**FLASK**

Contents

[**Chapter 1: Hello, World!** 1](#_Toc69754599)

[**Chapter 2: Templates** 3](#_Toc69754600)

[**Chapter 3: Web forms** 5](#_Toc69754601)

[**Chapter 4: Database** 9](#_Toc69754602)

[**Chapter 5: User Logins** 13](#_Toc69754603)

[**Chapter 6: Profile Page and Avatars** 20](#_Toc69754604)

[**Chapter 7: Error handling** 22](#_Toc69754605)

[**Chapter 8: Followers** 29](#_Toc69754606)

[**Chapter 9: Pagination** 38](#_Toc69754607)

[**Chapter 10: Email support** 47](#_Toc69754608)

[**Chapter 11: Facelift** 56](#_Toc69754609)

[**Chapter 12: Dates and Times** 60](#_Toc69754610)

[**Chapter 13: Internationalization and Localization (I18n and L10n)** 62](#_Toc69754611)

[**Chapter 14: Ajax** 70](#_Toc69754612)

[**Chapter 15: A Better Application Structure** 77](#_Toc69754613)

[**The end** 78](#_Toc69754614)

# **Chapter 1: Hello, World!**

Basic structure for Flask Application:

Application sẽ được nằm trong 1 package (package là 1 sub-directory có chứa file *\_\_init\_\_.py*.

File *app/\_\_init\_\_.py* sẽ bao gồm:

|  |  |
| --- | --- |
| from flask import Flask |  |
| app = Flask(\_\_name\_\_) | * Truyền vào pre-defined variable \_\_name\_\_ (sẽ được set thành tên của module khi chạy). Đây là central object của app, là starting point để load resources |
| from app import routes | * Dòng này thường được đặt ở cuối để tránh circular imports do routes module cũng sẽ import app module |

*routes* sẽ chứa tất cả các URL của app. Các *view functions* sẽ map với 1 hoặc nhiều URL để biết nên display gì khi user requests URL.

VD về view function: file *app/routes.py*

|  |  |
| --- | --- |
| from app import app |  |
| @app.route(‘/’) | * Decorator cho index() |
| @app.route(‘/index’) | * Decorator cho index() |
| def index():  return ‘Hello world!’ |  |

Để hoàn tất, cần tạo 1 python script ở top level directory để define Flask app instance: file *microblog.py*

from app import app

Và để chạy được, cần chỉ cho flask import nó như nào bằng cách setting FLASK\_APP environment variable: export FLASK\_APP=microblog.py

* Có thể để FLASK\_APP vào file *.flaskenv* để tự động set trong các lần chạy tới

# **Chapter 2: Templates**

Template thường được lưu ở folder riêng, và gọi bằng

from flask import render\_template

…

@...

def index():

return render\_template(‘template\_name.html’, title=’…’)

* render\_template() thường dung jinjja2 để thay các block {{ }} trong file html bằng các giá trị tương ứng khi chạy, bằng cách rtuyeefn arg đó vào render\_template()
* Flask sẽ mặc định tìm trong folder *‘templates’* (ở folder cùng cấp với *\_\_init\_\_.py*)

jinja2 conditional:

{% if title %}

…

{% else %}

…

{% endif %}

jinja2 loop:

{% for post in posts %}

{% endfor %}

Template inheritance:

base template: app/templates/base.html

….

<body>

{% block content %} {% endblock %}

</body>

derived template: app/templates/index.html

{% extends “base.html” %}

{% block content %}

… //something here

{% endblock %}

# **Chapter 3: Web forms**

FlaskWTF giúp handle Web Form

1. Configurations

Dùng class ở 1 separate module để lưu configuration variable (thường lưu thành file *config.py* ở top level directory)

File *config.py:*

import os

class Config(object):

SECRET\_KEY = os.environ.get(‘SECRET\_KEY’) or ‘abc123’

SECRET\_KEY là 1 phần quan trọng của mọi flask application, là crypto key để generate signatures/tokens, giúp bảo vệ tránh CSRF. Ta thường lấy nó từ environment variable từ os.environ.

Sau đó, ta chỉ cho Flask cách đọc config ở file *app/\_\_init\_\_.py:*

from flask import Flask

from config import Config

app = Flask(\_\_name\_\_)

app.config.from\_object(Config)

…

1. User login form:

Lưu các form vào *app/forms.py*. Flask có module chứa các form căn bản hay dùng:

from flask\_wtf import FlaskForm

from wtforms import StringField, PasswordField, BooleanField, SubmitField

from wtforms.validators import DataRequired

class LoginForm(FlaskForm):

username = StringField(‘Username’, validators = [DataRequired()])

….

1. Form Template

Sau đó cần tạo template cho các form. Các field StringField, … trong class LoginForm đều đã được render sẵn nên chỉ cần gọi vào trong file html

*app/templates/login.html*: Login form template

{% extends "base.html" %}

{% block content %}

<h1>Sign In</h1>

<form action="" method="post" novalidate>

{{ form.hidden\_tag() }}

<p>

{{ form.username.label }}<br>

{{ form.username(size=32) }}

</p>

<p>

{{ form.password.label }}<br>

{{ form.password(size=32) }}

</p>

<p>{{ form.remember\_me() }} {{ form.remember\_me.label }}</p>

<p>{{ form.submit() }}</p>

</form>

{% endblock %}

1 vài chú ý từ ví dụ này :

* action = ‘’ : string rỗng => form được submit đến URL ở address bar
* GET/POST:
  + GET: cho form fields vào URL (vd: /test/form.php?name=abc&password=123); có thể cache; remain trong browser history + length restriction; chỉ request data; không dung để handle sensitive data
  + POST: cho form fields vào body của HTTP request; không cached, không remain trong browser history, no length restriction
* novalidate: giúp bảo web browser không perform validation lên các fields, việc này server sẽ handle
* *{{ form.hidden\_tag() }}* vào thẻ <form></form> sẽ giúp tạo 1 field generate ra token để chống CSRF. Chỉ cần set SECRET\_KEY và them field này, Flask sẽ lo phần còn lại
* Các forms của wtforms tự render -> chỉ cần them các thẻ *{{ form.<field\_name>.label }}* vào nơi muốn để lable, và *{{ form.<field\_name>() }}* vào nơi muốn để field (có thể thêm argument vào đây, ví dụ size = 32)

1. Form views

Cuối cùng cần tạo view function cho form ở trong  *app/routes.py:*

*app/routes.py*: Login view function

from flask import render\_template

from app import app

from app.forms import LoginForm

# ...

@app.route('/login')

def login():

form = LoginForm()

return render\_template('login.html', title='Sign In', form=form)

1. Receiving form data

Để nhận data từ user, ta cần sửa lại view function trên như sau:

*app/routes.py*: Receiving login credentials

from flask import render\_template, flash, redirect

@app.route('/login', methods=['GET', 'POST'])

def login():

form = LoginForm()

if form.validate\_on\_submit():

flash('Login requested for user {}, remember\_me={}'.format(

form.username.data, form.remember\_me.data))

return redirect('/index')

return render\_template('login.html', title='Sign In', form=form)

1 vài lưu ý:

* methods ở đây nhận cả GET và POST, default sẽ chỉ nhận GET
* *form.validate\_on\_submit()* sẽ trả về False nếu request là GET => skip toàn bộ câu lệnh trong *if*
* nếu request là POST, *form.validate\_on\_submit()* sẽ lấy các data, chạy validator, nếu mọi thứ trả về True, các function tiếp theo sẽ được gọi.
* *flash()* giúp show message đến user.

Messages từ *flash()* cần được render trong template, ví dụ:

{% with messages = get\_flashed\_messages() %}

{% if messages %}

<ul>

{% for message in messages %}

<li>{{ message }}</li>

{% endfor %}

</ul>

{% endif %}

{% endwith %}

Để hiện error cho các field, ta truy cập đến

{% for error in form.username.errors %}

<span style="color: red;">[{{ error }}]</span>

{% endfor %}

Flask dung *url\_for()* để generate link đến các view function, ví dụ: *url\_for(‘login’)* trả về *‘/login’*

# **Chapter 4: Database**

1. SQLAlchemy

Flask có extension Flask\_SQLAlchemy, 1 flask wrapper cho SQLAlchemy – 1 Object Relational Mapper (ORM). ORMs cho phép app quản lý DB bằng class, object, method thay vì dung bảng và SQL. ORM sẽ translate ngôn ngữ bậc cao về DB commands.

SQLAlchemy là ORM cho nhiều relational DB: MySQL, PostgreSQL, SQLite => linh hoạt hơn

1. Database migration:

Khi structure của DB thay đổi, data đang có trong DB phải được migrate sang cấu trúc mới => flask-migrate giúp ta làm điều này.

1. Flask-SQLAlchemy Configuration

Config: cần them vào file *config.py* những dòng sau

*config.py*: Flask-SQLAlchemy configuration

import os

basedir = os.path.abspath(os.path.dirname(\_\_file\_\_))

class Config(object):

# ...

SQLALCHEMY\_DATABASE\_URI = os.environ.get('DATABASE\_URL') or \

'sqlite:///' + os.path.join(basedir, 'app.db')

SQLALCHEMY\_TRACK\_MODIFICATIONS = False

# giúp tắt tính năng signal mỗi khi sắp có thay đổi trong DB

Database sẽ được represented bằng 1 instance trong file *\_\_init\_\_.py­*. DB migration engine cũng sẽ có 1 instance.

*app/\_\_init\_\_.py*: Flask-SQLAlchemy and Flask-Migrate initialization

from flask import Flask

from config import Config

from flask\_sqlalchemy import SQLAlchemy

from flask\_migrate import Migrate

app = Flask(\_\_name\_\_)

app.config.from\_object(Config)

db = SQLAlchemy(app)

migrate = Migrate(app, db)

from app import routes, models

1. Database Models

Data sẽ được lưu dưới dạng collection of classes => ORM translate thành bảng.

