.html" %}
            {% block content %}
                <h2>Members page</h2>
                <p>This page can only be accessed by authenticated users.</p><br/>
                <p><a href={{ url_for('home_page') }}>Home page</a> (anyone)</p>
                <p><a href={{ url_for('members_page') }}>Members page</a> (login required)</p>
                <p><a href={{ url_for('special_page') }}>Special page</a> (login with username 'user007' and password 'Password1')</p>
            {% endblock %}
            """)

    # The Special page requires a user with 'special' and 'sauce' roles or with 'special' and 'agent' roles.
    @app.route('/special')
    @roles_required('secret', ['sauce', 'agent'])   # Use of @roles_required decorator
    def special_page():
        return render_template_string("""
            {% extends "base.html" %}
            {% block content %}
                <h2>Special Page</h2>
                <p>This page can only be accessed by user007.</p><br/>
                <p><a href={{ url_for('home_page') }}>Home page</a> (anyone)</p>
                <p><a href={{ url_for('members_page') }}>Members page</a> (login required)</p>
                <p><a href={{ url_for('special_page') }}>Special page</a> (login with username 'user007' and password 'Password1')</p>
            {% endblock %}
            """)
```

```
    return app


# Start development web server
if __name__=='__main__':
    app = create_app()
    app.run(host='0.0.0.0', port=5000, debug=True)
```

7.8.4 Run the Roles Required App

Run the Roles Required App with the following command:

```
cd ~/dev/example
python roles_required_app.py
```

And point your browser to `http://localhost:5000`.

7.8.5 Troubleshooting

If you receive an SendEmailError message, or if the Registration form does not respond quickly then you may have specified incorrect SMTP settings.

If you receive a 'AssertionError: No sender address has been set' error, you may be using an old version of Flask-Mail which uses DEFAULT_MAIL_SENDER instead of MAIL_DEFAULT_SENDER.

If you receive a SQLAlchemy error message, delete the roles_required_app.sqlite file and restart the app. You may be using an old DB schema in that file.

7.9 Base templates

templates/base.html

All Flask-User forms extend from the template file `tempates/base.h` and Flask-User supplies a built-in version that uses Bootstrap 3.

To make Flask-User use your page template, you will need to create a `base.html` template file in your application's `templates` directory.

Use `{% block content %}{% endblock %}` as a placeholder for the forms.

templates/flask_user/public_base.html

Public forms are forms that do not require a logged-in user:

- `templates/flask_user/forgot_password.html`,
- `templates/flask_user/login.html`,
- `templates/flask_user/login_or_register.html`,
- `templates/flask_user/register.html`,
- `templates/flask_user/resend_confirm_email.html`, and
- `templates/flask_user/reset_password.html`.

Public forms extend the template file `templates/flask_user/public_base.html`, which by default extends the template file `templates/base.html`.

If you want the public forms to use a base template file other than `templates/base.html`, create the `templates/flask_user/public_base.html` file in your application's `templates` directory with the following content:

```
{% extends 'my_public_base.html' %}
```

templates/flask_user/member_base.html

Member forms are forms that require a logged-in user:

- `templates/flask_user/change_password.html`,
- `templates/flask_user/change_username.html`, and
- `templates/flask_user/manage_emails.html`.

Member forms extend the template file `templates/flask_user/member_base.html`, which by default extends the template file `templates/base.html`.

If you want the member forms to use a base template file other than `templates/base.html`, create the `templates/flask_user/member_base.html` file in your application's `templates` directory with the following content:

```
{% extends 'my_member_base.html' %}
```

Summary

The following template files reside in the `templates` directory:

```
base.html                              # root template

flask_user/member_base.html            # extends base.html
flask_user/change_password.html        # extends flask_user/member_base.html
flask_user/change_username.html        # extends flask_user/member_base.html
flask_user/manage_emails.html          # extends flask_user/member_base.html

flask_user/public_base.html            # extends base.html
flask_user/forgot_password.html        # extends flask_user/public_base.html
flask_user/login.html                  # extends flask_user/public_base.html
flask_user/login_or_register.html      # extends flask_user/public_base.html
flask_user/register.html               # extends flask_user/public_base.html
flask_user/resend_confirm_email.html   # extends flask_user/public_base.html
flask_user/reset_password.html         # extends flask_user/public_base.html
```

7.10 Customization

Flask-User has been designed with full customization in mind, and and here is a list of behaviors that can be customized as needed:

- *Features*
- *Settings*
- *Emails*
- *Registration Form*

- *Labels and Messages*
- *Form Classes*
- *Form Templates*
- *View functions*
- *Password and Username validators*
- *Password hashing*
- *URLs*
- *Endpoints*
- *Email template filenames*
- *Form template filenames*
- *Token generation*

7.10.1 Features

The following Features can be customized through the application's config:

```
# Features                      # Default   # Description
USER_ENABLE_CHANGE_PASSWORD     = True      # Allow users to change their password

USER_ENABLE_CHANGE_USERNAME     = True      # Allow users to change their username
                                            # Requires USER_ENABLE_USERNAME=True

USER_ENABLE_CONFIRM_EMAIL       = True      # Force users to confirm their email
                                            # Requires USER_ENABLE_EMAIL=True

USER_ENABLE_FORGOT_PASSWORD     = True      # Allow users to reset their passwords
                                            # Requires USER_ENABLE_EMAIL=True

USER_ENABLE_LOGIN_WITHOUT_CONFIRM_EMAIL = False
                                            # Allow users to login without a
                                            # confirmed email address
                                            # Protect views using @confirm_email_
↪required

USER_ENABLE_EMAIL               = True      # Register with Email
                                            # Requires USER_ENABLE_REGISTRATION=True

USER_ENABLE_MULTIPLE_EMAILS     = False     # Users may register multiple emails
                                            # Requires USER_ENABLE_EMAIL=True

USER_ENABLE_REGISTRATION        = True      # Allow new users to register

USER_ENABLE_RETYPE_PASSWORD     = True      # Prompt for `retype password` in:
                                            #   - registration form,
                                            #   - change password form, and
                                            #   - reset password forms.

USER_ENABLE_USERNAME            = True      # Register and Login with username
```

The following config settings have been renamed and are now obsolete. Please rename to the new setting.

```
# Obsoleted setting              # New setting
USER_ENABLE_EMAILS               USER_ENABLE_EMAIL
USER_ENABLE_USERNAMES            USER_ENABLE_USERNAME
USER_ENABLE_RETYPE_PASSWORDS     USER_ENABLE_RETYPE_PASSWORD
USER_LOGIN_WITH_USERNAME         USER_ENABLE_USERNAME
USER_REGISTER_WITH_EMAIL         USER_ENABLE_EMAIL
USER_RETYPE_PASSWORD             USER_ENABLE_RETYPE_PASSWORD
```

7.10.2 Settings

The following Settings can be customized through the application's config:

```
# Settings                       # Default     # Description
USER_APP_NAME                    = 'AppName'   # Used by email templates

USER_AUTO_LOGIN                  = True

USER_AUTO_LOGIN_AFTER_CONFIRM    = USER_AUTO_LOGIN

USER_AUTO_LOGIN_AFTER_REGISTER   = USER_AUTO_LOGIN

USER_AUTO_LOGIN_AFTER_RESET_PASSWORD = USER_AUTO_LOGIN

USER_AUTO_LOGIN_AT_LOGIN         = USER_AUTO_LOGIN

USER_CONFIRM_EMAIL_EXPIRATION    = 2*24*3600   # Confirmation expiration in seconds
                                               # (2*24*3600 represents 2 days)

