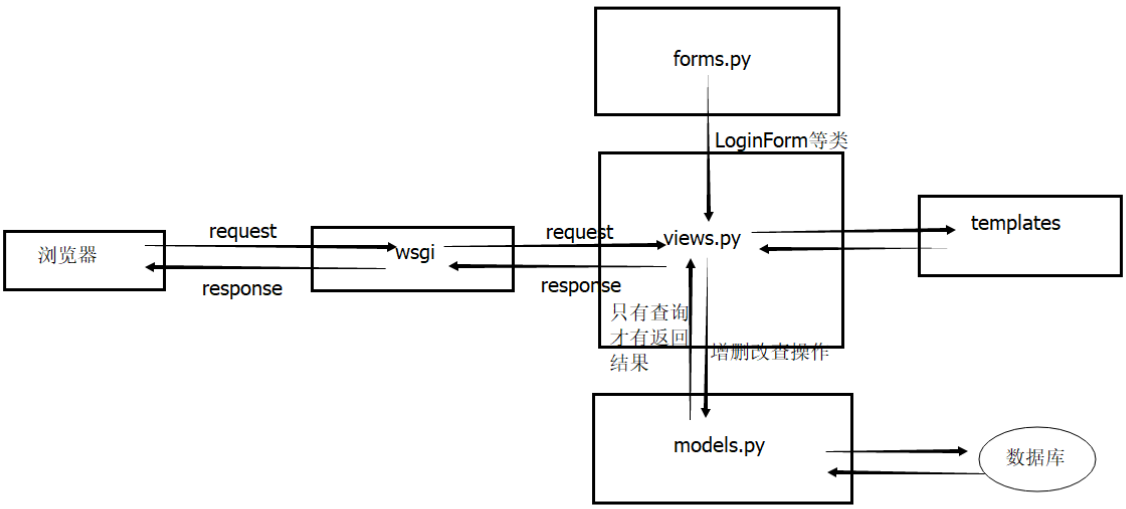
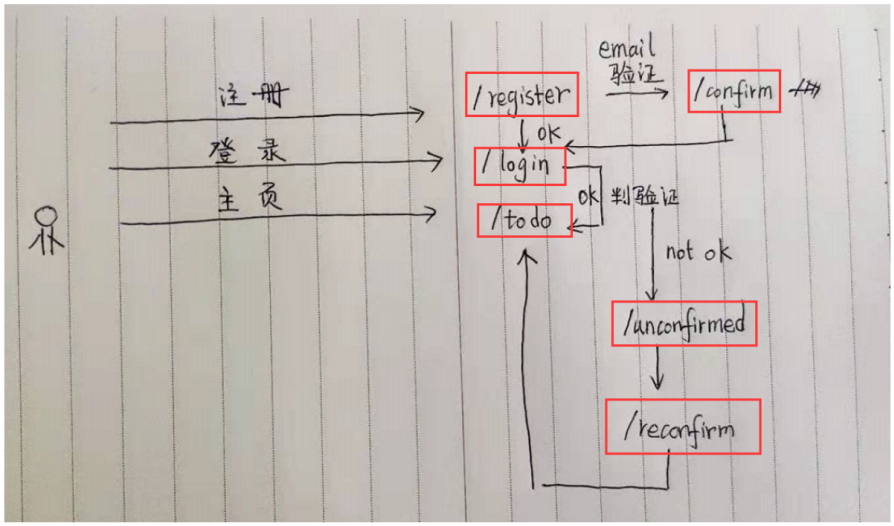
Flask的基本工作流程





用户认证

大多数程序都要进行用户跟踪。用户连接程序时会进行身份认证,通过这一过程,让程序知道自己的身份。程序知道用户是谁后,就能提供有针对性的体验。

数据库模型

技术要点

`Werkzeug` 中的 `security` 模块能够很方便地实现密码散列值的计算。这一功能的实现只需要两个函数,分别用在注册用户和验证用户阶段。

- `generate_password_hash(password, method= pbkdf2:sha1 , salt_length=8)` :密码加密的散列值。
- `check_password_hash(hash, password)` :密码散列值和用户输入的密码是否匹配.

@property 是经典类中的一种装饰器，新式类中具有三种:

装饰器表示	装饰器含义
@property	获取属性
@方法名.setter	修改属性
@方法名.deleter	删除属性

核心代码

```
# app/models.py
from app import db
from werkzeug.security import generate_password_hash, check_password_hash

class Role(db.Model):
    """用户类型"""
    __tablename__ = 'roles'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64), unique=True)
    users = db.relationship('User', backref='role')

    def __repr__(self):
        return '<Role % r>' % self.name


class User(db.Model):
    """用户"""
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), unique=True, index=True)
    password_hash = db.Column(db.String(128))  # 加密的密码
    role_id = db.Column(db.Integer, db.ForeignKey('roles.id'))

    @property
    def password(self):
        raise AttributeError('password is not a readable attribute')

    @password.setter
    def password(self, password):
        # generate_password_hash(password, method= pbkdf2:sha1 , salt_length=8)
:密码加密的散列值。
        self.password_hash = generate_password_hash(password)

    def verify_password(self, password):
        # check_password_hash(hash, password) :密码散列值和用户输入的密码是否匹配.
        return check_password_hash(self.password_hash, password)

    def __repr__(self):
        return '<User % r>' % self.username
```