Ta tạo file *app/models.py:*

*app/models.py*: User database model

from app import db

class User(db.Model):

id = db.Column(db.Integer, primary\_key=True)

username = db.Column(db.String(64), index=True, unique=True)

email = db.Column(db.String(120), index=True, unique=True)

password\_hash = db.Column(db.String(128))

def \_\_repr\_\_(self):

return '<User {}>'.format(self.username)

1. Creating Migration Repository

Khi app phát triển => ta thường phải thêm, sửa, xóa item trong DB

* Alembic (migration framework của Flask-Migrate) giúp ta làm điều này.

Alembic maintains 1 migration repository (1 directory chứa scripts). Mỗi khi có thay đổi đến database schema, 1 script được thêm vào repo, với detail về thay đổi đó. Để apply migration, các scripts sẽ được execute theo thứ tự được tạo.

Để tạo migration repo, chạy câu lệnh *flask db init* trong terminal. Điều này sẽ tạo ra 1 folder migrations.

1. First DB Migration:

Có 2 cách để tạo 1 DB Migration: manually / automatically.

Để migrate automatically, Alembic sẽ so sánh DB schema định nghĩa bởi models, và DB schema hiện đang có trong DB, rồi nó sẽ chạy script cần thiết để thực hiện thay đổi. Khi chưa có DB nào, nó sẽ tạo mới hoàn toàn.

*(venv): flask db migrate -m* (-m cho short description)

Khi đó, trong thư mục migrations sẽ có 1 script miêu tả quá trình này. Nếu muốn chạy script đó, ta dung lệnh

*(venv): flask db upgrade*

Lưu ý: Nếu làm việc với MySQL, PostgreSQL thì ta cần tạo DB trong server trước khi upgrade.

Muốn downgrade về migration trước, ta dùng lệnh:

*(venv) flask db downgrade*

1. DB relationship

Với model như sau:

Diagram

Description automatically generated with medium confidence

Ta có file *app/models.py:*

*app/models.py*: Posts database table and relationship

from datetime import datetime

from app import db

class User(db.Model):

id = db.Column(db.Integer, primary\_key=True)

username = db.Column(db.String(64), index=True, unique=True)

email = db.Column(db.String(120), index=True, unique=True)

password\_hash = db.Column(db.String(128))

posts = db.relationship('Post', backref='author', lazy='dynamic')

def \_\_repr\_\_(self):

return '<User {}>'.format(self.username)

class Post(db.Model):

id = db.Column(db.Integer, primary\_key=True)

body = db.Column(db.String(140))

timestamp = db.Column(db.DateTime, index=True, default=datetime.utcnow)

user\_id = db.Column(db.Integer, db.ForeignKey('user.id'))

def \_\_repr\_\_(self):

return '<Post {}>'.format(self.body)

Trong đó: *db.relationship()* và *db.ForeignKey()* giúp miêu tả Foreign Key relationship.

1. Flask Shell

Flask shell giúp chạy interpreter ngay trong context của app => không cần import.

*(venv): flask shell*

Ta có thể thêm function vào *microblog.py* để tạo shell context và add các pre-import symbols.

from app import app, db

from app.models import User, Post

@app.shell\_context\_processor

def make\_shell\_context():

return {'db': db, 'User': User, 'Post': Post}

Khi ta chạy *flask shell*, các instance như db, User, Post có thể được dùng trực tiếp và không cần import.

# **Chapter 5: User Logins**

1. Password hashing

Pasword hashing hay dùng package của Werkzeug

>>> from werkzeug.security import generate\_password\_hash

>>> hash = generate\_password\_hash('foobar')

>>> hash

'pbkdf2:sha256:50000$vT9fkZM8$04dfa35c6476acf7e788a1b5b3c35e217c78dc04539d295f011f01f18cd2175f'

>>> from werkzeug.security import check\_password\_hash

>>> check\_password\_hash(hash, 'foobar')

True

>>> check\_password\_hash(hash, 'barfoo')

False

Tools này sẽ tự động thêm thông tin vào để 2 chuỗi giống nhau sẽ luôn có hash khác nhau.

Quá trình hashing password có thể được implemented như methods trong user model:

*app/models.py*: Password hashing and verification

from werkzeug.security import generate\_password\_hash, check\_password\_hash

# ...

class User(db.Model):

# ...

def set\_password(self, password):

self.password\_hash = generate\_password\_hash(password)

def check\_password(self, password):

return check\_password\_hash(self.password\_hash, password)

1. Flask-Login

Flask-Login là flask extension giúp manage user logged-in state, khi users log in vào application và navigate đến các page khác nhau, thì application sẽ remember trạng thái logged in của user. Flask-Login cần được initialized trong *app/\_\_init\_\_.py*:

*app/\_\_init\_\_.py*: Flask-Login initialization

# ...

from flask\_login import LoginManager

app = Flask(\_\_name\_\_)

# ...

login = LoginManager(app)

# ...

1. User Model for Flask-Login

Flask-Login làm việc với application’s user model, và expects 1 vài properties và methods cần được implemented trong user model:

* is\_authenticated: property trả về True nếu user có valid credentials
* is\_active: property trả về True nếu user account đang active
* is\_anonymous: property trả về False với regular users, True cho special, anonymous user
* get\_id(): method trả về unique ID cho user dưới dạng string

Flask-Login có *UserMixin* class đã bao gồm sẵn generic implementations cho 4 methods này:

*app/models.py*: Flask-Login user mixin class

# ...

from flask\_login import UserMixin

class User(UserMixin, db.Model):

# ...

1. User loader function

Flask-Login keeps track của logged-in user bằng cách lưu ID của nó vào Flask’s user session, 1 storage dành riêng cho user connects to the application. Vì Flask-Login không liên kết đến DB, nên nó cần application nhận ID và load user giúp nó. VD trong file *app/models.py*:

*app/models.py*: Flask-Login user loader function

from app import login

# ...

@login.user\_loader

def load\_user(id):

return User.query.get(int(id))

1. Logging Users In

Giờ ta có thể hoàn thiện login function trong *app/routes.py*

*app/routes.py*: Login view function logic

from flask\_login import current\_user, login\_user

from app.models import User

# ...

@app.route('/login', methods=['GET', 'POST'])

def login():

if current\_user.is\_authenticated:

return redirect(url\_for('index'))

form = LoginForm()

if form.validate\_on\_submit():

user = User.query.filter\_by(username=form.username.data).first()

if user is None or not user.check\_password(form.password.data):

flash('Invalid username or password')

return redirect(url\_for('login'))

login\_user(user, remember=form.remember\_me.data)

return redirect(url\_for('index'))

return render\_template('login.html', title='Sign In', form=form)

* *current\_user* từ *flask\_login* và có thể dùng để lấy user object that represents the client of the request.

1. Logging Users Out

*app/routes.py*: Logout view function

# ...

from flask\_login import logout\_user

# ...

@app.route('/logout')

def logout():

logout\_user()

return redirect(url\_for('index'))

1. Requiring Users to Login

Ta có thể bắt users phải login trước khi được xem nội dung của page. Đầu tiên phải thêm vào file *\_\_init\_\_.py* view function nào handles logins:

# ...

login = LoginManager(app)

login.login\_view = 'login'

* *‘login’* sẽ phụ trách việc login

*app/routes.py*: @login\\_required decorator

from flask\_login import login\_required

@app.route('/')

@app.route('/index')

@login\_required

def index():

Sau đó chỉ cần thêm decorator *@login\_required* vào trước mỗi page được bảo vệ. Khi một user chưa log in và truy cập vào trang */index*, *@login\_required* sẽ redirect về login page, nhưng nó sẽ thêm 1 vài thông tin để application có thể quay lại trang này sau khi log in. VD *@login\_required* sẽ add 1 query string vào URL: */login?next=/index*.

Code để xử lý *next* query string argument:

*app/routes.py*: Redirect to "next" page

from flask import request

from werkzeug.urls import url\_parse

@app.route('/login', methods=['GET', 'POST'])

def login():

# ...

if form.validate\_on\_submit():

user = User.query.filter\_by(username=form.username.data).first()

if user is None or not user.check\_password(form.password.data):

flash('Invalid username or password')

return redirect(url\_for('login'))

login\_user(user, remember=form.remember\_me.data)

next\_page = request.args.get('next')

if not next\_page or url\_parse(next\_page).netloc != '':

next\_page = url\_for('index')

return redirect(next\_page)

# ...

Sau khi user logged in bằng cách gọi function *login\_user(),* value của *next* query string sẽ được cho vào biến *next\_page.* Flask cung cấp 1 biến *request*, chứa toàn bộ information mà client gửi cùng request. *request.args* thể hiện dữ liệu này dưới dạng dict. Có 3 trường hợp xảy ra sau khi user log in:

* nếu login URL không chứa *next* argument, thì user được redirect về index
* nếu URL chứa *next* được set đến relative path (URL mà không có domain portion), thì user được redirect đến URL đó
* nếu URL chứa *next* argument được set đến full URL, redirect về index.

Trường hợp thứ 3 là để tránh attacker insert URL vào web độc sau next argument. Để check URL là relative hay absolute, ta dùng *url\_parse()* của Werkzeug và check thành phần *netloc* của nó.

1. User Registration

Tạo user registration form trong *app/forms.py:*

*app/forms.py*: User registration form

from flask\_wtf import FlaskForm

from wtforms import StringField, PasswordField, BooleanField, SubmitField

from wtforms.validators import ValidationError, DataRequired, Email, EqualTo

from app.models import User

# ...

class RegistrationForm(FlaskForm):

username = StringField('Username', validators=[DataRequired()])

email = StringField('Email', validators=[DataRequired(), Email()])

password = PasswordField('Password', validators=[DataRequired()])

password2 = PasswordField(

'Repeat Password', validators=[DataRequired(), EqualTo('password')])

submit = SubmitField('Register')

def validate\_username(self, username):

user = User.query.filter\_by(username=username.data).first()

if user is not None:

raise ValidationError('Please use a different username.')

def validate\_email(self, email):

user = User.query.filter\_by(email=email.data).first()

if user is not None:

raise ValidationError('Please use a different email address.')