USER_INVITE_EXPIRATION           = 90*24*3600  # Invitation expiration in seconds
                                               # (90*24*3600 represents 90 days)
                                               # v0.6.2 and up

USER_PASSWORD_HASH               = 'bcrypt'    # Any passlib crypt algorithm

USER_PASSWORD_HASH_MODE          = 'passlib'   # Set to 'Flask-Security' for
                                               # Flask-Security compatible hashing

SECURITY_PASSWORD_SALT                         # Only needed for
                                               # Flask-Security compatible hashing

USER_REQUIRE_INVITATION          = False       # Registration requires invitation
                                               # Not yet implemented
                                               # Requires USER_ENABLE_EMAIL=True

USER_RESET_PASSWORD_EXPIRATION   = 2*24*3600   # Reset password expiration in seconds
                                               # (2*24*3600 represents 2 days)

USER_SEND_PASSWORD_CHANGED_EMAIL = True        # Send registered email
                                               # Requires USER_ENABLE_EMAIL=True

USER_SEND_REGISTERED_EMAIL       = True        # Send registered email
                                               # Requires USER_ENABLE_EMAIL=True

USER_SEND_USERNAME_CHANGED_EMAIL = True        # Send registered email
                                               # Requires USER_ENABLE_EMAIL=True
```

```
USER_SHOW_USERNAME_EMAIL_DOES_NOT_EXIST = USER_ENABLE_REGISTRATION
                                                    # Show 'Username/Email does not exist'␣
↪error message
                                                    # instead of 'Incorrect Username/Email␣
↪and/or password'
```

7.10.3 Labels and Messages

The following can be customized by editing the English Babel translation file:

- Flash messages (one-time system messages)
- Form field labels
- Validation messages

See *Internationalization*

7.10.4 Emails

Emails are generated using Flask Jinja2 template files. Flask will first look for template files in the application's `templates` directory before looking in Flask-User's `templates` directory.

Emails can thus be customized by copying the built-in Email template files from the Flask-User directory to your application's directory and editing the new copy.

Flask-User typically installs in the `flask_user` sub-directory of the Python packages directory. The location of this directory depends on Python, virtualenv and pip and can be determined with the following command:

```
python -c "from distutils.sysconfig import get_python_lib; print get_python_lib();"
```

Let's assume that:

- The Python packages dir is: `~/.virtualenvs/ENVNAME/lib/python2.7/site-packages/`
- The Flask-User dir is: `~/.virtualenvs/ENVNAME/lib/python2.7/site-packages/flask_user/`
- Your app directory is: `~/path/to/YOURAPP/YOURAPP` (your application directory typically contains the 'static' and 'templates' sub-directories).

The built-in Email template files can be copied like so:

```
cd ~/path/to/YOURAPP/YOURAPP
mkdir -p templates/flask_user/emails
cp ~/.virtualenvs/ENVNAME/lib/python2.7/site-packages/flask_user/templates/flask_user/
↪emails/* templates/flask_user/emails/.
```

Flask-User currently offers the following email messages:

```
confirm_email      # Sent after a user submitted a registration form
                   # - Requires USER_ENABLE_EMAIL = True
                   # - Requires USER_ENABLE_CONFIRM_EMAIL = True

forgot_password    # Sent after a user submitted a forgot password form
                   # - Requires USER_ENABLE_EMAIL = True
                   # - Requires USER_ENABLE_FORGOT_PASSWORD = True
```

```
password_changed  # Sent after a user submitted a change password or reset password␣
↪form
                  # - Requires USER_ENABLE_EMAIL = True
                  # - Requires USER_ENABLE_CHANGE_PASSWORD = True
                  # - Requires USER_SEND_PASSWORD_CHANGED_EMAIL = True

registered        # Sent to users after they submitted a registration form
                  # - Requires USER_ENABLE_EMAIL = True
                  # - Requires USER_ENABLE_CONFIRM_EMAIL = False
                  # - Requires USER_SEND_REGISTERED_EMAIL = True

username_changed  # Sent after a user submitted a change username form
                  # - Requires USER_ENABLE_EMAIL = True
                  # - Requires USER_ENABLE_CHANGE_USERNAME = True
                  # - Requires USER_SEND_USERNAME_CHANGED_EMAIL = True
```

Each email type has three email template files. The 'registered' email for example has the following files:

```
templates/flask_user/emails/registered_subject.txt   # The email subject line
templates/flask_user/emails/registered_message.html  # The email message in HTML␣
↪format
templates/flask_user/emails/registered_message.txt   # The email message in Text␣
↪format
```

Each file is extended from the base template file:

```
templates/flask_user/emails/base_subject.txt
templates/flask_user/emails/base_message.html
templates/flask_user/emails/base_message.txt
```

The base template files are used to define email elements that are similar in all types of email messages.

If, for example, for every email you want to:
- Set the background color and padding,
- Start with a logo and salutation, and
- End with a signature,
you can define `templates/flask_user/emails/base_message.html` like so

```
<div style="background-color: #f4f2dd; padding: 10px;">
    <p><img src="http://example.com/static/images/email-logo.png"></p>
    <p>Dear Customer,</p>
    {% block message %}{% endblock %}
    <p>Sincerely,<br/>
    The Flask-User Team</p>
</div>
```

and define the confirmation specific messages in `templates/flask_user/emails/confirm_email_message.html` like so:

```
{% extends "flask_user/emails/base_message.html" %}

{% block message %}
<p>Thank you for registering with Flask-User.</p>
<p>Visit the link below to complete your registration:</p>
```

```
<p><a href="{{ confirm_email_link }}">Confirm your email address</a>.</p>
<p>If you did not initiate this registration, you may safely ignore this email.</p>
{% endblock %}
```

The email template files, along with available template variables listed below:

- **Template variables available in any email template**
  - `user_manager` - For example: `{% if user_manager.enable_confirm_email %}`
  - `user` - For example: `{{ user.email }}`
- **templates/flask_user/confirm_email_[subject.txt|message.html|message.txt]**
  - `confirm_email_link` - For example: `{{ confirm_email_link }}`
- **templates/flask_user/forgot_password_[subject.txt|message.html|message.txt]**
  - `reset_password_link` - For example: `{{ reset_password_link }}`
- **templates/flask_user/password_changed_[subject.txt|message.html|message.txt]**
  - n/a
- **templates/flask_user/registered_[subject.txt|message.html|message.txt]**
  - n/a
- **templates/flask_user/username_changed_[subject.txt|message.html|message.txt]**
  - n/a

If you need other email notifications, please enter a feature request to our Github issue tracker. Thank you.

7.10.5 Registration Form

We recommend asking for as little information as possible during user registration, and to only prompt new users for additional information *after* the registration process has been completed.

Some Websites, however, do want to ask for additional information in the registration form itself.

Flask-User (v0.4.5 and up) has the capability to store extra registration fields in the User or the UserProfile records.

Extra registration fields in the User model

Extra fields must be defined in the User model:

```
class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    is_enabled = db.Column(db.Boolean(), nullable=False, default=False)
    email = db.Column(db.String(255), nullable=False, default='')
    password = db.Column(db.String(255), nullable=False, default='')
    # Extra model fields
    first_name = db.Column(db.String(50), nullable=False, default='')
    last_name  = db.Column(db.String(50), nullable=False, default='')

    def is_active(self):
      return self.is_enabled

db_adapter = SQLAlchemyAdapter(db, UserClass=User)
```

A custom RegisterForm must be defined with field names **exactly matching** the names of the model fields:

```
class MyRegisterForm(RegisterForm):
    first_name = StringField('First name', validators=[DataRequired('First name is␣
↪required')])
    last_name  = StringField('Last name',  validators=[DataRequired('Last name is␣
↪required')])

user_manager = UserManager(db_adapter, app, register_form=MyRegisterForm)
```

A custom `templates/flask_user/register.html` file must be copied and defined with the extra fields. See *Form Templates*.

When a new user submits the Register form, Flask-User examines the field names of the form and the User model. For each matching field name, the form field value will be stored in the corresponding User field.

See Github repository; example_apps/register_form_app

Extra registration fields in UserProfile model

- Add extra fields to the User data model
- Extend a custom MyRegisterForm class from the built-in flask_user.forms.RegisterForm class. See *Form Classes*.
- Add extra fields to the form **using identical field names**.
- Specify your custom registration form: `user_manager = UserManager(db_adapter, app, register_form=MyRegisterForm)`
- Copy the built-in `templates/flask_user/register.html` to your application's templates/flask_user directory. See *Form Templates*.
- Add the extra form fields to register.html

7.10.6 Form Classes

Forms can be customized by sub-classing one of the following built-in Form classes:

```
flask_user.forms.AddEmailForm
flask_user.forms.ChangeUsernameForm
flask_user.forms.ChangePasswordForm
flask_user.forms.ForgotPasswordForm
flask_user.forms.LoginForm
flask_user.forms.RegisterForm
flask_user.forms.ResetPasswordForm
```

and specifying the custom form in the call to UserManager():

```
from flask_user.forms import RegisterForm

class MyRegisterForm(RegisterForm):
    first_name = StringField('First name')
    last_name = StringField('Last name')

user_manager = UserManager(db_adapter, app,
        register_form = MyRegisterForm)
```

See also *Form Templates*.

7.10.7 Form Templates

Forms are generated using Flask Jinja2 template files. Flask will first look for template files in the application's `templates` directory before looking in Flask-User's `templates` directory.

Forms can thus be customized by copying the built-in Form template files from the Flask-User directory to your application's directory and editing the new copy.

Flask-User typically installs in the `flask_user` sub-directory of the Python packages directory. The location of this directory depends on Python, virtualenv and pip and can be determined with the following command:

```
python -c "from distutils.sysconfig import get_python_lib; print get_python_lib();"
```

Let's assume that:

- The Python packages dir is: `~/.virtualenvs/ENVNAME/lib/python2.7/site-packages/`
- The Flask-User dir is: `~/.virtualenvs/ENVNAME/lib/python2.7/site-packages/flask_user/`
- Your app directory is: `~/path/to/YOURAPP/YOURAPP` (your application directory typically contains the 'static' and 'templates' sub-directories).

Forms can be customized by copying the form template files like so:

```
cd ~/path/to/YOURAPP/YOURAPP
mkdir -p templates/flask_user
cp ~/.virtualenvs/ENVNAME/lib/python2.7/site-packages/flask_user/templates/flask_user/
↪*.html templates/flask_user/.
```

and by editing the copies to your liking.

The following form template files resides in the `templates` directory and can be customized:

```
base.html                               # root template

flask_user/member_base.html             # extends base.html
flask_user/change_password.html         # extends flask_user/member_base.html
flask_user/change_username.html         # extends flask_user/member_base.html
flask_user/manage_emails.html           # extends flask_user/member_base.html
flask_user/user_profile.html            # extends flask_user/member_base.html

flask_user/public_base.html             # extends base.html
flask_user/forgot_password.html         # extends flask_user/public_base.html
flask_user/login.html                   # extends flask_user/public_base.html
flask_user/login_or_register.html       # extends flask_user/public_base.html
flask_user/register.html                # extends flask_user/public_base.html
flask_user/resend_confirm_email.html    # extends flask_user/public_base.html
flask_user/reset_password.html          # extends flask_user/public_base.html
```

If you'd like the Login form and the Register form to appear on one page, you can use the following application config settings:

```
# Place the Login form and the Register form on one page:
# Only works for Flask-User v0.4.9 and up
USER_LOGIN_TEMPLATE                     = 'flask_user/login_or_register.html'
USER_REGISTER_TEMPLATE                  = 'flask_user/login_or_register.html'
```

See also *Form Classes*.

7.10.8 Password and Username Validators

Flask-User comes standard with a password validator (at least 6 chars, 1 upper case letter, 1 lower case letter, 1 digit) and with a username validator (at least 3 characters in "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-._").

Custom validators can be specified by setting an attribute on the Flask-User's UserManager object:

```
from wtforms.validators import ValidationError

def my_password_validator(form, field):
    password = field.data
    if len(password) < 8:
        raise ValidationError(_('Password must have at least 8 characters'))

def my_username_validator(form, field):
    username = field.data
    if len(username) < 4:
        raise ValidationError(_('Username must be at least 4 characters long'))
    if not username.isalnum():
        raise ValidationError(_('Username may only contain letters and numbers'))

user_manager = UserManager(db_adapter,
        password_validator=my_password_validator,
        username_validator=my_username_validator)
user_manager.init_app(app)
```

7.10.9 Password hashing

To hash a password, Flask-User:

- calls `user_manager.hash_password()`,
- which calls `user_manager.password_crypt_context`,
- which is initialized to `CryptContext(schemes=[app.config['USER_PASSWORD_HASH']])`,
- where `USER_PASSWORD_HASH = 'bcrypt'`.

See http://pythonhosted.org/passlib/new_app_quickstart.html

Developers can customize the password hashing in the following ways:

By changing an application config setting:

```
USER_PASSWORD_HASH = 'sha512_crypt'
```

By changing the crypt_context:

```
my_password_crypt_context = CryptContext(
        schemes=['bcrypt', 'sha512_crypt', 'pbkdf2_sha512', 'plaintext'])
user_manager = UserManager(db_adapter, app,
        password_crypt_context=my_password_crypt_context)
```

By sub-classing hash_password():

```
class MyUserManager(UserManager):
    def hash_password(self, password):
        return self.password
```

```
    def verify_password(self, password, user)
        return self.hash_password(password) == self.get_password(user)
```

Backward compatibility with Flask-Security

Flask-Security performs a SHA512 HMAC prior to calling passlib. To continue using passwords that have been generated with Flask-Security, add the following settings to your application config:

```
# Keep the following Flaks and Flask-Security settings the same
SECRET_KEY = ...
SECURITY_PASSWORD_HASH = ...
SECURITY_PASSWORD_SALT = ...