测试代码

```
# tests/test_user_model.py

import unittest

from app.models import User


class UserModelTestCase(unittest.TestCase):
    """
    用户数据库模型测试
    """
    def test_password_setter(self):
        """测试新建的用户密码是否为空?"""
        u = User(password='cat')
        self.assertTrue(u.password_hash is not None)

    def test_no_password_getter(self):
        """测试获取密码信息是否报错?"""
        u = User(password='cat')
        with self.assertRaises(AttributeError):
            password =u.password

    def test_password_verification(self):
        """测试加密密码和明文密码是否验证正确?"""
        u = User(password='cat')
        self.assertTrue(u.verify_password('cat'))
        self.assertFalse(u.verify_password('dog'))

    def test_password_salts_are_random(self):
        """测试每次密码加密的加密字符是否不一致?"""
        u = User(password='cat')
        u2 = User(password='cat')
        self.assertTrue(u.password_hash != u2.password_hash)
```

测试用例运行效果如下:

```
[kiosk@foundation0 Todolist]$ python3 manage.py  test
test_app_exists (test_basics.BasicsTestCase)
测试当前app是否存在? ... ok
test_app_is_testing (test_basics.BasicsTestCase)
测试当前app是否为测试环境? ... ok
test_choice (test_number.TestSequenceFunctions) ... ok
test_sample (test_number.TestSequenceFunctions) ... ok
test_shuffle (test_number.TestSequenceFunctions) ... ok
test_no_password_getter (test_user_model.UserModelTestCase)
测试获取密码信息是否报错? ... ok
test_password_salts_are_random (test_user_model.UserModelTestCase)
测试每次密码加密的加密字符是否不一致? ... ok
test_password_setter (test_user_model.UserModelTestCase)
测试新建的用户密码是否为空? ... ok
test_password_verification (test_user_model.UserModelTestCase)
测试加密密码和明文密码是否验证正确? ... ok

----------------------------------------------------------------------
Ran 9 tests in 0.302s

OK
```

测试用例执行的命令

测试用例全部通过

Flask-Login优化数据库模型

技术要点

用户登录程序后,他们的认证状态要被记录下来,这样浏览不同的页面时才能记住这个状态。Flask-Login是个非常有用的小型扩展,专门用来管理用户认证系统中的认证状态,且不依赖特定的认证机制。

Flask-Login 提供了一个 `UserMixin` 类,包含常用方法的默认实现,且能满足大多数需求。

方法名或属性名	功能
is_authenticated	用户是否已经登录?
is_active	是否允许用户登录?False代表用户禁用
is_anonymous	是否 匿名用户?
get_id()	返回用户的唯一标识符

核心代码

Flask-Login 在程序的工厂函数中初始化, 修改文件`app/__init__.py`:

session_protection 属性提供不同的安全等级防止用户会话遭篡改。可以设为:

- None
- 'basic'
- 'strong' : 记录客户端 `IP` 地址和浏览器的用户代理信息,如果发现异动就登出用户。

```
# app/__init__.py

from flask_login import LoginManager

# .......此处省略前面重复的代码
login_manager = LoginManager()
# session_protection 属性提供不同的安全等级防止用户会话遭篡改。
login_manager.session_protection = 'strong'
# login_view 属性设置登录页面的端点。
login_manager.login_view = 'auth.login'


def create_app(config_name='development'):
    #  .......
    # 用户认证新加扩展
    login_manager.init_app(app)
    # ........
    return app
```

`app/models.py` :修改 User 模型,支持用户登录, 同时Flask-Login 要求程序实现一个回调函数,使用指定的标识符加载用户。

```
# app/models.py
from flask_login import UserMixin
from . import  login_manager

"""
Flask-Login 提供了一个 UserMixin 类,包含常用方法的默认实现,且能满足大多数需求。
    1). is_authenticated     用户是否已经登录?
```

```
    2). is_active           是否允许用户登录?False代表用户禁用
    3). is_anonymous        是否匿名用户?
    4). get_id()            返回用户的唯一标识符
"""


class User(UserMixin, db.Model):
    """用户"""
    # ............
    # 电子邮件地址email,相对于用户名而言,用户更不容易忘记自己的电子邮件地址。
    email = db.Column(db.String(64), unique=True, index=True)
    # ............

# 加载用户的回调函数;如果能找到用户,返回用户对象;否则返回 None 。
@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))
```

数据库创建

Flask-Migrate 插件提供`Alembic`（`Database migration`数据迁移跟踪记录）提供的数据库升级和降级的功能。它所能实现的效果有如 Git 管理项目代码一般。

在新数据库中创建数据表。可使用如下Bash命令创建数据表或者升级到最新修订版本:

```
# 初始化数据库，创建migrations目录，存放了所有迁移脚本。只需要执行一次。
python3 manage.py  db init
# 创建迁移脚本
python3 manage.py  db migrate
# 更新数据库
python3 manage.py  db upgrade
```

用户注册

如果新用户想成为程序的成员,必须在程序中注册,这样程序才能识别并登入用户。程序的登录页面中要显示一个链接,把用户带到注册页面,让用户输入电子邮件地址、用户名和密码。