1 vài lưu ý:

* *Email()* giúp check string nhập vào có theo format của email không
* *EqualTo()* giúp check password nhập lần 2 có giống lần 1 không
* 2 hàm *validate\_username()* và *validate\_email()* giúp kiểm tra xem username và email có bị trùng với dữ liệu trong DB không

Sau đó, ta tạo file template .html, và thêm view function trong *app/routes.py*

*app/routes.py*: User registration view function

from app import db

from app.forms import RegistrationForm

# ...

@app.route('/register', methods=['GET', 'POST'])

def register():

if current\_user.is\_authenticated:

return redirect(url\_for('index'))

form = RegistrationForm()

if form.validate\_on\_submit():

user = User(username=form.username.data, email=form.email.data)

user.set\_password(form.password.data)

db.session.add(user)

db.session.commit()

flash('Congratulations, you are now a registered user!')

return redirect(url\_for('login'))

return render\_template('register.html', title='Register', form=form)

# **Chapter 6: Profile Page and Avatars**

1. jinja2 Sub-Templates

Ta có thể tạo 1 sub-templates, ví dụ *\_post.html* (\_ là naming convention cho sub-template), và include nó bằng cách

{ % include ‘\_post.html’ % }

1. Recording the last visit time for a user

Để record được lần cuối user vào web, ta cần 1 view function luôn chạy trước mỗi khi các view function khác được gọi. Để làm được điều này, ta dùng decorator *@app.before\_request*

*app/routes.py*: Record time of last visit

from datetime import datetime

@app.before\_request

def before\_request():

if current\_user.is\_authenticated:

current\_user.last\_seen = datetime.utcnow()

db.session.commit()

1. Profile Editor

Đầu tiên là tạo form để edit profile:

*app/forms.py*: Profile editor form

from wtforms import StringField, TextAreaField, SubmitField

from wtforms.validators import DataRequired, Length

# ...

class EditProfileForm(FlaskForm):

username = StringField('Username', validators=[DataRequired()])

about\_me = TextAreaField('About me', validators=[Length(min=0, max=140)])

submit = SubmitField('Submit')

Ở đây ta dùng validator mới: *Length()* để kiểm tra độ dài.

# **Chapter 7: Error handling**

VD khi sửa tên 1 user thành tên của 1 user khác đã có sẵn trong DB -> sẽ tạo ra Internal Server Error (code: 500), do trong SQL Alchemy đã để *username* column có *unique=True*.

1. **Flask Debug Mode**

Để chạy Flask với debug mode on, ta thêm biến *FLASK\_DEBUG=1* vào file *.flaskenv*. Khi đó nếu có lỗi, browser sẽ hiển thị ra debugger cùng chi tiết về lỗi đó.

1. **Custom Error Pages**

Flask cho phép tự tạo custom error pages. Ví dụ, ta sẽ design 2 custom error pages cho 2 lỗi 404 và 500 bằng *@errorhandler* vào file *app/errors.py*

*app/errors.py*: Custom error handlers

from flask import render\_template

from app import app, db

@app.errorhandler(404)

def not\_found\_error(error):

return render\_template('404.html'), 404

@app.errorhandler(500)

def internal\_error(error):

db.session.rollback()

return render\_template('500.html'), 500

Error functions hoạt động tương tự view functions. View functions thường trả về 2 giá trị, trong các view function trước nó trả về 200 (success response) nên không handle, ở đây ta lấy 2 giá trị 404 và 500 từ error function. Với lỗi 500, do liên quan đến DB, nên ta sẽ rollback session về clean state để đảm bảo an toàn cho DB.

Sau đó cần tạo template cho 2 file *404.html* và *500.html*

*app/templates/404.html*: Not found error template

{% extends "base.html" %}

{% block content %}

<h1>File Not Found</h1>

<p><a href="{{ url\_for('index') }}">Back</a></p>

{% endblock %}

*app/templates/500.html*: Internal server error template

{% extends "base.html" %}

{% block content %}

<h1>An unexpected error has occurred</h1>

<p>The administrator has been notified. Sorry for the inconvenience!</p>

<p><a href="{{ url\_for('index') }}">Back</a></p>

{% endblock %}

Và để register các error handlers với Flask, ta import nó vào file *app/\_\_init\_\_.py*.

*app/\_\_init\_\_.py*: Import error handlers

# ...

from app import routes, models, errors

1. **Sending errors by Email**

Vấn đề với error handling của flask là nó sẽ hiện ra terminal => phải có người thường xuyên quan sát nó => không hợp với thực tiễn.

Phương pháp đầu tiên cho vấn đề này là gửi error đến mail.

Bước 1 là add email server vào config file.

*config.py*: Email configuration

class Config(object):

# ...

MAIL\_SERVER = os.environ.get('MAIL\_SERVER')

MAIL\_PORT = int(os.environ.get('MAIL\_PORT') or 25)

MAIL\_USE\_TLS = os.environ.get('MAIL\_USE\_TLS') is not None

MAIL\_USERNAME = os.environ.get('MAIL\_USERNAME')

MAIL\_PASSWORD = os.environ.get('MAIL\_PASSWORD')

ADMINS = ['your-email@example.com']

*ADMINS* là danh sách các email sẽ nhận được error reports.

Flask sử dụng Python’s logging package để log, và package này có sẵn tính năng gửi log theo email.. Ta chỉ cần thêm 1 *SMTPHandler* instance để gửi error qua email.

*app/\_\_init\_\_.py*: Log errors by email

import logging

from logging.handlers import SMTPHandler

# ...

if not app.debug:

if app.config['MAIL\_SERVER']:

auth = None

if app.config['MAIL\_USERNAME'] or app.config['MAIL\_PASSWORD']:

auth = (app.config['MAIL\_USERNAME'], app.config['MAIL\_PASSWORD'])

secure = None

if app.config['MAIL\_USE\_TLS']:

secure = ()

mail\_handler = SMTPHandler(

mailhost=(app.config['MAIL\_SERVER'], app.config['MAIL\_PORT']),

fromaddr='no-reply@' + app.config['MAIL\_SERVER'],

toaddrs=app.config['ADMINS'], subject='Microblog Failure',

credentials=auth, secure=secure)

mail\_handler.setLevel(logging.ERROR)

app.logger.addHandler(mail\_handler)

1 vài chú ý:

* email logger chỉ hoạt động khi debug mode đã tắt.
* về cơ bản, đoạn code trên sẽ tạo 1 *SMTPHandler* instance, set level để nó chỉ gửi error, không gửi warnings, và attach nó vào *app.logger* object của Flask.
* log level để ở ERROR, để chỉ gửi về ERROR, không gửi warning.

**PYTHON LOGGING LEVELS:**

| **Level** | **Numeric value** |
| --- | --- |
| CRITICAL | 50 |
| ERROR | 40 |
| WARNING | 30 |
| INFO | 20 |
| DEBUG | 10 |
| NOTSET | 0 |

Để test, mở 1 terminal khác và chạy

(venv) $ python -m smtpd -n -c DebuggingServer localhost:8025

và thêm 2 biến *MAIL\_SERVER=localhost* và *MAIL\_PORT=8025* vào file *.flashenv.* Khi đó, terminal thứ 2 sẽ là fake email server, và print nội dung của email đó ra console.

Hoặc setup real email server. VD với GMAIL:

export MAIL\_SERVER=smtp.googlemail.com

export MAIL\_PORT=587

export MAIL\_USE\_TLS=1

export MAIL\_USERNAME=<your-gmail-username>

export MAIL\_PASSWORD=<your-gmail-password>

Gmail có thể sẽ chặn do các app này less secure.

1. **Logging to a file**

Một cách khác là duy trì 1 log file cho application. Để enable file based log, 1 handler khác thuộc kiểu *RotatingFileHandler* cần được attached vào application logger, theo cách tương tự.

*app/\_\_init\_\_.py*: Logging to a file

# ...

from logging.handlers import RotatingFileHandler

import os

# ...

if not app.debug:

# ...

if not os.path.exists('logs'):

os.mkdir('logs')

file\_handler = RotatingFileHandler('logs/microblog.log', maxBytes=10240,

backupCount=10)

file\_handler.setFormatter(logging.Formatter(

'%(asctime)s %(levelname)s: %(message)s [in %(pathname)s:%(lineno)d]'))

file\_handler.setLevel(logging.INFO)

app.logger.addHandler(file\_handler)

app.logger.setLevel(logging.INFO)

app.logger.info('Microblog startup')

1 vài lưu ý:

* Trong VD này, ta tạo 1 file *microblog.log* trong thư mục *logs*
* *RotatingFileHandler* sẽ rotates các logs, đảm bảo log files không quá lớn khi ứng dụng chạy đã lâu. Trong VD này ta set max size của file là 10 KB, và sẽ giữ lại 10 files log mới nhất.
* *logging.Formatter* cung cấp custom formatting cho log file. Trong VD này ta dùng format bao gồm timestamp, logging level, message, source file, line number.
* để thêm nhiều info vào log, ta hạ log level xuống *INFO* category, cả ở trong application logger và file logger handler.
* khi ta bắt đầu chạy server, 1 dòng sẽ được thêm vào log với thời gian server starts/restarts.

1. **Fixing the Duplicate Username Bug**

Bây giờ ta sẽ thêm hàm validate username cho Edit Profile để tránh lỗi trên. Cần chú ý là ta chỉ check nếu username đã được sửa so với original.

*app/forms.py*: Validate username in edit profile form.

class EditProfileForm(FlaskForm):

username = StringField('Username', validators=[DataRequired()])

about\_me = TextAreaField('About me', validators=[Length(min=0, max=140)])

submit = SubmitField('Submit')

def \_\_init\_\_(self, original\_username, \*args, \*\*kwargs):

super(EditProfileForm, self).\_\_init\_\_(\*args, \*\*kwargs)

self.original\_username = original\_username

def validate\_username(self, username):

if username.data != self.original\_username:

user = User.query.filter\_by(username=self.username.data).first()

if user is not None:

raise ValidationError('Please use a different username.')

Trong đoạn code trên có 1 overloaded constructor, nhận original username làm argument..

Để dùng validation method này, ta cần thêm original username argument vào view function:

*app/routes.py*: Validate username in edit profile form.