# Set Flask-Security backward compatibility mode
USER_PASSWORD_HASH_MODE = 'Flask-Security'
USER_PASSWORD_HASH      = SECURITY_PASSWORD_HASH
USER_PASSWORD_SALT      = SECURITY_PASSWORD_SALT
```

7.10.10 View Functions

The built-in View Functions contain considerable business logic, so we recommend first trying the approach of *Form Templates* before making use of customized View Functions.

Custom view functions are specified by setting an attribute on the Flask-User's UserManager object:

```
# View functions
user_manager = UserManager(db_adapter,
        change_password_view_function      = my_view_function1,
        change_username_view_function      = my_view_function2,
        confirm_email_view_function        = my_view_function3,
        email_action_view_function         = my_view_function4,
        forgot_password_view_function      = my_view_function5,
        login_view_function                = my_view_function6,
        logout_view_function               = my_view_function7,
        manage_emails_view_function        = my_view_function8,
        register_view_function             = my_view_function9,
        resend_confirm_email_view_function = my_view_function10,
        reset_password_view_function       = my_view_function11,
        )
user_manager.init_app(app)
```

7.10.11 URLs

URLs can be customized through the application's config

```
# URLs                          # Default
USER_CHANGE_PASSWORD_URL      = '/user/change-password'
USER_CHANGE_USERNAME_URL      = '/user/change-username'
USER_CONFIRM_EMAIL_URL        = '/user/confirm-email/<token>'
USER_EMAIL_ACTION_URL         = '/user/email/<id>/<action>'     # v0.5.1 and up
USER_FORGOT_PASSWORD_URL      = '/user/forgot-password'
USER_INVITE_URL               = '/user/invite'                  # v0.6.2 and up
USER_LOGIN_URL                = '/user/login'
USER_LOGOUT_URL               = '/user/logout'
```

```
USER_MANAGE_EMAILS_URL        = '/user/manage-emails'
USER_REGISTER_URL             = '/user/register'
USER_RESEND_CONFIRM_EMAIL_URL = '/user/resend-confirm-email'    # v0.5.0 and up
USER_RESET_PASSWORD_URL       = '/user/reset-password/<token>'
```

7.10.12 Endpoints

Endpoints can be customized through the application's config

```
# Endpoints are converted to URLs using url_for()
# The empty endpoint ('') will be mapped to the root URL ('/')
USER_AFTER_CHANGE_PASSWORD_ENDPOINT      = ''              # v0.5.3 and up
USER_AFTER_CHANGE_USERNAME_ENDPOINT      = ''              # v0.5.3 and up
USER_AFTER_CONFIRM_ENDPOINT              = ''              # v0.5.3 and up
USER_AFTER_FORGOT_PASSWORD_ENDPOINT      = ''              # v0.5.3 and up
USER_AFTER_LOGIN_ENDPOINT                = ''              # v0.5.3 and up
USER_AFTER_LOGOUT_ENDPOINT               = 'user.login'    # v0.5.3 and up
USER_AFTER_REGISTER_ENDPOINT             = ''              # v0.5.3 and up
USER_AFTER_RESEND_CONFIRM_EMAIL_ENDPOINT = ''              # v0.5.3 and up
USER_AFTER_RESET_PASSWORD_ENDPOINT       = ''              # v0.6 and up
USER_INVITE_ENDPOINT                     = ''              # v0.6.2 and up

# Users with an unconfirmed email trying to access a view that has been
# decorated with @confirm_email_required will be redirected to this endpoint
USER_UNCONFIRMED_EMAIL_ENDPOINT          = 'user.login'    # v0.6 and up

# Unauthenticated users trying to access a view that has been decorated
# with @login_required or @roles_required will be redirected to this endpoint
USER_UNAUTHENTICATED_ENDPOINT            = 'user.login'    # v0.5.3 and up

# Unauthorized users trying to access a view that has been decorated
# with @roles_required will be redirected to this endpoint
USER_UNAUTHORIZED_ENDPOINT               = ''              # v0.5.3 and up
```

7.10.13 Email Template filenames

Email template filenames can be customized through the application's config

```
# Email template files                  # Defaults
USER_CONFIRM_EMAIL_EMAIL_TEMPLATE       = 'flask_user/emails/confirm_email'
USER_FORGOT_PASSWORD_EMAIL_TEMPLATE     = 'flask_user/emails/forgot_password'
USER_INVITE_EMAIL_TEMPLATE              = 'flask_user/emails/invite'
USER_PASSWORD_CHANGED_EMAIL_TEMPLATE    = 'flask_user/emails/password_changed'
USER_REGISTERED_EMAIL_TEMPLATE          = 'flask_user/emails/registered'
USER_USERNAME_CHANGED_EMAIL_TEMPLATE    = 'flask_user/emails/username_changed'

# These settings correspond to the start of three template files:
# SOMETHING_subject.txt   # Email subject
# SOMETHING_message.html  # Email message in HTML format
# SOMETHING_message.txt   # Email message in Text format
```

These path settings are relative to the application's `templates` directory.

7.10.14 Form Template filenames

Form template filenames can be customized through the application's config

```
# Form template files                    # Defaults
USER_CHANGE_PASSWORD_TEMPLATE            = 'flask_user/change_password.html'
USER_CHANGE_USERNAME_TEMPLATE            = 'flask_user/change_username.html'
USER_FORGOT_PASSWORD_TEMPLATE            = 'flask_user/forgot_password.html'
USER_INVITE_TEMPLATE                     = 'flask_user/invite.html'                #␣
↪v0.6.2 and up
USER_INVITE_ACCEPT_TEMPLATE              = 'flask_user/register.html'              #␣
↪v0.6.2 and up
USER_LOGIN_TEMPLATE                      = 'flask_user/login.html'
USER_MANAGE_EMAILS_TEMPLATE              = 'flask_user/manage_emails.html'         #␣
↪v0.5.1 and up
USER_REGISTER_TEMPLATE                   = 'flask_user/register.html'
USER_RESEND_CONFIRM_EMAIL_TEMPLATE       = 'flask_user/resend_confirm_email.html'  #␣
↪v0.5.0 and up
USER_RESET_PASSWORD_TEMPLATE             = 'flask_user/reset_password.html'

# Place the Login form and the Register form on one page:
# Only works for Flask-User v0.4.9 and up
USER_LOGIN_TEMPLATE                      = 'flask_user/login_or_register.html'
USER_REGISTER_TEMPLATE                   = 'flask_user/login_or_register.html'
```

These path settings are relative to the application's `templates` directory.

7.10.15 Token Generation

To be documented.

7.11 Signals (event hooking)

Flask Applications that want to be kept informed about certain actions that took place in the underlying Flask-User extensions can do so by subscribing 'signals' to receive event notification.

Flask-User defines the following signals:

```
# Signal                    # Sent when ...
user_changed_password       # a user changed their password
user_changed_username       # a user changed their username
user_confirmed_email        # a user confirmed their email
user_forgot_password        # a user submitted a reset password request
user_logged_in              # a user logged in
user_logged_out             # a user logged out
user_registered             # a user registered for a new account
user_reset_password         # a user reset their password (forgot password)
```

See the http://flask.pocoo.org/docs/signals/

7.11.1 Installing Blinker – REQUIRED!

NB: Flask-User relies on Flask signals, which relies on the 'blinker' package. Event notification WILL NOT WORK without first installing the 'blinker' package.