用户注册表单

注册页面使用的表单要求用户输入电子邮件地址、用户名和密码。

```
# app/auth/forms.py:用户注册表单

from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, SubmitField
from wtforms.validators import DataRequired, Length, Email, Regexp, EqualTo,
ValidationError
from app.models import User

class RegistrationForm(FlaskForm):
    email = StringField('电子邮箱', validators=[
        DataRequired(), Length(1, 64), Email()])
    username = StringField('用户名', validators=[
        DataRequired(), Length(1, 64),
        Regexp('^\w*$', message='用户名只能由字母数字或者下划线组成')])
```

```
    password = PasswordField('密码', validators=[
        DataRequired(), EqualTo('repassword', message='密码不一致')])
    repassword = PasswordField('确认密码', validators=[DataRequired()])
    submit = SubmitField('注册')

    # 两个自定义的验证函数, 以validate_ 开头且跟着字段名的方法,这个方法和常规的验证函数一起调用。
    def validate_email(self, field):
        if User.query.filter_by(email=field.data).first():
            # 自定义的验证函数要想表示验证失败,可以抛出 ValidationError 异常,其参数就是错误消息。
            raise ValidationError('电子邮箱已经注册.')

    def validate_username(self, field):
        if User.query.filter_by(username=field.data).first():
            raise ValidationError('用户名已经占用.')
```

用户注册业务逻辑

处理用户注册的过程没有什么难以理解的地方。提交注册表单,通过验证后,系统就使用用户填写的信息在数据库中添加一个新用户。处理这个任务的视图函数如下所示:

```
# app/auth/views.py:用户注册路由

from flask import request, redirect, url_for, flash, render_template
from flask_login import login_user, login_required, logout_user
from app import db
from app.auth.forms import RegistrationForm
from app.models import User
from . import auth

# .........

@auth.route('/register', methods=['GET', 'POST'])
def register():
    form = RegistrationForm()
    if form.validate_on_submit():
        user = User()
        user.email = form.email.data
        user.username = form.username.data
        user.password = form.password.data
        db.session.add(user)
        flash('注册成功, 请登录')
        return redirect(url_for('auth.login'))
    return render_template('auth/register.html', form=form)

# ...........
```

用户注册前端渲染

登录页面使用的模板保存在 `templates/auth/register.html` 文件中。这个模板只需使用 Flask-Bootstrap 提供的 `wtf.quick_form()` 宏渲染表单即可。

```
# templates/auth/login.html

{% extends 'bootstrap/base.html' %}
```

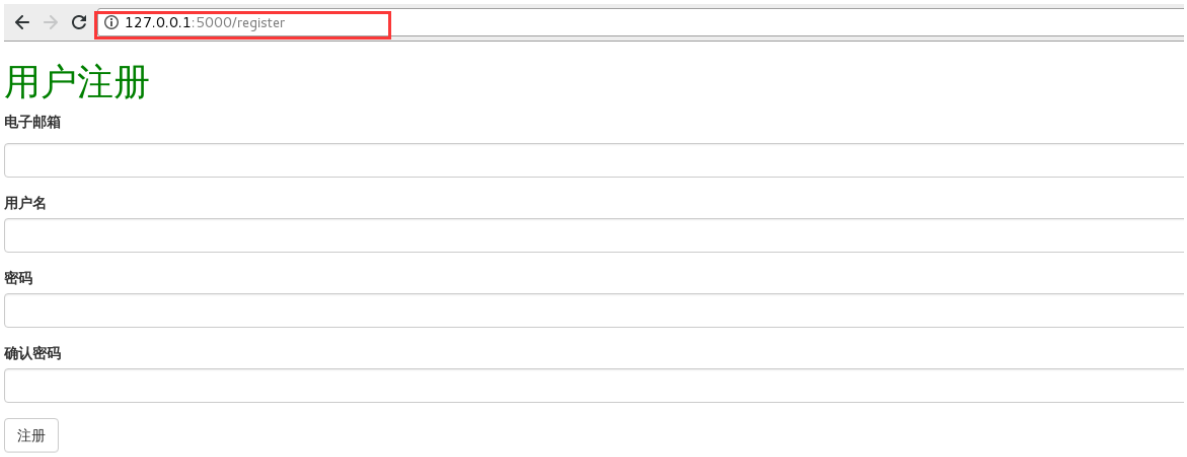
```
{% import 'bootstrap/wtf.html'  as wtf %}


{% block navbar %}
    <ul class="nav navbar-nav navbar-right">
        {% if current_user.is_authenticated %}
            <li><a href="{{ url_for('auth.logout') }}">注销</a></li>
        {% else %}
            <li><a href="{{ url_for('auth.login') }}">登录</a></li>
        {% endif %}

        <p style="color: red">{{ get_flashed_messages() }}</p>
    </ul>

{% endblock %}
{% block content %}
    <h1 style="color:green">用户注册</h1>
    {{ wtf.quick_form(form) }}
{% endblock %}
```

用户注册页面简易效果展示



用户登录

用户登录表单

用户的登录表单中包含一个用于输入电子邮件地址的文本字段、一个密码字段、提交按钮。

```
# app/auth/forms.py:用户登录表单

class LoginForm(FlaskForm):
    """用户登录表单"""
    email = StringField('电子邮箱', validators=[DataRequired(), Length(1, 64),
Email()])
    password = PasswordField('密码', validators=[DataRequired()])
    submit = SubmitField('登录')
```

用户登录业务逻辑

- 根据用户提交的 email 从数据库中加载用户， 判断用户存在?