@app.route('/edit\_profile', methods=['GET', 'POST'])

@login\_required

def edit\_profile():

form = EditProfileForm(current\_user.username)

# ...

# **Chapter 8: Followers**

Chapter này sẽ giúp user này follow user khác, tức là sẽ phải extend DB để track xem ai đang follow ai.

1. **Representing Followers**

Followers model sẽ là dạng many-to-many self-referential relationship (class linked đến instance khác của cùng class đó). Bảng followers sẽ là association table của relationship này.

Diagram

Description automatically generated

1. **Database Model Representation**

Bước đầu là tạo association table *followers:*

*app/models.py*: Followers association table

followers = db.Table('followers',

db.Column('follower\_id', db.Integer, db.ForeignKey('user.id')),

db.Column('followed\_id', db.Integer, db.ForeignKey('user.id'))

)

Do đây chỉ là 1 association table với các giá trị toàn là foreign key, nên ta không tạo nó thành 1 class riêng.

Sau đó ta declare many-to-many relationship trong bảng *Users*

*app/models.py*: Many-to-many followers relationship

class User(UserMixin, db.Model):

# ...

followed = db.relationship(

'User', secondary=followers,

primaryjoin=(followers.c.follower\_id == id),

secondaryjoin=(followers.c.followed\_id == id),

backref=db.backref('followers', lazy='dynamic'), lazy='dynamic')

1 vài lưu ý:

* Trong này ta link *User* với chính nó, và left side user đang follow right side user. Ta đặt tên nó là *followed* vì khi execute query này từ left side, ta sẽ nhận được list followed users (tức là những user ở right side).
* *secondary=followers* giúp configure association table cho relationship này, đã được định nghĩa ở trên.
* *primaryjoin* chỉ ra condition links với left side entity (follower user) với association table. Condition này là nếu user ID match với *follower\_id* trong bảng association table (lấy từ *followers.c.follower\_id)*
* *secondaryjoin* chỉ ra condition links với right side entity (followed user) với association table, nếu User ID ở right side match với *followed\_id* trong bảng đó (lấy từ *followers.c.followed\_id)*
* *backref* chỉ ra cách right side entity nhìn relationship này. Từ left side góc nhìn là *followed,* nên ta để từ right side sẽ nhìn thành *follower. lazy* argument chỉ ra execution mode cho query, *dynamic* giúp set up query chỉ chạy khi được requested.
* *lazy* ở ngoài tương tự như trên, nhưng apply cho left side query thay vì right side.

Sau đó ta chạy 2 lệnh *flask db migrate -m “followers”* và *flask db upgrade* để thực hiện thay đổi.

1. **Adding and Removing “follows”**

Ta tiếp tục tạo methods cho class Users giúp unfollow/follow bằng cách dùng *append()* và *remove()*:

*app/models.py*: Add and remove followers

class User(UserMixin, db.Model):

#...

def follow(self, user):

if not self.is\_following(user):

self.followed.append(user)

def unfollow(self, user):

if self.is\_following(user):

self.followed.remove(user)

def is\_following(self, user):

return self.followed.filter(

followers.c.followed\_id == user.id).count() > 0

1. **Obtaining the Posts from Followed Users and himself**

Để hiện post của những người user đang follow và của chính bản thân user, ta dùng query sau:

*app/models.py*: Followed posts query with user's own posts.

def followed\_posts(self):

followed = Post.query.join(

followers, (followers.c.followed\_id == Post.user\_id)).filter(

followers.c.follower\_id == self.id)

own = Post.query.filter\_by(user\_id=self.id)

return followed.union(own).order\_by(Post.timestamp.desc())

Trong đó:

* *join()* giúp lọc ra post của tất cả các user đang được follow
* *filter()* giúp lọc ra những post của người đang được chính user đó (self) follow
* *union()* giúp show cả post của bản thân
* *desc()* giúp short theo thời gian.

1. **Unit Testing the User Model**

Một vấn đề với các complex feature là ta phải đảm bảo nó sẽ tiếp tục hoạt động trong tương lai, khi ta thay đổi các phần khác của ứng dụng. 1 Cách để đảm bảo viejc này là tạo ra 1 dãy các automated tests mà ta có thể re-run mỗi khi có thay đổi.

Python có package *unittest* giúp ta thực hiện việc unit test này.

*tests.py*: User model unit tests.

from datetime import datetime, timedelta

import unittest

from app import app, db

from app.models import User, Post

class UserModelCase(unittest.TestCase):

def setUp(self):

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite://'

db.create\_all()

def tearDown(self):

db.session.remove()

db.drop\_all()

def test\_password\_hashing(self):

u = User(username='susan')

u.set\_password('cat')

self.assertFalse(u.check\_password('dog'))

self.assertTrue(u.check\_password('cat'))

def test\_avatar(self):

u = User(username='john', email='john@example.com')

self.assertEqual(u.avatar(128), ('https://www.gravatar.com/avatar/'

'd4c74594d841139328695756648b6bd6'

'?d=identicon&s=128'))

def test\_follow(self):

u1 = User(username='john', email='john@example.com')

u2 = User(username='susan', email='susan@example.com')

db.session.add(u1)

db.session.add(u2)

db.session.commit()

self.assertEqual(u1.followed.all(), [])

self.assertEqual(u1.followers.all(), [])

u1.follow(u2)

db.session.commit()

self.assertTrue(u1.is\_following(u2))

self.assertEqual(u1.followed.count(), 1)

self.assertEqual(u1.followed.first().username, 'susan')

self.assertEqual(u2.followers.count(), 1)

self.assertEqual(u2.followers.first().username, 'john')

u1.unfollow(u2)

db.session.commit()

self.assertFalse(u1.is\_following(u2))

self.assertEqual(u1.followed.count(), 0)

self.assertEqual(u2.followers.count(), 0)

def test\_follow\_posts(self):

# create four users

u1 = User(username='john', email='john@example.com')

u2 = User(username='susan', email='susan@example.com')

u3 = User(username='mary', email='mary@example.com')

u4 = User(username='david', email='david@example.com')

db.session.add\_all([u1, u2, u3, u4])

# create four posts

now = datetime.utcnow()

p1 = Post(body="post from john", author=u1,

timestamp=now + timedelta(seconds=1))

p2 = Post(body="post from susan", author=u2,

timestamp=now + timedelta(seconds=4))

p3 = Post(body="post from mary", author=u3,

timestamp=now + timedelta(seconds=3))

p4 = Post(body="post from david", author=u4,

timestamp=now + timedelta(seconds=2))

db.session.add\_all([p1, p2, p3, p4])

db.session.commit()

# setup the followers

u1.follow(u2) # john follows susan

u1.follow(u4) # john follows david

u2.follow(u3) # susan follows mary

u3.follow(u4) # mary follows david

db.session.commit()

# check the followed posts of each user

f1 = u1.followed\_posts().all()

f2 = u2.followed\_posts().all()

f3 = u3.followed\_posts().all()

f4 = u4.followed\_posts().all()

self.assertEqual(f1, [p2, p4, p1])

self.assertEqual(f2, [p2, p3])

self.assertEqual(f3, [p3, p4])

self.assertEqual(f4, [p4])

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main(verbosity=2)

Ta có thể chạy test này bằng *python tests.py*

1. **Integrating Followers with the Application**

Giờ ta sẽ thêm tính năng followers này vào application. Để thêm tính năng follow/unfollow, ta dùng POST request, form chứa nút này sẽ không có data fields nào (chỉ có field chống CSRF).

*app/forms.py*: Empty form for following and unfollowing.

class EmptyForm(FlaskForm):

submit = SubmitField('Submit')

Và thêm view function:

*app/routes.py*: Follow and unfollow routes.

from app.forms import EmptyForm

# ...

@app.route('/follow/<username>', methods=['POST'])

@login\_required

def follow(username):

form = EmptyForm()

if form.validate\_on\_submit():

user = User.query.filter\_by(username=username).first()

if user is None:

flash('User {} not found.'.format(username))

return redirect(url\_for('index'))

if user == current\_user:

flash('You cannot follow yourself!')

return redirect(url\_for('user', username=username))

current\_user.follow(user)

db.session.commit()

flash('You are following {}!'.format(username))

return redirect(url\_for('user', username=username))

else:

return redirect(url\_for('index'))

@app.route('/unfollow/<username>', methods=['POST'])

@login\_required

def unfollow(username):

form = EmptyForm()

if form.validate\_on\_submit():

user = User.query.filter\_by(username=username).first()

if user is None:

flash('User {} not found.'.format(username))

return redirect(url\_for('index'))

if user == current\_user:

flash('You cannot unfollow yourself!')

return redirect(url\_for('user', username=username))

current\_user.unfollow(user)

db.session.commit()

flash('You are not following {}.'.format(username))

return redirect(url\_for('user', username=username))

else:

return redirect(url\_for('index'))

Khác với 2 forms login và edit profile, 2 form mới này không có pages của riêng nó, nó sẽ được render bằng *user()* route và xuất hiện trên page của *user()*. Sau đó, ta chỉ cần instantiate *EmptyForm* object trong class route đến /*user* và thêm vào trong file *user.html*

# **Chapter 9: Pagination**

Ở chapter này application sẽ accept blog post từ user, và hiển thị chúng ở home page và profile page

1. **Submission of Blog posts**

Đầu tiên ta tạo 1 form để user có thể type new posts:

*app/forms.py*: Blog submission form.

class PostForm(FlaskForm):

post = TextAreaField('Say something', validators=[

DataRequired(), Length(min=1, max=140)])

submit = SubmitField('Submit')

Và thêm nó vào file html

*app/templates/index.html*: Post submission form in index template

{% extends "base.html" %}

{% block content %}

<h1>Hi, {{ current\_user.username }}!</h1>

<form action="" method="post">

{{ form.hidden\_tag() }}

<p>

{{ form.post.label }}<br>

{{ form.post(cols=32, rows=4) }}<br>

{% for error in form.post.errors %}

<span style="color: red;">[{{ error }}]</span>

{% endfor %}

</p>

<p>{{ form.submit() }}</p>

</form>

{% for post in posts %}

<p>

{{ post.author.username }} says: <b>{{ post.body }}</b>

</p>

{% endfor %}

{% endblock %}

Cuối cùng là tạo view function cho nó:

*app/routes.py*: Post submission form in index view function.

from app.forms import PostForm

from app.models import Post

@app.route('/', methods=['GET', 'POST'])

@app.route('/index', methods=['GET', 'POST'])

@login\_required

def index():

form = PostForm()

if form.validate\_on\_submit():

post = Post(body=form.post.data, author=current\_user)

db.session.add(post)

db.session.commit()

flash('Your post is now live!')

return redirect(url\_for('index'))

posts = current\_user.followed\_posts().all()

return render\_template("index.html", title='Home Page', form=form,

posts=posts)

return render\_template("index.html", title='Home Page', form=form,

posts=posts)

Lưu ý: Sau khi process form data, browser sẽ redirect về lại homepage. Ta có thể bỏ qua bước redirect này, và cho function tiếp tục chạy xuống dưới, nhưng đây là 1 best practice để respond lại 1 *POST* request, bởi vì điều này giúp chúng ta tránh được rắc rối khi refresh command được thực thi trong web browser. Khi web browser refresh, chúng sẽ thực thi lại last request, vậy nên khi *POST* request đi kèm với form submission không được redirect, thì khi refresh ta sẽ submit lại form => browser sẽ hỏi user confirm duplicate submission. Nếu ta redirect về lại homepage, thì khi này nó sẽ chuyển thành *GET* request, và khi refresh mọi thứ sẽ bình thường.

* đây gọi là POST/REDIRECT/GET pattern

Diagram

Description automatically generated

Hình này thể hiện double *POST* problem

Diagram

Description automatically generated

Double *POST* solved bằng POST/REQUEST/GET

1. **Explore page**

Giờ chúng ta tạo 1 explore page để giúp các User có thể tìm thấy nhau và follow dễ dàng hơn.

Explore page sẽ có giao diện tương tự index, nhưng sẽ hiển thị post của tất cả các user.

View function như sau:

*app/routes.py*: Explore view function.

@app.route('/explore')

@login\_required

def explore():

posts = Post.query.order\_by(Post.timestamp.desc()).all()

return render\_template('index.html', title='Explore', posts=posts)

Và để prevent index crashing khi không cố render web form, ta thêm điều kiện if vào cho nó:

*app/templates/index.html*: Make the blog post submission form optional.

{% extends "base.html" %}

{% block content %}

<h1>Hi, {{ current\_user.username }}!</h1>

{% if form %}

<form action="" method="post">

...

</form>

{% endif %}

...

{% endblock %}

Ta cũng sẽ sửa file *\_post.html* để nó show username của blog post là 1 đường link

*app/templates/index.html*: Make the blog post submission form optional.

{% extends "base.html" %}

{% block content %}

<h1>Hi, {{ current\_user.username }}!</h1>

{% if form %}

<form action="" method="post">

...

</form>

{% endif %}

...

{% endblock %}

Sau đó include sub-template này ở file *index.html*

*app/templates/index.html*: Use blog post sub-template.

...

{% for post in posts %}

{% include '\_post.html' %}

{% endfor %}

...

1. **Pagination**

Pagination giúp giải quyết vấn đề khi 1 user có quá nhiều post ở Home page.

Flask\_SQLAlchemy hỗ trợ pagination với query *paginate()*. VD nếu muốn hiển thị first 20 posts:

>>> user.followed\_posts().paginate(1, 20, False).items

3 arguments của *paginate():*

* page number, bắt đầu từ 1
* số items per page
* error flag: nếu set là *True*, khi out of range sẽ trả về 404 error, nếu set là *False* sẽ trả về 1 empty list

Đầu tiên, ta nên chỉ định số post trong mỗi page ở file *config.py*

*config.py*: Posts per page configuration.

class Config(object):

# ...

POSTS\_PER\_PAGE = 3

Ta sẽ dùng query string argument để chọn page number:

* Page 1, implicit: http://localhost:5000/index
* Page 1, explicit: http://localhost:5000/index?page=1
* Page 3: <http://localhost:5000/index?page=3>

Để lấy arguments ở query string, ta dùng Flask’s *request.args*. View function sẽ như sau :

*app/routes.py*: Followers association table

@app.route('/', methods=['GET', 'POST'])

@app.route('/index', methods=['GET', 'POST'])

@login\_required

def index():

# ...

page = request.args.get('page', 1, type=int)

posts = current\_user.followed\_posts().paginate(

page, app.config['POSTS\_PER\_PAGE'], False)

return render\_template('index.html', title='Home', form=form,

posts=posts.items)

@app.route('/explore')

@login\_required

def explore():

page = request.args.get('page', 1, type=int)

posts = Post.query.order\_by(Post.timestamp.desc()).paginate(

page, app.config['POSTS\_PER\_PAGE'], False)

return render\_template("index.html", title='Explore', posts=posts.items)

1. **Page Navigation**

*paginate()* trả về 1 object thuộc class *Pagination,* và ngoài attribute items, thì còn các attributes khác sau:

* has\_next: True nếu còn ít nhất 1 page sau page hiện tại
* has\_prev: True nếu còn ít nhất 1 page trước page hiện tại
* next\_num: page number của next page
* prev\_num: page number của prev page.

Ta có thể sửa lại view functions để thêm page navigation như sau:

@app.route('/', methods=['GET', 'POST'])

@app.route('/index', methods=['GET', 'POST'])

@login\_required

def index():

# ...

page = request.args.get('page', 1, type=int)

posts = current\_user.followed\_posts().paginate(

page, app.config['POSTS\_PER\_PAGE'], False)

next\_url = url\_for('index', page=posts.next\_num) \

if posts.has\_next else None

prev\_url = url\_for('index', page=posts.prev\_num) \

if posts.has\_prev else None

return render\_template('index.html', title='Home', form=form,

posts=posts.items, next\_url=next\_url,

prev\_url=prev\_url)

@app.route('/explore')

@login\_required

def explore():

page = request.args.get('page', 1, type=int)

posts = Post.query.order\_by(Post.timestamp.desc()).paginate(

page, app.config['POSTS\_PER\_PAGE'], False)

next\_url = url\_for('explore', page=posts.next\_num) \

if posts.has\_next else None

prev\_url = url\_for('explore', page=posts.prev\_num) \

if posts.has\_prev else None

return render\_template("index.html", title='Explore', posts=posts.items,

next\_url=next\_url, prev\_url=prev\_url)

1 vài lưu ý:

* *next\_url* và *prev\_url* chỉ được set nếu có page theo direction đó.
* *url\_for()* có thể thêm được các argument tùy ý, và nếu argument đó không được referenced trực tiếp trong URL, thì Flask sẽ coi nó là query arguments => URL sẽ có dạng */explore?page=2*

Cuối cùng, ta chỉ cần sửa file html để hiển thị 2 nút prev và next.

Tương tự, ta thêm pagination vào User Profile Page.

# **Chapter 10: Email support**

Chapter này sẽ thêm password reset feature thông qua email.

Flask có extension Flask-Mail giúp gửi email, phải install qua *pip insstall flask-mail*

Password reset link sẽ phải có secure token, và để generate chúng ta dùng *JSON Web Tokens: pip install pyjwt*

Như mọi extension khác, ta phải khởi tạo instance cho nó trong *\_\_init\_\_.py*

*app/\_\_init\_\_.py*: Flask-Mail instance.

# ...

from flask\_mail import Mail

app = Flask(\_\_name\_\_)

# ...

mail = Mail(app)

1. **Simple Email Framework**

*app/email.py*: Email sending wrapper function.

from flask\_mail import Message

from app import mail

def send\_email(subject, sender, recipients, text\_body, html\_body):

msg = Message(subject, sender=sender, recipients=recipients)

msg.body = text\_body

msg.html = html\_body

mail.send(msg)

1. **Requesting a Password Reset:**

Đầu tiên, ta thêm 1 link để request password vào trang *‘login.html’*.

Tiếp theo, ta tạo class cho form reset password trong *app/forms.py*:

*app/forms.py*: Reset password request form.

class ResetPasswordRequestForm(FlaskForm):

email = StringField('Email', validators=[DataRequired(), Email()])

submit = SubmitField('Request Password Reset')

Và tạo file html tương ứng:

*app/templates/reset\_password\_request.html*: Reset password request template.

{% extends "base.html" %}

{% block content %}

<h1>Reset Password</h1>

<form action="" method="post">

{{ form.hidden\_tag() }}

<p>

{{ form.email.label }}<br>

{{ form.email(size=64) }}<br>

{% for error in form.email.errors %}

<span style="color: red;">[{{ error }}]</span>

{% endfor %}

</p>

<p>{{ form.submit() }}</p>

</form>

{% endblock %}

Cuối cùng là tạo view function để handle form này:

*app/routes.py*: Reset password request view function.

from app.forms import ResetPasswordRequestForm

from app.email import send\_password\_reset\_email

@app.route('/reset\_password\_request', methods=['GET', 'POST'])

def reset\_password\_request():

if current\_user.is\_authenticated:

return redirect(url\_for('index'))

form = ResetPasswordRequestForm()

if form.validate\_on\_submit():

user = User.query.filter\_by(email=form.email.data).first()

if user:

send\_password\_reset\_email(user)

flash('Check your email for the instructions to reset your password')

return redirect(url\_for('login'))

return render\_template('reset\_password\_request.html',

title='Reset Password', form=form)

Function *send\_password\_reset\_email()* sẽ được tạo ở dưới.

Lưu ý ta luôn show message bằng flash, kể cả khi user không tồn tại, để chống việc hacker dùng form này để kiểm tra email có tồn tại trong DB không.

1. **Password Reset Tokens**

Ta cần implement 1 cách để generate password request link, ta cũng cần phải làm cho chỉ valid reset links mới có thể giúp reset password. Một cách để làm điều này là cho token cung cấp link cho app, và sẽ validate token trước khi cho phép reset password. Ta thường hay dùng *JSON Web Token*.