```
pip install blinker
```

See http://pythonhosted.org/blinker/

7.11.2 Subscribing to Signals

AFTER BLINKER HAS BEEN INSTALLED, An application can receive event notifications by using the event signal's `connect_via()` decorator:

```
from flask_user import user_logged_in

@user_logged_in.connect_via(app)
def _after_login_hook(sender, user, **extra):
    sender.logger.info('user logged in')
```

For all Flask-User event signals:
- `sender` points to the app, and
- `user` points to the user that is associated with this event.

See Subscribing to signals

7.11.3 Troubleshooting

If the code looks right, but the tracking functions are not called, make sure to check to see if the 'blinker' package has been installed (analyze the output of `pip freeze`).

7.12 Recipes

Here we explain the use of Flask-User through code recipes.

7.12.1 Login Form and Register Form on one page

Some websites may prefer to show the login form and the register form on one page.

Flask-User (v0.4.9 and up) ships with a `login_or_register.html` form template which requires the following application config settings:

- `USER_LOGIN_TEMPLATE='flask_user/login_or_register.html'`
- `USER_REGISTER_TEMPLATE='flask_user/login_or_register.html'`

This would accomplish the following:

- The `/user/login` and `user/register` URLs will now render `login_or_register.html`.
- `login_or_register.html` now displays a Login form and a Register form.
- The Login button will post to `/user/login`
- The Register button will post to `/user/register`

7.12.2 After registration hook

Some applications require code to execute just after a new user registered for a new account. This can be achieved by subscribing to the `user_registered` signal as follows:

```
from flask_user.signals import user_registered

@user_registered.connect_via(app)
def _after_registration_hook(sender, user, **extra):
    sender.logger.info('user registered')
```

See also: *Signals (event hooking)*

7.12.3 Hashing Passwords

If you want to populate your database with User records with hashed passwords use `user_manager.hash_password()`:

```
user = User(
        email='user1@example.com',
        password=user_manager.hash_password('Password1'),
        )
db.session.add(user)
db.session.commit()
```

You can verify a password with `user_manager.verify_password()`:

```
does_match = user_manager.verify_password(password, user)
```

7.12.4 Account Tracking

Flask-User deliberately stayed away from implementing account tracking features because:

- What to track is often customer specific
- Where to store it is often customer specific
- Custom tracking is easy to implement using signals

Here's an example of tracking login_count and last_login_ip:

```
# This code has not been tested

from flask import request
from flask_user.signals import user_logged_in

@user_logged_in.connect_via(app)
def _track_logins(sender, user, **extra):
    user.login_count += 1
    user.last_login_ip = request.remote_addr
    db.session.commit()
```

7.12.5 Up Next

Customization

7.13 Internationalization

Flask-User allows the developer to translate their user account management forms into other languages. This allows us to:

- Customize built-in English text to custom English text
- Translate built-in English text into another language

Flask-User ships with the following languages:

- de - German (v0.6.9+)
- en - English (v0.1+)
- es - Spanish (v0.6.10+)
- fa - Farsi (v0.6.9+)
- fi - Finnish (v0.6.9+)
- fr - French (v0.6.1+)
- it - Italian (v0.6.9+)
- nl - Dutch (v0.1+)
- ru - Russian (v0.6.12+)
- sv - Swedish (v0.6.9+)
- tr - Turkish (v0.6.9+)
- zh - Chinese (Simplified) (v0.6.1+)

7.13.1 REQUIRED: Installing Flask-Babel

Flask-User relies on the Flask-Babel package to translate the account management forms. Without Flask-Babel installed, these forms WILL NOT BE translated.

Install Flask-Babel with

```
pip install Flask-Babel
```

7.13.2 REQUIRED: Initializing Flask-Babel

Flask-Babel must be initialized just after the Flask application has been initialized and after the application configuration has been read:

```
from flask_babel import Babel

...

app = Flask(__name__)
app.config.from_object('app.config.settings')

...

# Initialize Flask-Babel
babel = Babel(app)
```

```
# Use the browser's language preferences to select an available translation
@babel.localeselector
def get_locale():
    translations = [str(translation) for translation in babel.list_translations()]
    return request.accept_languages.best_match(translations)
```

7.13.3 How Flask-Babel works

- Flask-Babel calls a translatable string a 'Message'.
- Messages are marked with `gettext('string')`, `_('string')` or `{%trans%}string{%endtrans%}`.
- `pybabel extract` extracts Messages into a `.pot` template file.
- `pybabel update` converts the `.pot` template file into a language specific `.po` translations file.
  - A `.po` file contains `msgid/msgstr` (key/value) pairs for each translatable string
  - The `msgid` represents the built-in English message (key)
  - The `msgstr` represents the translated message (value)
- Translators edit the `msgstr` portion of the `.po` translation files.
- `pybabel compile` compiles human readable `.po` translation files into machine readable `.mo` complied translation files.
- At runtime:
  - the browser specifies the preferred language code (`'en'` for English, `'es'` for Spanish, `'nl'` for Dutch, etc.).
  - The web server loads the corresponding compiled translation file. For example: `app/translations/en/LC_MESSAGES/flask_user.mo`.
  - gettext('string') looks up the `msgid=='string'` entry in the `.mo` file.
  - If a `msgstr` is defined: it will return the translated message, if not: it will return the built-in English message.

7.13.4 Preparing for translation

We need to copy the Flask-User `translations` directory to your application directory.

Flask-User typically installs in the `flask_user` sub-directory of the Python packages directory. The location of this directory depends on Python, virtualenv and pip and can be determined with the following command:

```
python -c "from distutils.sysconfig import get_python_lib; print get_python_lib();"
```

Let's assume that:

- The Python packages dir is: `~/.virtualenvs/ENVNAME/lib/python2.7/site-packages/`
- The Flask-User dir is: `~/.virtualenvs/ENVNAME/lib/python2.7/site-packages/flask_user/`
- Your app directory is: `~/path/to/YOURAPP/YOURAPP` (your application directory typically contains the 'static' and 'templates' sub-directories).

Copy the `translations` directory from flask_user to your application directory:

```
cd ~/path/to/YOURAPP/YOURAPP
cp -r ~/.virtualenvs/YOURENV/lib/python2.7/site-packages/flask_user/translations .
```

To edit the translations file. We recommend using a translation editor such as Poedit

Download poedit

If you run Mac OS 10.6 or older, you'll need to download version 1.5 from here.

7.13.5 Customizing Messages

Customization is achieved by 'translating' built-in English messages to the custom English messages of your choice. The two-letter language code for English is 'en'.

Customize .po file

Edit `translations/en/LC_MESSAGES/flask_user.po`

We recommend using a translation program such as `poedit`. If you want to edit the .po file manually make sure to leave `msgid` strings as-is and to only edit the `msgstr` strings.

Customize only those message that need to be different from the built-in message. Entries with an empty `msgstr` will display the built-in `msgid`.

Safe the .po file when you're done.

Compile .mo file

Compile a .mo compiled translation file from a .po translation file like so:

```
cd ~/path/to/YOURAPP/YOURAPP
pybabel compile -d translations -D flask_user -f
```

Verify

`.mo` files are read when your web server starts, so make sure to restart your web server.

Point your browser to your app and your custom messages should appear.

7.13.6 Translating Messages

Determine the language code

The ISO 639-1 standard defines two-letter codes for languages. Find your two-letter codes here.

This document assumes that you chose 'es' for Spanish.

Create .po file (One-time only)

.po translation files are generated from .pot template files using `pybabel init`.

```
cd ~/path/to/YOURAPP/YOURAPP
pybabel init -d translations -l es -D flask_user -i translations/flask_user.pot
```

Update .po files

The `pybabel init` command will over-write any existing .po files.

If you need to update the .po files (for example if a new Flask-User version releases a new flask_user.pot template file), you can use the `pybabel update` command to keep your prior translations.