- 如果存在, 调用用户对象的 verify_password() 方法,其参数是表单中填写的密码。判断用户密码是否正确?
- 如果密码正确,则调用 Flask-Login 中的 login_user() 函数实现用户登录。

login_user() 函数的参数是要登录的用户,以及可选的"记住我"布尔值,"记住我"也在表单中填写。

- 如果值为 False ,那么关闭浏览器后用户会话就过期了,所以下次用户访问时要重新登录。
- 如果值为 True ,那么会在用户浏览器中写入一个长期有效的 cookie,使用这个 cookie 可以复现用户会话。

```
# app/auth/views.py
# ...........
@auth.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        user = User.query.filter_by(email=form.email.data).first()
        if user is not None and user.verify_password(form.password.data):
            # 调用 Flask-Login 中的 login_user() 函数,在用户会话中把用户标记为已登录。
            # login_user() 函数的参数是要登录的用户,以及可选的"记住我"布尔值,"记住我"也在
表单中填写。
            login_user(user)
            return redirect(request.args.get('next') or url_for('auth.login'))
        flash('无效的用户名和密码.')
    return render_template('auth/login.html', form=form)

# ........
```

用户登录前端渲染

登录页面使用的模板保存在 `templates/auth/login.html` 文件中。这个模板只需使用 Flask-Bootstrap 提供的 `wtf.quick_form()` 宏渲染表单即可。

```
# templates/auth/login.html

{% extends 'bootstrap/base.html' %}
{% import 'bootstrap/wtf.html'  as wtf %}


{% block navbar %}
    <ul class="nav navbar-nav navbar-right">
        {% if current_user.is_authenticated %}

            <li><a href="{{ url_for('auth.logout') }}">注销</a></li>
        {% else %}
            <li><a href="{{ url_for('auth.login') }}">登录</a></li>
        {% endif %}

    <p style="color: red">{{ get_flashed_messages() }}</p>
    </ul>

{% endblock %}
{% block content %}
    <h1 style="color:green">用户登录</h1>
```

```
    {{ wtf.quick_form(form) }}
{% endblock %}
```

用户登录页面简易效果展示

用户注销

为了登出用户,这个视图函数调用 Flask-Login 中的 logout_user() 函数,删除并重设用户会话。随后会显示一个 Flash 消息,确认这次操作,再重定向到首页,这样登出就完成了。

```
# app/auth/views.py

@auth.route('/logout')
@login_required
def logout():
    logout_user()
    flash('用户注销成功.')
    return redirect(url_for('auth.login'))
```

Github与用户认证

命令行提交项目代码到Github， 命令如下:

```
git add *
git commit -m "基于Flask的任务清单管理系统(二): 用户认证基本功能实现"
git push origin master
```

效果如下:

```
[kiosk@foundation0 Todolist]$ git push origin master
Username for 'https://github.com': lvah
Password for 'https://lvah@github.com':
Counting objects: 62, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (41/41), done.
Writing objects: 100% (46/46), 21.57 KiB | 0 bytes/s, done.
Total 46 (delta 5), reused 0 (delta 0)
remote: Resolving deltas: 100% (5/5), completed with 4 local objects.
To https://github.com/lvah/TodoList.git
   a1d7289..3ee21d7  master -> master
```

单元测试技术详解

web单元测试的分类

- 测试对象较独立，无需依赖于cookie之类的上下文
- 依赖于上下文

- web前端的测试。

测试方式

- 第一种类型只需要使用`unittest`的常规测试即可
- 第二种类型，例如对于login_required类型的endpoint，可使用`app.test_client()`返回测试客户端，同时附带上合适的数据。推荐使用**flask-testing**插件。同时，由于这类依赖比较常见，所以推荐将其独立成类。
- 第三种类型可使用selenium，但是编写selenium工作量比较大，且不够直观，且不够只管，建议使用其他方式或者人工测试

代码组织推荐：

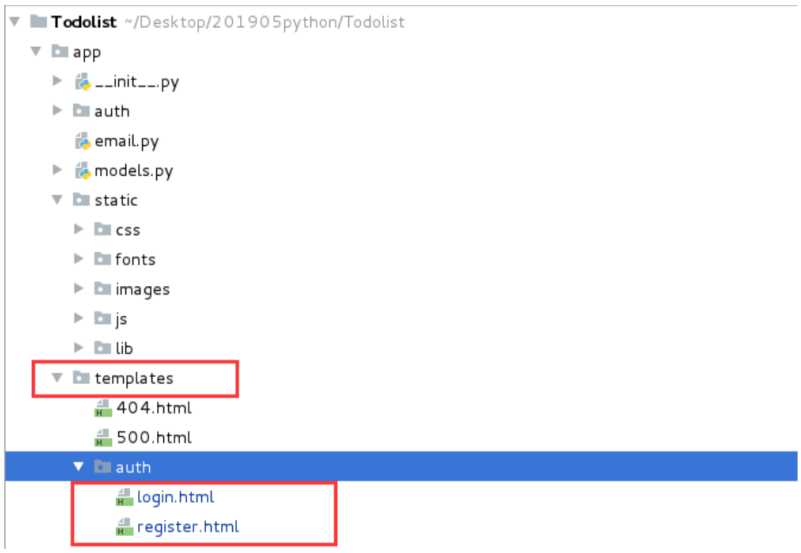
- 测试时，依赖于某些数据，除非测试数据的增删改，否则建议编写**数据导入函数**（后续写），可以减少工作量

前端页面优化

下面的步骤是根据登录界面进行修改， 注册页面修改也是同样的方式。

1). 导入前端页面需要的静态资源， 如下图所示:

2). 资源文件夹中包含`login.html`和`register.html`两个文件, 拷贝文件到`templates/auth`目录, 如下图所示:



3). 查找文件中访问静态资源的 `html` 代码并修改静态资源:

```
<!-- Bootstrap -->
<link href="https://cdn.jsdelivr.net/npm/bootstrap@3.3.7/dist/css/bootstrap.min.css" rel="stylesheet">
<link rel="stylesheet" href="./css/font.css">
<link rel="stylesheet" href="./css/xadmin.css">

<link rel="stylesheet" href="./css/login.css">
<script type="text/javascript" src="https://cdn.bootcss.com/jquery/3.2.1/jquery.min.js"></script>
<!-- jQuery (Bootstrap 的所有 JavaScript 插件都依赖 jQuery，所以必须放在前边) -->
<script src="https://cdn.jsdelivr.net/npm/jquery@1.12.4/dist/jquery.min.js"></script>
<!-- 加载 Bootstrap 的所有 JavaScript 插件。你也可以根据需要只加载单个插件。 -->
<script src="https://cdn.jsdelivr.net/npm/bootstrap@3.3.7/dist/js/bootstrap.min.js"></script>
<script src="./lib/layui/layui.js"></script>
<!--[if lt IE 9]>
<script src="https://cdn.staticfile.org/html5shiv/r29/html5.min.js"></script>
<script src="https://cdn.staticfile.org/respond.js/1.4.2/respond.min.js"></script>
<![endif]-->
```

修改静态资源如下:

```
<!-- Bootstrap -->
<link href="https://cdn.jsdelivr.net/npm/bootstrap@3.3.7/dist/css/bootstrap.min.css" rel="stylesheet">
<link rel="stylesheet" href="{{ url_for('static', filename='css/font.css') }}">
<link rel="stylesheet" href="{{ url_for('static', filename='css/xadmin.css') }} ">

<link rel="stylesheet" href="{{ url_for('static', filename='css/login.css') }} ">
<script type="text/javascript" src="https://cdn.bootcss.com/jquery/3.2.1/jquery.min.js"></script>
<!-- jQuery (Bootstrap 的所有 JavaScript 插件都依赖 jQuery，所以必须放在前边) -->
<script src="https://cdn.jsdelivr.net/npm/jquery@1.12.4/dist/jquery.min.js"></script>
<!-- 加载 Bootstrap 的所有 JavaScript 插件。你也可以根据需要只加载单个插件。 -->
<script src="https://cdn.jsdelivr.net/npm/bootstrap@3.3.7/dist/js/bootstrap.min.js"></script>
<script src="{{ url_for('static', filename='lib/layui/layui.js') }} "></script>
<!--[if lt IE 9]>
<script src="https://cdn.staticfile.org/html5shiv/r29/html5.min.js"></script>
<script src="https://cdn.staticfile.org/respond.js/1.4.2/respond.min.js"></script>
<![endif]-->
```

访问到下图页面， 即可认为静态资源位置修改成功， 就可以接着做下面的其他操作了。

4). 修改文件`templates/auth/login.html`表单提交需要的信息， 如下图所示:

```
<form method="post" class="layui-form" action="{{ url_for('auth.login') }}">  表单信息提交的路由地址
    {{ form.hidden_tag() }}隐藏的csrf
    {{ form.email() }}生成邮箱对应的html
    {#        <input name="username" placeholder="用户名" type="text" lay-verify="required" class="layui-input">#}
    {#        <p class="error">用户登录失败</p>#}
    <p class="error">{{ form.email.errors[0] }}</p> 生成邮箱错误提示的html
    <hr class="hr15">
    {{ form.password() }} 生成密码对应的html
    {#        <input name="password" lay-verify="required" placeholder="密码" type="password" class="layui-input">#}
    {#        <p class="error"> 密码失败</p>#}
    <p class="error"> {{ form.password.errors[0] }}</p> 生成密码错误提示的html
    <hr class="hr15">
    {{ form.submit() }} 生成提交按钮的html
    {#        <input value="登录"  style="width:100%;" type="submit">#}
    <hr class="hr20">
</form>
```