>>> import jwt

>>> token = jwt.encode({'a': 'b'}, 'my-secret', algorithm='HS256')

>>> token

'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhIjoiYiJ9.dvOo58OBDHiuSHD4uW88nfJikhYAXc\_sfUHq1mDi4G0'

>>> jwt.decode(token, 'my-secret', algorithms=['HS256'])

{'a': 'b'}

JWT có 3 argument: 1 payload dưới dạng dictionary, 1 secret key (*my-secret*), cuối cùng là algorithm để generate token (thường dùng *HS256*).

Payload sẽ sử dụng trong bài có dạng: *{‘reset\_password’: user\_id, ‘exp’: token\_expiration}. exp* chỉ ra time token sẽ expire.

Khi user click vào emailed link, token sẽ được gửi về application dưới dạng 1 phần của URL, app sẽ verify token, và dùng ID trong payload để identify user.

Vì token thuộc về user, nên ta sẽ generate và verify nó dưới dạng methods của *User*:

*app/models.py*: Reset password token methods.

from time import time

import jwt

from app import app

class User(UserMixin, db.Model):

# ...

def get\_reset\_password\_token(self, expires\_in=600):

return jwt.encode(

{'reset\_password': self.id, 'exp': time() + expires\_in},

app.config['SECRET\_KEY'], algorithm='HS256')

@staticmethod

def verify\_reset\_password\_token(token):

try:

id = jwt.decode(token, app.config['SECRET\_KEY'],

algorithms=['HS256'])['reset\_password']

except:

return

return User.query.get(id)

*verify\_reset\_password\_token()* là 1 static method, để có thể được gọi từ ngoài (không cần qua instance của class).

1. **Sending a Password Reset Email**

Khi đã có token rồi, ta có thể tiến hành generate password reset email, đầu tiên là thêm function *send\_password\_reset\_email()* ở file *email.py*

*app/email.py*: Send password reset email function.

from flask import render\_template

from app import app

# ...

def send\_password\_reset\_email(user):

token = user.get\_reset\_password\_token()

send\_email('[Microblog] Reset Your Password',

sender=app.config['ADMINS'][0],

recipients=[user.email],

text\_body=render\_template('email/reset\_password.txt',

user=user, token=token),

html\_body=render\_template('email/reset\_password.html',

user=user, token=token))

Ta có thể thấy *text\_body* và *html\_body*  của email sẽ dùng *render\_template()* để render 2 file mới như sau:

*app/templates/email/reset\_password.txt*: Text for password reset email.

Dear {{ user.username }},

To reset your password click on the following link:

{{ url\_for('reset\_password', token=token, \_external=True) }}

If you have not requested a password reset simply ignore this message.

Sincerely,

The Microblog Team

*app/templates/email/reset\_password.html*: HTML for password reset email.

<p>Dear {{ user.username }},</p>

<p>

To reset your password

<a href="{{ url\_for('reset\_password', token=token, \_external=True) }}">

click here

</a>.

</p>

<p>Alternatively, you can paste the following link in your browser's address bar:</p>

<p>{{ url\_for('reset\_password', token=token, \_external=True) }}</p>

<p>If you have not requested a password reset simply ignore this message.</p>

<p>Sincerely,</p>

<p>The Microblog Team</p>

Ở đây, *url\_for()* gọi thêm argument *\_external=True*: Thường thì URLs generated bởi *url\_for()* là relative URL, tức là ví dụ *url\_for(‘user’, username=’susan’)* sẽ return */user/susan.* Khi ta cần gửi URL qua email, ta không có phần đầu của URL trên, nên set *\_external=True* setx giúp generate ra complete URLs. Ví dụ trên sẽ tạo ra link [*http://localhost:5000/user/susan*](http://localhost:5000/user/susan).

1. **Resetting a User Password**

Khi user click vào email link, sẽ có 1 route nữa được thêm vào, và đây là view function của nó:

*app/routes.py*: Password reset view function.

from app.forms import ResetPasswordForm

@app.route('/reset\_password/<token>', methods=['GET', 'POST'])

def reset\_password(token):

if current\_user.is\_authenticated:

return redirect(url\_for('index'))

user = User.verify\_reset\_password\_token(token)

if not user:

return redirect(url\_for('index'))

form = ResetPasswordForm()

if form.validate\_on\_submit():

user.set\_password(form.password.data)

db.session.commit()

flash('Your password has been reset.')

return redirect(url\_for('login'))

return render\_template('reset\_password.html', form=form)

1 vài lưu ý:

* Trong view function này, trước hết ta đảm bảo user không logged in, sau đó ta identify user bằng token verificatin.
* Nếu token valid, ta đưa user đến 1 second form để nhập pasword mới

Đây là form để reset password:

*app/forms.py*: Password reset form.

class ResetPasswordForm(FlaskForm):

password = PasswordField('Password', validators=[DataRequired()])

password2 = PasswordField(

'Repeat Password', validators=[DataRequired(), EqualTo('password')])

submit = SubmitField('Request Password Reset')

và html tương ứng:

*app/templates/reset\_password.html*: Password reset form template.

{% extends "base.html" %}

{% block content %}

<h1>Reset Your Password</h1>

<form action="" method="post">

{{ form.hidden\_tag() }}

<p>

{{ form.password.label }}<br>

{{ form.password(size=32) }}<br>

{% for error in form.password.errors %}

<span style="color: red;">[{{ error }}]</span>

{% endfor %}

</p>

<p>

{{ form.password2.label }}<br>

{{ form.password2(size=32) }}<br>

{% for error in form.password2.errors %}

<span style="color: red;">[{{ error }}]</span>

{% endfor %}

</p>

<p>{{ form.submit() }}</p>

</form>

{% endblock %}

1. **Asynchronous Emails**

Để app chạy nhanh hơn, ta nên cho nó chạy ngầm: có 2 cách: *threading* và *multiprocessing*. Do start thread mới nhẹ hơn, nên ta sẽ dùng *threading*

*app/email.py*: Send emails asynchronously.

from threading import Thread

# ...

def send\_async\_email(app, msg):

with app.app\_context():

mail.send(msg)

def send\_email(subject, sender, recipients, text\_body, html\_body):

msg = Message(subject, sender=sender, recipients=recipients)

msg.body = text\_body

msg.html = html\_body

Thread(target=send\_async\_email, args=(app, msg)).start()

arguments cho thread bao gồm cả *msg* và *app*. Khi làm việc với threads có 1 design aspect của Flask cần được lưu ý: Flask sử dụng *contexts*  để tránh việc phải pass arguments across function. Có 2 kiểu context: application context và request context. Đọc thêm: [The Application Context — Flask Documentation (1.1.x) (palletsprojects.com)](https://flask.palletsprojects.com/en/1.1.x/appcontext/#:~:text=The%20application%20context%20is%20a,lifetime%20as%20an%20application%20context.)

Nhiều extension cần application context để llafm việc, để giúp chúng tìm ra Flask application instance mà không cần passed nó như 1 argument, vì chúng cần configuration được lưu trong *app.config* object để làm việc. Flask-Mail làm việc như vậy, chúng cần biết config values cho email server.

Application context được tạo với *with app.app\_context()* giúp application instance có thể được accessed qua *current\_app* variable.

# **Chapter 11: Facelift**

This chapter will focus on the front end, to make the web look good.

1. **Bootstrap**

1 vài lợi ích của bootstrap:

* giống nhau trong phần lớn các trình duyệt
* handle desktop/tablet/phone screen sizes
* customizable layouts
* có sẵn style cho navigation bars, forms, buttons, alerts, popups, etc

Cách trực tiếp nhất để dùng Bootstrap là import bootstrap.min.css file từ template. Ta có thể download 1 bản copy của file này và add vào project, hoặc import từ 1 CDN.

Flask có một extension là Flask-Bootstrap cung cấp các ready to use base template có sẵn Bootstrap framework installed.

(venv) $ pip install flask-bootstrap

Để dùng Flask-Bootstrap, ta cần initialize nó:

*app/\_\_init\_\_.py*: Flask-Bootstrap instance.

# ...

from flask\_bootstrap import Bootstrap

app = Flask(\_\_name\_\_)

# ...

bootstrap = Bootstrap(app)

Sau khi initialized, một *bootstrap/base.html* template sẽ available, và có thể được referenced bằng *extends*

Như các bài trước, chúng ta đang cho các file template *extends* từ 1 file *base.html*. Bây giờ ta sẽ sửa lại thành kiến trúc 3 level: *bootstrap/base.html* sẽ cung cấp basic structures cho toàn bộ page, bao gồm cả Bootstrap framework files. *base.html* sẽ thừa kế từ *bootstrap/base.html* và các file khác sẽ thừa kế từ *base.html*.

File *base.html* sẽ được sửa lại như sau:

*app/templates/base.html*: Redesigned base template.

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

{% block title %}

{% if title %}{{ title }} - Microblog{% else %}Welcome to Microblog{% endif %}

{% endblock %}

{% block navbar %}

<nav class="navbar navbar-default">

... navigation bar here (see complete code on GitHub) ...

</nav>

{% endblock %}

{% block content %}

<div class="container">

{% with messages = get\_flashed\_messages() %}

{% if messages %}

{% for message in messages %}

<div class="alert alert-info" role="alert">{{ message }}</div>

{% endfor %}

{% endif %}

{% endwith %}

{# application content needs to be provided in the app\_content block #}

{% block app\_content %}{% endblock %}

</div>

{% endblock %}

File này sẽ thừa kế từ *bootstrap/base.html*, sau đó implement 3 blocks cho 3 phần page title, navigation và content.

Lưu ý: do *content* là 1 block của Flask-Bootstrap, nên ta cần sửa lại tên block của chúng ta thành *app\_content*, và phải sửa ở các file tương ứng. Ví dụ file *404.html :*

*app/templates/404.html*: Redesigned 404 error template.

{% extends "base.html" %}

{% block app\_content %}

<h1>File Not Found</h1>

<p><a href="{{ url\_for('index') }}">Back</a></p>

{% endblock %}

1. **Rendering Bootstrap Forms**

Bootstrap hỗ trợ render form. Thay vì phải edit style form field từng cái 1, bootstrap có 1 macro nhận Flask-WTF Form object là 1 argument và renders complete form bằng Bootstrap styles.