```
cd ~/path/to/YOURAPP/YOURAPP
pybabel update -d translations -l es -D flask_user -i translations/flask_user.pot
```

Translate .po file

Edit `translations/es/LC_MESSAGES/flask_user.po`

We recommend using a translation program such as `poedit`. If you want to edit the .po file manually make sure to leave `msgid` strings as-is and to only edit the `msgstr` strings.

Safe the .po file when you're done.

Compile .mo file

Compile a .mo compiled translation file from a .po translation file like so:

```
cd ~/path/to/YOURAPP/YOURAPP
pybabel compile -d translations -D flask_user -f
```

Verify

Make sure you have this code somewhere:

```
@babel.localeselector
def get_locale():
    translations = [str(translation) for translation in babel.list_translations()]
    return request.accept_languages.best_match(translations)
```

Make sure to prioritize the Spanish language in your browser settings.

`.mo` files are read when your web server starts, so make sure to restart your web server.

Point your browser to your app and your translated messages should appear.

7.13.7 Troubleshooting

If the code looks right, but the account management forms are not being translated:

- Check to see if the 'Flask-Babel' package has been installed (try using `pip freeze`).
- Check to see if the browser has been configured to prefer the language you are testing.
- Check to see if the 'translations/' directory is in the right place.

7.14 F.A.Q.

Q: Can I see a preview?
A: Yes you can: Flask-User Demo

Q: What are the differences between Flask-User and Flask-Security?
A: The main reason why I wrote Flask-User was because I found it difficult to customize Flask-Security messages and functionality (in 2013) and because it didn't offer Username login, multiple emails per user, and Internationalization.

Flask-Security has been around since at least March 2012 and additionally offers Json/AJAX, MongoDB, Peewee, and Basic HTTP Authentication.

Flask-User has been designed with *Full customization* in mind and additionally offers Username login and Internationalization. It exists since December 2013 and contains 661 statements with a 98% test coverage.

Q: Can users login with usernames and email addresses?
A: Yes. Flask-User can be configured to enable usernames, email addresses or both. If both are enabled, users can log in with either their username or their email address.

Q: Does Flask-User work with existing hashed passwords?
A: Yes. It supports the following:
- passwords hashed by any `passlib` hashing algorithm (via a config setting)
- passwords hashed by Flask-Security (via a config setting)
- custom password hashes (via custom functions)

Q: What databases does Flask-User support?
A: Any database that SQLAlchemy supports (via SqlAlchemyAdapter)
and other databases (via custom DBAdapters)

Flask-User shields itself from database operations through a DBAdapter. It ships with a SQLAlchemyAdapter, but the API is very simple, so other adapters can be easily added. See *SQLAlchemyAdapter()*.

7.15 Flask-User API

- *Config Settings*
- *SQLAlchemyAdapter()*
- *UserManager*
- *Template variables*
- *Template functions*
- *Signals*

7.15.1 Config Settings

```
# Features                      # Default   # Description
USER_ENABLE_CHANGE_PASSWORD     = True      # Allow users to change their password

USER_ENABLE_CHANGE_USERNAME     = True      # Allow users to change their username
                                            # Requires USER_ENABLE_USERNAME=True

USER_ENABLE_CONFIRM_EMAIL       = True      # Force users to confirm their email
                                            # Requires USER_ENABLE_EMAIL=True

USER_ENABLE_FORGOT_PASSWORD     = True      # Allow users to reset their passwords
                                            # Requires USER_ENABLE_EMAIL=True
```

```
USER_ENABLE_LOGIN_WITHOUT_CONFIRM_EMAIL = False
                                        # Allow users to login without a
                                        # confirmed email address
                                        # Protect views using @confirm_email_
↪required

USER_ENABLE_EMAIL               = True      # Register with Email
                                            # Requires USER_ENABLE_REGISTRATION=True

USER_ENABLE_MULTIPLE_EMAILS     = False     # Users may register multiple emails
                                            # Requires USER_ENABLE_EMAIL=True

USER_ENABLE_REGISTRATION        = True      # Allow new users to register

USER_ENABLE_RETYPE_PASSWORD     = True      # Prompt for `retype password` in:
                                            #   - registration form,
                                            #   - change password form, and
                                            #   - reset password forms.

USER_ENABLE_USERNAME            = True      # Register and Login with username
```

```
# Settings                      # Default     # Description
USER_APP_NAME                   = 'AppName'   # Used by email templates

USER_AUTO_LOGIN                 = True

USER_AUTO_LOGIN_AFTER_CONFIRM   = USER_AUTO_LOGIN

USER_AUTO_LOGIN_AFTER_REGISTER  = USER_AUTO_LOGIN

USER_AUTO_LOGIN_AFTER_RESET_PASSWORD = USER_AUTO_LOGIN

USER_AUTO_LOGIN_AT_LOGIN        = USER_AUTO_LOGIN

USER_CONFIRM_EMAIL_EXPIRATION   = 2*24*3600   # Confirmation expiration in seconds
                                              # (2*24*3600 represents 2 days)

USER_INVITE_EXPIRATION          = 90*24*3600  # Invitation expiration in seconds
                                              # (90*24*3600 represents 90 days)
                                              # v0.6.2 and up

USER_PASSWORD_HASH              = 'bcrypt'    # Any passlib crypt algorithm

USER_PASSWORD_HASH_MODE         = 'passlib'   # Set to 'Flask-Security' for
                                              # Flask-Security compatible hashing

SECURITY_PASSWORD_SALT                        # Only needed for
                                              # Flask-Security compatible hashing

USER_REQUIRE_INVITATION         = False       # Registration requires invitation
                                              # Not yet implemented
                                              # Requires USER_ENABLE_EMAIL=True

USER_RESET_PASSWORD_EXPIRATION  = 2*24*3600   # Reset password expiration in seconds
                                              # (2*24*3600 represents 2 days)

USER_SEND_PASSWORD_CHANGED_EMAIL = True       # Send registered email
```

```
                                            # Requires USER_ENABLE_EMAIL=True

USER_SEND_REGISTERED_EMAIL       = True     # Send registered email
                                            # Requires USER_ENABLE_EMAIL=True

USER_SEND_USERNAME_CHANGED_EMAIL = True     # Send registered email
                                            # Requires USER_ENABLE_EMAIL=True

USER_SHOW_USERNAME_EMAIL_DOES_NOT_EXIST = USER_ENABLE_REGISTRATION
                                            # Show 'Username/Email does not exist'␣
↪error message
                                            # instead of 'Incorrect Username/Email␣
↪and/or password'
```

```
# URLs                          # Default
USER_CHANGE_PASSWORD_URL      = '/user/change-password'
USER_CHANGE_USERNAME_URL      = '/user/change-username'
USER_CONFIRM_EMAIL_URL        = '/user/confirm-email/<token>'
USER_EMAIL_ACTION_URL         = '/user/email/<id>/<action>'     # v0.5.1 and up
USER_FORGOT_PASSWORD_URL      = '/user/forgot-password'
USER_INVITE_URL               = '/user/invite'                  # v0.6.2 and up
USER_LOGIN_URL                = '/user/login'
USER_LOGOUT_URL               = '/user/logout'
USER_MANAGE_EMAILS_URL        = '/user/manage-emails'
USER_REGISTER_URL             = '/user/register'
USER_RESEND_CONFIRM_EMAIL_URL = '/user/resend-confirm-email'    # v0.5.0 and up
USER_RESET_PASSWORD_URL       = '/user/reset-password/<token>'
```