这些样式需要在app/auth/forms.py文件中添加

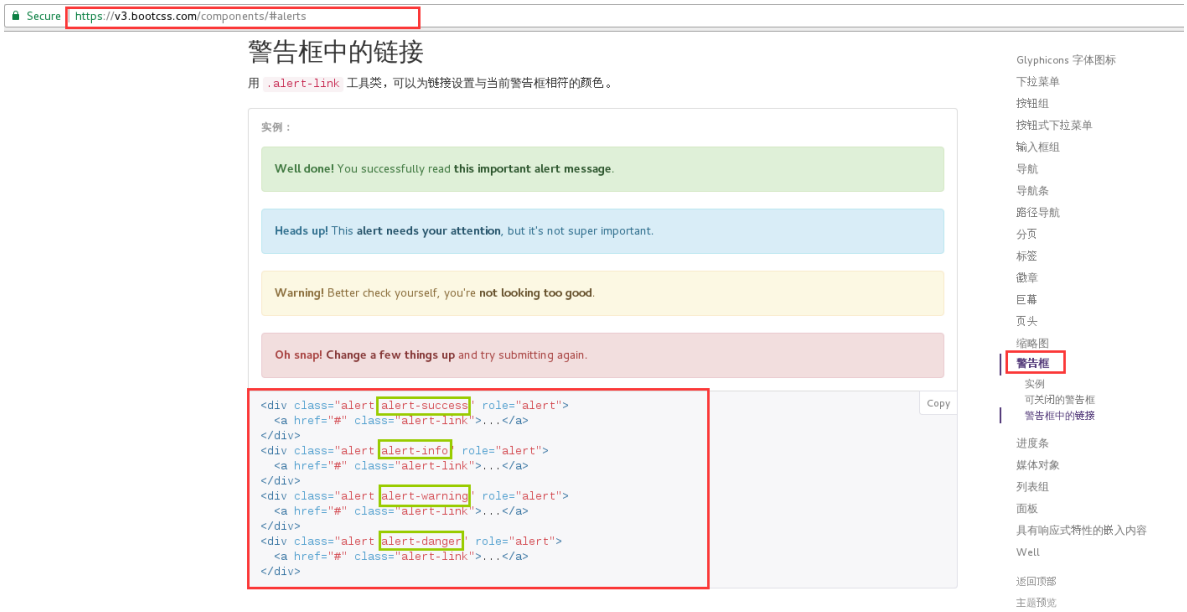
5). 实现分类闪现，详细的参考消息闪现flash的文档。

当我们flash闪现信息时， 指定闪现信息的类型， 便于前端的分类展示 ， 修改文件`app/auth/views.py`如下所示:

```
@auth.route('/logout')
@login_required
def logout():
    logout_user()
    flash('用户注销成功.', category='error')
    return redirect(url_for('auth.login'))

@auth.route('/register', methods=['GET', 'POST'])
def register():
    form = RegistrationForm()
    if form.validate_on_submit():
        user = User()
        user.email = form.email.data
        user.username = form.username.data
        user.password = form.password.data
        db.session.add(user)
        flash('注册成功， 请登录', category='success')
        return redirect(url_for('auth.login'))
    return render_template('auth/register.html', form=form)
```

Bootstrap提供的闪现组件警告框网址。 警告框组件通过提供一些灵活的预定义消息，为常见的用户动作提供反馈消息。



而闪现信息的显示在很多场景都会使用， 我们把它独立成一个文件`templates/flash.html`， 代码如下:

```
{% for message in get_flashed_messages(category_filter=['success']) %}
    <div class="alert alert-warning alert-dismissible" role="alert">
        <button type="button" class="close" data-dismiss="alert" aria-
label="Close"><span
                aria-hidden="true">&times;</span>
        </button>
        <strong>Success! </strong> {{ message }}
    </div>
{% endfor %}

{% for message in get_flashed_messages(category_filter=['error']) %}
    <div class="alert alert-warning alert-dismissible" role="alert">
        <button type="button" class="close" data-dismiss="alert" aria-
label="Close"><span
                aria-hidden="true">&times;</span>
        </button>
        <strong>Warning! </strong> {{ message }}
    </div>
{% endfor %}
```

登录的前端删除自带的闪现代码， include刚才编写的前端文件`templates/auth/login.html`即可，如下图所示:

```
<div class="login layui-anim layui-anim-up">
    <div class="title">任务清单系统登录</div>
    <div id="darkbannerwrap"></div>
    {% include 'flash.html' %}
    <form method="post" class="layui-form" action="{{ url_for('auth.login') }}">
        {{ form.hidden_tag() }}
        {{ form.email() }}
        {#        <input name="username" placeholder="用户名" type="text" lay-
verify="required" class="layui-input">#}
        {#        <p class="error">用户登录失败</p>#}
        <p class="error">{{ form.email.errors[0] }}</p>
        <hr class="hr15">
        {{ form.password() }}
```

```
        {#        <input name="password" lay-verify="required" placeholder="密码"
type="password" class="layui-input">#}
        {#        <p class="error"> 密码失败</p>#}
        <p class="error"> {{ form.password.errors[0] }}</p>
        <hr class="hr15">
        {{ form.submit() }}
        {#        <input value="登录"  style="width:100%;" type="submit">#}
        <hr class="hr20">
    </form>
</div>
```

6). 为了让前端的`CSS`样式生效， 还需要在表单的字段域中租用前端的属性信息， 如下图所示:

```
# app/auth/forms.py
# 此处省略...........
class LoginForm(FlaskForm):
    email = StringField('电子邮箱', validators=[DataRequired(), Length(1, 64),
Email()],
                        # 给前端的标签添加下面的属性信息;
                        render_kw={'class': 'layui-input',
                                   'placeholder': '电子邮箱',
                                   'lay-verify': 'required'})
    password = PasswordField('密码', validators=[DataRequired()],
                        # 给前端的标签添加下面的属性信息;
                        render_kw={'class': 'layui-input',
                                   'placeholder': '密码',
                                   'lay-verify': 'required'}
                             )
    # .......此处省略
```

通过上面的一番折腾， 好看的登录页面和注册页面就搞定了， 可以开始注册和登录了. 需要测试的几个要点:

- 页面是否完整显示?
- 登录注册功能是否可以测试通过?
- 闪现信息是否分类显示?

开始动手测一测吧。

用户邮箱验证

- 如何确认注册用户提供的信息是否正确?

  常用方式是验证电子邮件地址。用户注册后,程序会立即发送一封确认邮件。新账户先被标记成待确认状态,用户按照邮件中的说明操作后,才能证明自己可以被联系上。账户确认过程中,往往会要求用户点击一个包含确认令牌的特殊 URL 链接。

- 确认令牌的特殊 URL 链接该如何设计?
  - 链接形式是 `http://www.example.com/auth/confirm/<id>` , id 是用户的 id 。代表账户id确认成功。
  - 出现的问题: id可任意指定， 从而确认任意账户。
  - 解决方法： id 进行安全加密后得到令牌。

- 如何生成包含用户 id 的安全令牌?