VD với file *register.html*

*app/templates/register.html*: User registration template.

{% extends "base.html" %}

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

{% block app\_content %}

<h1>Register</h1>

<div class="row">

<div class="col-md-4">

{{ wtf.quick\_form(form) }}

</div>

</div>

{% endblock %}

Ta có thể thấy *wtf.quick\_form()* macro giúp render form trong 1 dòng, bao gồm cả support cho display errors và style theo Bootstrap framework.

1. **Rendering Pagination Links**

*app/templates/index.html*: Redesigned pagination links.

...

<nav aria-label="...">

<ul class="pager">

<li class="previous{% if not prev\_url %} disabled{% endif %}">

<a href="{{ prev\_url or '#' }}">

<span aria-hidden="true">&larr;</span> Newer posts

</a>

</li>

<li class="next{% if not next\_url %} disabled{% endif %}">

<a href="{{ next\_url or '#' }}">

Older posts <span aria-hidden="true">&rarr;</span>

</a>

</li>

</ul>

</nav>

Thay vì hiding next/previous link, ta dùng *disabled state*, giúp nó chuyển sang màu xám.

# **Chapter 12: Dates and Times**

Ở server nên dùng chung 1 mốc timezone, thường dùng UTC. Ở users side thì nên lấy timezone từ OS của user.

Có 1 cách dùng javascript và lấy local timezone của người dùng

**1. Introducing Moment.js and Flaks-Moment**

Moment.js là 1 open-source JS library nhận vào date time và render chúng.

Để dùng flask-moment, trước hết ta tạo instance cho chúgn trong *\_\_init\_\_.py:*

*app/\_\_init\_\_.py*: Flask-Moment instance.

# ...

from flask\_moment import Moment

app = Flask(\_\_name\_\_)

# ...

moment = Moment(app)

Khác với các extensions khác, Flask-Moment hoạt động cùng với moment.js, nên mọi template phải include thư viện này. Để đảm bảo việc đó, ta có thể include nó ở trong base template:

*app/templates/base.html*: Including moment.js in the base template.

...

{% block scripts %}

{{ super() }}

{{ moment.include\_moment() }}

{% endblock %}

**2. Using Moment.js**

1 VD là ta có thể giúp hiện last seen của user như sau:

*app/templates/user.html*: Render timestamp with moment.js.

{% if user.last\_seen %}

<p>Last seen on: {{ moment(user.last\_seen).format('LLL') }}</p>

{% endif %}

# **Chapter 13: Internationalization and Localization (I18n and L10n)**

This chapter will implement a translator workflow to offer the application in a choice of languages.

**1. Introduction to Flask-Babel**

Cài đặt và initialize cho Flask-Babel

*app/\_\_init\_\_.py*: Flask-Babel instance.

# ...

from flask\_babel import Babel

app = Flask(\_\_name\_\_)

# ...

babel = Babel(app)

Sau đó thêm các ngôn ngữ available vào file *config.py*

*config.py*: Supported languages list.

class Config(object):

# ...

LANGUAGES = ['en', 'es']

*Babel* instance có 1 decorator có tên *localeselector*, được invoked cho mỗi request để select language translation:

*app/\_\_init\_\_.py*: Select best language.

from flask import request

# ...

@babel.localeselector

def get\_locale():

return request.accept\_languages.best\_match(app.config['LANGUAGES'])

*accept\_language* là 1 object của Flask’s request, giúp làm việc với Accept-Language header của request. Header này chỉ ra client language và locale preferences như 1 weighted list.

**2. Marking Texts to Translate in Python Source Code**

Để làm cho 1 app có nhiều language, ta cần mark tất cả các đoạn text cần translate. Sau khi được mark, Flask-Babel sẽ scan toàn bộ files, và extract những đoạn text đó bằng *gettext* tool.

Text thường được mark bằng 1 dấu gạch dưới: *\_()­*. Đơn giản nhất là với các literal strings trong source code:

from flask\_babel import \_

# ...

flash(\_('Your post is now live!'))

Với text có dynamic component:

flash(\_('User %(username)s not found.', username=username))

Một vài string sẽ được assigned khi application starting up, ví dụ với các labels ở form field, nên ta sẽ không thể evaluate được nên dùng language nào. Ta sẽ xử lý chúng với *lazy\_gettext()*, giúp delay việc evaluation đến khi string được sử dụng, tức là khi chúng được vào 1 request.

from flask\_babel import lazy\_gettext as \_l

class LoginForm(FlaskForm):

username = StringField(\_l('Username'), validators=[DataRequired()])

# ...

**3. Marking texts to Translate in Templates**

Ta cũng có thể dùng \_() cho html file. VD từ

<h1>File Not Found</h1>

thành

<h1>{{ \_('File Not Found') }}</h1>

Lưu ý là ta phải thêm cả {{ }} để chúng được evaluated.

Với các câu cần translate có dynamic component:

<h1>{{ \_('Hi, %(username)s!', username=current\_user.username) }}</h1>

**4. Extracting Text To Translate**

Sau khi đã mark, ta có thể dùng command pybabel để extract chúng vào file *.pot* (portable object template).

Công đoạn extraction này cần 1 file config để chỉ ra file nào cần được scanned for translatable text. *babel.cfg*: PyBabel configuration file.

[python: app/\*\*.py]

[jinja2: app/templates/\*\*.html]

extensions=jinja2.ext.autoescape,jinja2.ext.with\_

Sau đó chạy command:

(venv) $ pybabel extract -F babel.cfg -k \_l -o messages.pot .

*The pybabel extract command reads the configuration file given in the -F option, then scans all the code and template files in the directories that match the configured sources, starting from the directory given in the command (the current directory or . in this case). By default, pybabel will look for \_() as a text marker, but I have also used the lazy version, which I imported as \_l(), so I need to tell the tool to look for those too with the -k \_l. The -o option provides the name of the output file.*

**5. Generating a Language Catalog**

Bước tiếp theo là sẽ cần tạo translation cho mỗi language. VD để tạo cho tiếng TBN, code là ‘es’, ta chạy command sau:

(venv) $ pybabel init -i messages.pot -d app/translations -l es

creating catalog app/translations/es/LC\_MESSAGES/messages.po based on messages.pot

*The pybabel init command takes the*messages.pot*file as input and writes a new language catalog to the directory given in the -d option for the language specified in the -l option. I'm going to be installing all the translations in the*app/translations*directory, because that is where Flask-Babel will expect translation files to be by default. The command will create a*es*subdirectory inside this directory for the Spanish data files. In particular, there will be a new file named*app/translations/es/LC\_MESSAGES/messages.po*, that is where the translations need to be made.*

File *messages.po* sẽ chứa các cặp *msgid* (của base language) và *msgstr* (để điền bản dịch).

Sau đó, ta cần compile file đó thành file *.mo*:

(venv) $ pybabel compile -d app/translations

compiling catalog app/translations/es/LC\_MESSAGES/messages.po to

app/translations/es/LC\_MESSAGES/messages.mo

Trong trường hợp quên đặt mark \_() và \_l(), ta có thể thêm mark đó vào, và update lại translation như sau:

(venv) $ pybabel extract -F babel.cfg -k \_l -o messages.pot .

(venv) $ pybabel update -i messages.pot -d app/translations

*The extract command is identical to the one I issued earlier, but now it will generate a new version of*messages.pot*with all the previous texts plus anything new that you recently wrapped with \_() or \_l(). The update call takes the new messages.pot file and merges it into all the*messages.po*files associated with the project. This is going to be an intelligent merge, in which any existing texts will be left alone, while only entries that were added or removed in*messages.pot*will be affected.*

**6. Translating Dates and Times**

Ta có thể translate các mốc timestamp bằng moment.js

Flask-Babel trả về selected language và locale cho 1 request thông qua *get\_locale()*, và ta sẽ thêm nó vào *g* object để access được từ base template:

*app/routes.py*: Store selected language in flask.g.

# ...

from flask import g

from flask\_babel import get\_locale

# ...

@app.before\_request

def before\_request():

# ...

g.locale = str(get\_locale())

**7. Command-Line Enhancements**

Nâng cao ([The Flask Mega-Tutorial Part XIII: I18n and L10n - miguelgrinberg.com](https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-xiii-i18n-and-l10n))

Ta tạo commands để thực hiện các lệnh của pybabel một cách user friendly hơn.

Commands sẽ bao gồm các lệnh sau:

* flask translate init LANG to add a new language
* flask translate update to update all language repositories
* flask translate compile to compile all language repositories

Flask dựa vào *Click* cho các Command-line operations của nó. Command *translate* ở trên sẽ là root cho 3 command *init, update, compile*, được tạo bằng decorator *app.cli.group()*.

*app/cli.py*: Translate command group.

from app import app

@app.cli.group()

def translate():

"""Translation and localization commands."""

pass

Do nó là parent command nên không cần làm gì cả => *pass.*

Phần docstring """Translation and localization commands.""" Sẽ hiện ở trong *–help* menu

Command *update* và *compile* không nhận tham số gì:

*app/cli.py*: Update and compile sub-commands.

import os

# ...

@translate.command()

def update():

"""Update all languages."""

if os.system('pybabel extract -F babel.cfg -k \_l -o messages.pot .'):

raise RuntimeError('extract command failed')

if os.system('pybabel update -i messages.pot -d app/translations'):

raise RuntimeError('update command failed')

os.remove('messages.pot')

@translate.command()

def compile():

"""Compile all languages."""

if os.system('pybabel compile -d app/translations'):

raise RuntimeError('compile command failed')

Có thể thấy chúng nhận decorator từ *translate.* Lệnh *os.system()* sẽ trả về 0 nếu không có lỗi => ta dùng *if* để raise exception nếu có.