```
# Endpoints are converted to URLs using url_for()
# The empty endpoint ('') will be mapped to the root URL ('/')
USER_AFTER_CHANGE_PASSWORD_ENDPOINT      = ''              # v0.5.3 and up
USER_AFTER_CHANGE_USERNAME_ENDPOINT      = ''              # v0.5.3 and up
USER_AFTER_CONFIRM_ENDPOINT              = ''              # v0.5.3 and up
USER_AFTER_FORGOT_PASSWORD_ENDPOINT      = ''              # v0.5.3 and up
USER_AFTER_LOGIN_ENDPOINT                = ''              # v0.5.3 and up
USER_AFTER_LOGOUT_ENDPOINT               = 'user.login'    # v0.5.3 and up
USER_AFTER_REGISTER_ENDPOINT             = ''              # v0.5.3 and up
USER_AFTER_RESEND_CONFIRM_EMAIL_ENDPOINT = ''              # v0.5.3 and up
USER_AFTER_RESET_PASSWORD_ENDPOINT       = ''              # v0.6 and up
USER_INVITE_ENDPOINT                     = ''              # v0.6.2 and up

# Users with an unconfirmed email trying to access a view that has been
# decorated with @confirm_email_required will be redirected to this endpoint
USER_UNCONFIRMED_EMAIL_ENDPOINT          = 'user.login'    # v0.6 and up

# Unauthenticated users trying to access a view that has been decorated
# with @login_required or @roles_required will be redirected to this endpoint
USER_UNAUTHENTICATED_ENDPOINT            = 'user.login'    # v0.5.3 and up

# Unauthorized users trying to access a view that has been decorated
# with @roles_required will be redirected to this endpoint
USER_UNAUTHORIZED_ENDPOINT               = ''              # v0.5.3 and up
```

```
# Email template files                  # Defaults
USER_CONFIRM_EMAIL_EMAIL_TEMPLATE       = 'flask_user/emails/confirm_email'
USER_FORGOT_PASSWORD_EMAIL_TEMPLATE     = 'flask_user/emails/forgot_password'
USER_INVITE_EMAIL_TEMPLATE              = 'flask_user/emails/invite'
```

```
USER_PASSWORD_CHANGED_EMAIL_TEMPLATE    = 'flask_user/emails/password_changed'
USER_REGISTERED_EMAIL_TEMPLATE          = 'flask_user/emails/registered'
USER_USERNAME_CHANGED_EMAIL_TEMPLATE    = 'flask_user/emails/username_changed'

# These settings correspond to the start of three template files:
# SOMETHING_subject.txt    # Email subject
# SOMETHING_message.html   # Email message in HTML format
# SOMETHING_message.txt    # Email message in Text format
```

```
# Form template files                   # Defaults
USER_CHANGE_PASSWORD_TEMPLATE           = 'flask_user/change_password.html'
USER_CHANGE_USERNAME_TEMPLATE           = 'flask_user/change_username.html'
USER_FORGOT_PASSWORD_TEMPLATE           = 'flask_user/forgot_password.html'
USER_INVITE_TEMPLATE                    = 'flask_user/invite.html'                  #␣
↪v0.6.2 and up
USER_INVITE_ACCEPT_TEMPLATE             = 'flask_user/register.html'                #␣
↪v0.6.2 and up
USER_LOGIN_TEMPLATE                     = 'flask_user/login.html'
USER_MANAGE_EMAILS_TEMPLATE             = 'flask_user/manage_emails.html'           #␣
↪v0.5.1 and up
USER_REGISTER_TEMPLATE                  = 'flask_user/register.html'
USER_RESEND_CONFIRM_EMAIL_TEMPLATE      = 'flask_user/resend_confirm_email.html'    #␣
↪v0.5.0 and up
USER_RESET_PASSWORD_TEMPLATE            = 'flask_user/reset_password.html'

# Place the Login form and the Register form on one page:
# Only works for Flask-User v0.4.9 and up
USER_LOGIN_TEMPLATE                     = 'flask_user/login_or_register.html'
USER_REGISTER_TEMPLATE                  = 'flask_user/login_or_register.html'
```

7.15.2 SQLAlchemyAdapter()

Flask-User shields itself from database operations through a DBAdapter. It ships with a SQLAlchemyAdapter, but the API is very simple, so other adapters can be easily added.

```
class SQLAlchemyAdapter(DBAdapter):
    """ This object shields Flask-User from database specific functions. """

    def get_object(self, ObjectClass, pk):
        """ Retrieve one object specified by the primary key 'pk' """

    def find_all_objects(self, ObjectClass, **kwargs):
        """ Retrieve all objects matching the case sensitive filters in 'kwargs'. """

    def find_first_object(self, ObjectClass, **kwargs):
        """ Retrieve the first object matching the case sensitive filters in 'kwargs'.
↪ """

    def ifind_first_object(self, ObjectClass, **kwargs):
        """ Retrieve the first object matching the case insensitive filters in 'kwargs
↪'. """

    def add_object(self, ObjectClass, **kwargs):
        """ Add an object with fields and values specified in kwargs. """

    def update_object(self, object, **kwargs):
```

```
        """ Update an object with fields and values specified in kwargs. """

    def delete_object(self, object):
        """ Delete an object. """

    def commit(self):
        """ Commit an Add, Update or Delete. """
```

7.15.3 Template variables

The following template variables are available for use in email and form templates:

```
user_manager        # points to the UserManager object
```

7.15.4 Template functions

The following template functions are available for use in email and form templates:

```
# Function                          Setting                     # Default
url_for('user.change_password')     USER_CHANGE_PASSWORD_URL      = '/user/change-
↪password'
url_for('user.change_username')     USER_CHANGE_USERNAME_URL      = '/user/change-
↪username'
url_for('user.confirm_email')       USER_CONFIRM_EMAIL_URL        = '/user/confirm-
↪email/<token>'
url_for('user.email_action')        USER_EMAIL_ACTION_URL         = '/user/email/<id>
↪/<action>'    # v0.5.1 and up
url_for('user.forgot_password')     USER_FORGOT_PASSWORD_URL      = '/user/forgot-
↪password'
url_for('user.login')               USER_LOGIN_URL                = '/user/sign-in'
url_for('user.logout')              USER_LOGOUT_URL               = '/user/sign-out'
url_for('user.register')            USER_REGISTER_URL             = '/user/register'
url_for('user.resend_confirm_email') USER_RESEND_CONFIRM_EMAIL_URL = '/user/resend-
↪confirm-email'   # v0.5.0 and up
url_for('user.reset_password')      USER_RESET_PASSWORD_URL       = '/user/reset-
↪password/<token>'
url_for('user.profile')             USER_PROFILE_URL              = '/user/profile'␣
↪              # v0.5.5 and up
```

7.15.5 UserManager

UserManager()

```
user_manager = UserManager(
        db_adapter,                     # typically from SQLAlchemyAdapter()
        app = None,                     # typically from Flask() or None

        # Forms
        add_email_form                  = forms.AddEmailForm,
        change_username_form            = forms.ChangeUsernameForm,
        forgot_password_form            = forms.ForgotPasswordForm,
        login_form                      = forms.LoginForm,
```

```
        register_form                   = forms.RegisterForm,
        resend_confirm_email_form       = forms.ResendConfirmEmailForm,
        reset_password_form             = forms.ResetPasswordForm,

        # Validators
        username_validator              = forms.username_validator,
        password_validator              = forms.password_validator,