  `itsdangerous` 模块中的 `TimedJSONWebSignatureSerializer` 类生成具有过期时间的 `JSON Web` 签名(`JSON Web Signatures,JWS`)

```
from itsdangerous import TimedJSONWebSignatureSerializer
# 生成具有过期时间的 `JSON Web` 签名对象,
#   1). westos是一个密钥,在 Flask 程序中可使用 SECRET_KEY 设置。
#   2). expires_in 参数设置令牌的过期时间,单位为秒。
s = TimedJSONWebSignatureSerializer('westos', expires_in = 3600)
# dumps() 方法为指定的数据生成一个加密签名,然后再对数据和签名进行序列化,生成令牌字符串。
token = s.dumps({ 'confirm': 23 })
# loads() 方法会检验签名和过期时间,如果通过,返回原始数据。否则抛出异常。
data = s.loads(token)
# data = { 'confirm': 23 }
```

  如何将这种生成和检验令牌的功能可添加到 User 模型中。可以按照下面的步骤实现:

`/register/ -> /auth/email/confirm`

邮箱验证数据库模型

- 模型中新加入了一个列confirmed用来保存账户的确认状态。
- generate_confirmation_token() 方法生成一个令牌,有效期默认为一小时。
- confirm()方法检验令牌和检查令牌中id和已登录用户id是否匹配?
  - 如果检验通过,则把新添加的 confirmed 属性设为 True

```
# app/models.py
```

```
class User(UserMixin, db.Model):
    # ..................
    confirmed = db.Column(db.Boolean, default=False)

    def generate_confirmation_token(self, expiration=3600):
        """生成一个令牌,有效期默认为一小时。"""
        s = TimedJSONWebSignatureSerializer(current_app.config['SECRET_KEY'],
expiration)
        return s.dumps({'confirm': self.id})

    def confirm(self, token):
        """
        检验令牌和检查令牌中id和已登录用户id是否匹配?如果检验通过,则把新添加的 confirmed 属
性设为 True
        """
        s = TimedJSONWebSignatureSerializer(current_app.config['SECRET_KEY'])
        try:
            data = s.loads(token)
        except:
            return False
        if data.get('confirm') != self.id:
            return False
        self.confirmed = True
        db.session.add(self)
        db.session.commit()
        return True
```

由于模型中新加入了一个列用来保存账户的确认状态,因此要生成并执行一个新数据库迁移。 执行shell命令如下:

```
# 对数据库（db）进行迁移（migrate）。-m选项用来添加迁移备注信息。
python3 manage.py  db migrate -m "添加账户的确认状态"
# 生成了迁移脚本后，使用upgrade子命令即可更新数据库.
python3 manage.py  db upgrade
```

用户注册验证邮件的业务逻辑

当前的 /register 路由把新用户添加到数据库中后,会重定向到 /index。在重定向之前,这个路由需要发送确认邮件。

电子邮件模板准备

认证蓝本使用的电子邮件模板保存在 `templates/auth/confirm.html` 文件中, 参考微信的模板。可在资料包中下载 `email.html` 文件。 修改如下:

- `url_for()` 函数中的 _external=True 参数要求程序生成完整的 `URL`

```
<p>你好!</p>
<p>
    感谢你注册任务管理平台。 <br>
    你的登录邮箱为：<a href="mailto:{{ user.username }}" rel="noopener" target="_blank">{{ user.username }}</a>。请点击以下链接激活
</p>
<p style="...">
    <a href="{{ url_for('auth.confirm', token=token, _external=True) }}"
       target="_blank" rel="noopener">
        {{ url_for('auth.confirm', token=token, _external=True) }}
    </a>
```

电子邮件配置信息准备

```
# config.py

class DevelopmentConfig(Config):
    # ..........
    # 添加的发送邮件的配置信息
    MAIL_SERVER = 'smtp.qq.com'
    # 指定端口， 默认25， 但qq邮箱默认为 端口号465或587;
    MAIL_PORT = 465
    """
    由于QQ邮箱不支持非加密的协议，那么使用加密协议，分为两种加密协议，选择其中之一即可
    """
    MAIL_USE_SSL = True
    # MAIL_USE_TLS = True
    MAIL_USERNAME = '976131979'
    # 此处的密码并非邮箱登录密码， 而是开启pop3
    MAIL_PASSWORD = "自己的密码123sdjuoeyoxjoubedb"
    # ............
```

发送电子邮件的业务逻辑

如果发送了多封测试邮件,页面停滞了几秒钟,在这个过程中浏览器就像无响应一样。为了避免处理请求过程中不必要的延迟,我们可以把发送电子邮件的函数移到后台线程中， 实现多线程发送用户验证邮件。

```
# app/email.py

from threading import Thread
```

```python
from flask import render_template, current_app
from flask_mail import Mail, Message

def thread_task(app, mail, msg):
    with app.app_context():
        mail.send(msg)


def send_mail(to, subject, filename, **kwargs):
    """
    发送邮件的封装
    :param to: 收件人
    :param subject: 邮件主题
    :param filename: 邮件正文对应的html名称
    :param kwargs: 关键字参数， 模版中需要的变量名
    :return:
    """
    app = current_app._get_current_object()
    # 初始化mail对象, 一定要先配置邮件信息;
    mail = Mail(app)
    msg = Message(subject=subject,
                  sender='976131979@qq.com',
                  recipients=to,
                  )
    # msg.body = info
    msg.html = render_template(filename + '.html', **kwargs)
    # with app.app_context():
    #     mail.send(msg)
    thread = Thread(target=thread_task, args=(app, mail, msg))
    thread.start()
    return thread
```

注册与确认验证的业务逻辑

- 注册与发送验证邮件的视图函数

当前的 /register 路由把新用户添加到数据库中后,会重定向到 /index。在重定向之前,这个路由需要发送确认邮件, 确认邮件信息参考验证邮件模板。

此处如果注册成功且验证通过， 应该跳转到网站的首页， 请自行编写主页的业务逻辑.

```python
# app/auth/views.py

# .........
@auth.route('/register', methods=['GET', 'POST'])
def register():
        # ..........
        db.session.add(user)
        # 提交数据库之后才能赋予新用户 id 值,而确认令牌需要用到 id ,所以不能延后提交。
        db.session.commit()
        token = user.generate_confirmation_token()
        # 发送邮件的函数Flask-Mail封装过， 可参考之前讲的内容;
        send_mail(user.email, '请激活你的任务管理平台帐号',
                  'auth/confirm', user=user, token=token)
        flash('平台验证消息已经发送到你的邮箱， 请确认后登录.', category='success')
```

```
        return redirect(url_for('todo.index'))
    return render_template('auth/register.html', form=form)
```

- 确认账户的视图函数如下面代码所示。

  Flask-Login 提供的 login_required 修饰器会保护这个路由,因此,用户点击确认邮件中的链接后,要先登录,然后才能执行这个视图函数。这个函数先检查已登录的用户是否已经确认过,如果确认过,则重定向到首页,因为很
  显然此时不用做什么操作。这样处理可以避免用户不小心多次点击确认令牌带来的额外工作。

```
# app/auth/views.py
@auth.route('/confirm/<token>')
@login_required
def confirm(token):
    if current_user.confirmed:
        return redirect(url_for('todo.index'))
    if current_user.confirm(token):
        flash('验证邮箱通过', category='success')
    else:
        flash('验证连接失效', category='error')
    return redirect(url_for('todo.index'))
```

- 过滤未确认的账户

每个程序都可以决定用户确认账户之前可以做哪些操作。比如,允许未确认的用户登录,但只显示一个未确认页面`unconfirmed.html`,这个页面要求用户在获取权限之前先确认账户。这一步可使用 Flask 提供的 before_request 钩子完成。

对蓝图来说, before_request 钩子只能应用到属于蓝图的请求上。若想在蓝图中使用针对程序全局请求的钩子,必须使用`before_app_request`修饰器。

同时满足以下 3 个条件时,`before_app_request`处理程序会拦截请求。

- 用户已登录(`current_user.is_authenticated`必须返回 True)。
- 用户的账户还未确认。
- 请求的端点(使用`request.endpoint`获取)不在`auth`蓝图中。访问`auth`路由要获取权限,因为这些路由的作用是让用户确认账户或执行其他账户管理操作。

  如果请求满足以上 3 个条件,则会被重定向到`/auth/unconfirmed`路由,显示一个确认账户相关信息的页面。

```
# app/auth/views.py

@auth.before_app_request
def before_request():
    if current_user.is_authenticated \
            and not current_user.confirmed \
            and request.endpoint[:5] != 'auth.' \
            and request.endpoint != 'static':
        return redirect(url_for('auth.unconfirmed'))

@auth.route('/unconfirmed')
```

```
def unconfirmed():
    # 如果当前用户是匿名用户或者已经验证的用户， 则访问主页，否则进入未验证界面;
    if current_user.is_anonymous or current_user.confirmed:
        return redirect(url_for('todo.index'))
    token = current_user.generate_confirmation_token()
    return render_template('auth/unconfirmed.html')
```

未确认页面的`html`可自行设计， 此处给出范例代码:

```
# templates/auth/unconfirmed.html

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<p>尊敬的用户:</p>
你好， 你还没有确认邮箱账户， 请点击下面连接进行确认.
<a href="{{ url_for('auth.resend_confirmation', _external=True ) }}">再次发送确认
邮件</a>
</body>
</html>
```

显示给未确认用户的页面提供了一个链接,用于请求发送新的确认邮件,以防之前的邮件丢失。重新发送确认邮件的视图函数代码如下:

```
# app/auth/views.py

@auth.route('/reconfirm')
@login_required
def resend_confirmation():
    token = current_user.generate_confirmation_token()
    try:
        send_mail([current_user.email], '请激活你的任务管理平台帐号',
                  'auth/confirm', user=current_user, token=token)
    except Exception as e:
        print(e)
        flash(str(e), category='error')
        return redirect(url_for('auth.register'))
    else:
        flash('新的平台验证消息已经发送到你的邮箱， 请确认后登录.', category='success')
        return redirect(url_for('todo.index'))
```

综上所述我们的用户邮箱验证全部完成。

拥有程序账户的用户有时可能需要修改账户信息。下面这些操作可使用本章介绍的技术添加到验证蓝本中。

- 修改密码

- 重设密码
  为避免用户忘记密码无法登入的情况,程序可以提供重设密码功能。安全起见,有必要使用类似于确认账户时用到的令牌。用户请求重设密码后,程序会向用户注册时提供的电子邮件地址发送一封包含重设令牌的邮件。用户点击邮件中的链接,令牌验证后,会显示一个用于输入新密码的表单。
- 修改电子邮件地址