        # View functions
        change_password_view_function   = views.change_password,
        change_username_view_function   = views.change_username,
        confirm_email_view_function     = views.confirm_email,
        email_action_view_function      = views.email_action,
        forgot_password_view_function   = views.forgot_password,
        login_view_function             = views.login,
        logout_view_function            = views.logout,
        manage_emails_view_function     = views.manage_emails,
        register_view_function          = views.register,
        resend_confirm_email_view_function = views.resend_confirm_email_view_function,
        reset_password_view_function    = views.reset_password,
        user_profile_view_function      = views.user_profile,
        unauthenticated_view_function   = views.unauthenticated,
        unauthorized_view_function      = views.unauthorized,

        # Miscellaneous
        login_manager                   = LoginManager(),
        password_crypt_context          = None,
        send_email_function             = emails.send_email,
        make_safe_url_function          = views.make_safe_url,
        token_manager                   = tokens.TokenManager(),
        )
```

Typical use:

```
app = Flask(__name__)
db = SQLAlchemy(app)
db_adapter = SQLAlchemyAdapter(db, User)
user_manager = UserManager(db_adapter, app,
        register_form=my_register_form,
        register_view_function=my_register_view_function)
```

Work in progress. See *Basic App* for now.

init_app()

init_app() is used by Application Factories to bind the UserManager to a specific app.

typical use:

```
db = SQLAlchemy()
db_adapter = SQLAlchemyAdapter(db, User)
user_manager = UserManager(db_adapter)

def create_app():
    app = Flask(__name__)
    db.init_app(app)
    user_manager.init_app(app)
```

Work in progress. See *Basic App* for now.

hash_password()

```
user_manager.hash_password(password)
# Returns hashed 'password' using the configured password hash
# Config settings: USER_PASSWORD_HASH_MODE = 'passlib'
#                  USER_PASSWORD_HASH      = 'bcrypt'
#                  USER_PASSWORD_SALT      = SECRET_KEY
```

verify_password()

```
user_manager.verify_password(password, user)
# Returns True if 'password' matches the user's 'hashed password'
# Returns False otherwise.
```

7.15.6 Signals

```
# Signal                    # Sent when ...
user_changed_password       # a user changed their password
user_changed_username       # a user changed their username
user_confirmed_email        # a user confirmed their email
user_forgot_password        # a user submitted a reset password request
user_logged_in              # a user logged in
user_logged_out             # a user logged out
user_registered             # a user registered for a new account
user_reset_password         # a user reset their password (forgot password)
```

CHAPTER 8

Revision History

- v0.6.21 - Properly send `user_reset_password` signal
- v0.6.20 - Fixed make_url_safe() with queries and fragments
- v0.6.19 - Fixed install problem
- v0.6.14-18 - (Skipped)
- v0.6.13 - Added make_safe_url() to prevent cross-domain redirections.
- v0.6.12 - Added Russian translation.
- v0.6.11 - (Skipped)
- v0.6.10 - Added Spanish translation.
- **v0.6.9 - Added support for Flask-Login v0.4+.** Replaced pycrypto with pycryptodome. Added Farsi, Italian and Turkish translations.
- v0.6.8 - Added support for Flask-Login v0.3+
- v0.6.7 - Uses Python package bcrypt instead of py-bcrypt.
- v0.6.6 - Forgot password form now honors USER_SHOW_USERNAME_OR_EMAIL_DOES_NOT_EXIST setting.
- v0.6.5 - Added USER_SHOW_USERNAME_OR_EMAIL_DOES_NOT_EXIST setting.
- v0.6.4 - Moved custom params from __init__() to init_app(). Added send_reset_password_email().
- v0.6.3 - Fix for Python 3.4 and signals. Added UserMixin.has_role() and @roles_accepted().
- v0.6.2 - Added support for invitation-only registrations.
- v0.6.1 - Added Chinese (Simplified) and French translations‘.
- v0.6 - Changed User/UserProfile DataModels into UserAuth/User DataModels.

```
v0.6 Is backwards compatible with v0.5 but UserProfileClass will be deprecated
in the future. See the 'Data Models' section in this documentation.
```

- v0.5.5 Added user_profile view. Cleaned up base template. Support for UserProfile.roles.
- v0.5.4 Decoupled Flask-User from Flask-Babel and speaklater
- v0.5.3
  - Added remember-me feature.
  - Added support for a primary key name other than 'id'.
  - Added USER_AUTO_LOGIN_... settings.
  - Added USER_AFTER_..._ENDPOINT settings.
  - Cleaned up email templates.

```
v0.5.3 API changes
The 'confirm_email' emails are now sent only after a resend confirm email request.
The 'registered' email is now sent after registration, whether
USER_ENABLE_CONFIRM_EMAIL is True or False.

(Previously, the 'confirm_email' email was sent after registration
and after a resend confirm email request, and the 'registered' email was sent only
after registration and when USER_ENABLE_CONFIRM_EMAIL was False)
```

- v0.5.2 Added USER_AUTO_LOGIN setting.
- v0.5.1 Added Support for multiple emails per user.
- v0.5.0 Added `resend_confirm_email`.
- v0.4.9 Added `login_or_register.html`. Cleaned up example_apps.
- v0.4.8 Removed the need for app.mail, app.babel, app.db and app.User properties.
- v0.4.7 Added 'confirm_email', 'password_changed' and 'username_changed' emails.

```
v0.4.7 API changes
The 'registered' email was split into 'confirm_email' and 'registered' emails.
If you've customized 'templates/flask_user/email/registered_*':
rename the 'registered_*' files into 'confirm_email_*'.
```

- v0.4.6 Added 'next' query parameter to confirm_email link
- v0.4.5 Save custom Register fields to User or UserProfile

```
v0.4.5 API changes
db_adapter.add_object()/update_object()/delete_object() now require a separate
call to db_adapter.commit()
```

- v0.4.4 Enhancements and Fixes: Github issues #6, #7 & #8
- v0.4.3 base.html, flask_user/public_base.html, flask_user/member_base.html. Cleanup. Reduced package size from 83KB to 30KB.

```
v0.4.3 API changes
Form templates now inherit from templates/flask_user/public_base.html,
templates/flask_user/member_base.html and templates/base.html.
```

- v0.4.2 Cleanup of SQLAlchemyAdapter. Added tox for Python 3.4
- v0.4.1 Cleanup of customized email messages and signals.

```
v0.4.1 API changes
- User.email_confirmed_at                              --> confirmed_at
- templates/flask_user/emails/confirmation_email_*     --> registered_*
- signals.confirmation_email_set                       --> user_registered
- template variable {{ confirmation_link }}            --> {{ confirm_email_link }}
- templates/flask_user/emails/reset_password_*         --> forgot_password_*
- signals.reset_password_email_sent                    --> user_forgot_password
```

- v0.4.0 Beta release. Translations via Babel.
- v0.3.8 Role-based authorization via @roles_required.
- v0.3.5 Support for Python 2.6, 2.7 and 3.3, Event notifications.
- v0.3.1 Alpha release. Email sending, Confirm email, Forgot password, Reset password.
- v0.2 Change username, Change password.
- v0.1 Register, Login, Logout.

CHAPTER 9

Acknowledgements

This project would not be possible without the use of the following amazing offerings:

- Flask
- Flask-Babel
- Flask-Login
- Flask-Mail
- SQLAlchemy and Flask-SQLAlchemy
- WTForms and Flask-WTF

CHAPTER 10

Contributors

- https://github.com/neurosnap : Register by invitation only
- https://github.com/lilac : Chinese translation
- https://github.com/cranberyxl : Bugfix for login_endpoint & macros.label
- https://github.com/markosys : Early testing and feedback

CHAPTER 11

Alternative Flask extensions

- Flask-Login
- Flask-Security