Command *init* có nhận tham số là language code:

*app/cli.py*: Init sub-command.

import click

@translate.command()

@click.argument('lang')

def init(lang):

"""Initialize a new language."""

if os.system('pybabel extract -F babel.cfg -k \_l -o messages.pot .'):

raise RuntimeError('extract command failed')

if os.system(

'pybabel init -i messages.pot -d app/translations -l ' + lang):

raise RuntimeError('init command failed')

os.remove('messages.pot')

Bước cuối cùng là enable commands này trong *microblog.py*

*microblog.py*: Register command-line commands.

from app import cli

Từ bây giờ, ta có thể chạy lệnh *flask translate --help* trong command line:

(venv) $ flask translate --help

Usage: flask translate [OPTIONS] COMMAND [ARGS]...

Translation and localization commands.

Options:

--help Show this message and exit.

Commands:

compile Compile all languages.

init Initialize a new language.

update Update all languages.

Và sử dụng chúng như sau:

(venv) $ flask translate init <language-code>

(venv) $ flask translate update

(venv) $ flask translate compile

# **Chapter 14: Ajax**

Chapter này sẽ thêm tính năng live translation cho các post của user.

Ở traditional server-side model mà ta xây dựng ở các chapter trước, thì phía client chỉ thực hiện HTTP requests đến server. Server sẽ phụ trách hầu hết toàn bộ công việc, và gửi 1 new web page cho client directly hoặc qua redirect.

Có những model khác mà client sẽ có vai trò chủ động hơn. Nó issues request đến server, server trả lời với 1 web page, nhưng page data không chỉ đơn thuần là HTML, mà có cả các đoạn code, thường viết bằng JS. Khi client nhận được page, nó hiển thị phần HTML, và execute phần code. Từ đó ta có thể có 1 client có thể tự làm việc mà không cần hoặc cần rất ít tương tác với server.

Thường các application bây giờ là hybrid giữa 2 models trên. Để implement tính năng real time translation, client browser sẽ gửi asynchronous requests đến server, server sẽ respond mà không cần phải refresh page. Technique này tên là Ajax (Asynchronous JavaScript and XML)

Workflow:

- Identify language of the text to translate

- Identify the preferred language for each user

- When user clicks on translation link, need to send Ajax request to the server, and server contact with 3rd party translation API.

- After the server sends back the response, client-side javascript code will dynamically insert the text into the page.

**1. Language identification:**

Để identify language, ta dùng 1 library tên là *guess\_language*

(venv) $ pip install guess\_language-spirit

Sau đó thêm nó vào *Post* model:

*app/models.py*: Add detected language to Post model.

class Post(db.Model):

# ...

language = db.Column(db.String(5))

Và migrate + update database:

(venv) $ flask db migrate -m "add language to posts"

(venv) $ flask db upgrade

Giờ ta có thể dùng để detect language:

*app/routes.py*: Save language for new posts.

from guess\_language import guess\_language

@app.route('/', methods=['GET', 'POST'])

@app.route('/index', methods=['GET', 'POST'])

@login\_required

def index():

form = PostForm()

if form.validate\_on\_submit():

language = guess\_language(form.post.data)

if language == 'UNKNOWN' or len(language) > 5:

language = ''

post = Post(body=form.post.data, author=current\_user,

language=language)

# ...

Ta cũng sẽ thêm 1 translate link vào các post:

*app/templates/\_post.html*: Add a translate link to posts.

{% if post.language and post.language != g.locale %}

<br><br><a href="#">{{ \_('Translate') }}</a>

{% endif %}

**2. Using a 3rd party Translation Service**

Có 2 translation service chính là của google và microsoft. Hướng dẫn dùng với microsoft như sau: [The Flask Mega-Tutorial Part XIV: Ajax - miguelgrinberg.com](https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-xiv-ajax).

Sau khi đã có translator API, ta thêm key vào file config:

*config.py*: Add Microsoft Translator API key to the configuration.

class Config(object):

# ...

MS\_TRANSLATOR\_KEY = os.environ.get('MS\_TRANSLATOR\_KEY')

Cài đặt HTTP requests để giúp gửi request đến API

(venv) $ pip install requests

Sau đó viết hàm để translate bằng API:

*app/translate.py*: Text translation function.

import json

import requests

from flask\_babel import \_

from app import app

def translate(text, source\_language, dest\_language):

if 'MS\_TRANSLATOR\_KEY' not in app.config or \

not app.config['MS\_TRANSLATOR\_KEY']:

return \_('Error: the translation service is not configured.')

auth = {

'Ocp-Apim-Subscription-Key': app.config['MS\_TRANSLATOR\_KEY'],

'Ocp-Apim-Subscription-Region': 'westus2'}

r = requests.post(

'https://api.cognitive.microsofttranslator.com'

'/translate?api-version=3.0&from={}&to={}'.format(

source\_language, dest\_language), headers=auth, json=[{'Text': text}])

if r.status\_code != 200:

return \_('Error: the translation service failed.')

return r.json()[0]['translations'][0]['text']

1 Vài lưu ý:

- đầu tiên ta check xem key có tồn tại không. Nếu không trả về 1 error msg dưới dạng string, để có thể hiện thị trên browser.

- để authenticate, ta cần truyền vào key lấy từ API. Nó sẽ nằm trong custom HTTP Header với tên là *Ocp-Apim-Subscription-Key*, cùng với region.

- *post()* method giúp gửi HTTP Post request đến URL trong 1st argument, lấy từ Keys and Endpoints page của API. Text được translate sẽ được gửi đi dưới dạng JSON format: {"Text": "the text to translate here"}.

- *request.post()* sẽ trả về object. Nếu status code là 200 (success) => dùng json() method để lấy thông tin từ JSON object.

**3. Ajax from the Server**

Khi user click vào translate link, 1 asynchronous HTTP request sẽ được gửi đến server. Asynchronous (ajax) request tương tự với routes và view function thường, nhưng nó sẽ return data, format theo kiểu XML/JSON.

*app/routes.py*: Text translation view function.

from flask import jsonify

from app.translate import translate

@app.route('/translate', methods=['POST'])

@login\_required

def translate\_text():

return jsonify({'text': translate(request.form['text'],

request.form['source\_language'],

request.form['dest\_language'])})

**4. Ajax from the Client**

Khi làm việc với JS trong browser, ta sẽ làm việc với DOM model. JS sẽ lấy tên node trong DOM model để thay đổi nội dung. Vậy nên ta cần thêm id cho các thẻ cần sửa, ví dụ:

*app/templates/\_post.html*: Add an ID to each blog post.

<span id="post{{ post.id }}">{{ post.body }}</span>

Việc này sẽ giúp thêm cái ID như *post1, post2* cho các bài post, và ta có thể dùng jQuery để locate nó với cú pháp sau:

$('#post1').text()

$ là 1 function trong jQuery library, # là selector syntax của jQuery.

Ta cũng cần có chỗ để insert translated text, với mỗi translation ID tương ứng với post ID ở trên:

*app/templates/\_post.html*: Add an ID to the translate link.

<span id="translation{{ post.id }}">

<a href="#">{{ \_('Translate') }}</a>

</span>

Bước tiếp theo là viết function thực hiện việc translate, giúp lấy input và output từ DOM nodes, source+dest languages và gửi asynchronous request đến server.

*app/templates/base.html*: Client-side translate function.

{% block scripts %}

...

<script>

function translate(sourceElem, destElem, sourceLang, destLang) {

$(destElem).html('<img src="{{ url\_for('static', filename='loading.gif') }}">');

$.post('/translate', {

text: $(sourceElem).text(),

source\_language: sourceLang,

dest\_language: destLang

}).done(function(response) {

$(destElem).text(response['text'])

}).fail(function() {

$(destElem).text("{{ \_('Error: Could not contact server.') }}");

});

}

</script>

{% endblock %}

1 Vài lưu ý:

- 2 arguments đầu là IDs của post và translate node.

- dòng đầu của function giúp thay 1 biểu tượng loading dạng gif vào translate link để user biết được quá trình dịch đagn trong progress.

- tiếp theo ta dùng *$.post()* để gửi POST request đến /translate. Function này nhận 2 argument: đầu tiên là URL để gửi request đến, tiếp theo là dictionary chứa data.

- JS thường có nhiều callback functions *(promises)*. Ta sẽ thêm các promises này vào để chỉ ra việc cần làm khi request đã thực hiện xong, vì trong JS mọi thứ là asynchronous. Vậy nên ta thêm vào 2 callback function *.done().fail()*.

Cuối cùng, ta trigger *translate()* function khi user click vào translate link bằng thẻ *href*:

*app/templates/\_post.html*: Translate link handler.

<span id="translation{{ post.id }}">

<a href="javascript:translate(

'#post{{ post.id }}',

'#translation{{ post.id }}',

'{{ post.language }}',

'{{ g.locale }}');">{{ \_('Translate') }}</a>

</span>

Thẻ *href* có thể nhận JS code với prefixed *javascript:*. Vì link này sẽ được rendered ở server khi client request page, nên ta có thể dùng {{ }} để generate arguments cho function translate. # ở trước post và translation chỉ ra rằng theo sau dấu # đó là một element ID.

Cuối cùng ta cần update và compile lại language:

(venv) $ flask translate update

(venv) $ flask translate compile

# **Chapter 15: A Better Application Structure**

**1. Blueprints**

Blueprint trong Flask là 1 logical structure represents a subset of the application. Một blueprint có thể chứa routes, view functions, forms, templates, static files.

Contents của blueprint sẽ ở trạng thái dormant (inactive). Để associate chúng, blueprint cần được registered với application.

**a) Error handling blueprint**

Cấu trúc của error handling blueprint như sau:

app/

errors/ <-- blueprint package

\_\_init\_\_.py <-- blueprint creation

handlers.py <-- error handlers

templates/

errors/ <-- error templates

404.html

500.html

\_\_init\_\_.py <-- blueprint registration

Hoặc ta có thể chuyển toàn bộ file templates của phần error vào trong thư mục errors/templates bằng cách thêm arguments *template\_folder=’templates’* vào Blueprint() constructor.

Sau đó ở file *\_\_init\_\_.py* ta sửa như sau:

*app/errors/\_\_init\_\_.py*: Errors blueprint.

from flask import Blueprint

bp = Blueprint('errors', \_\_name\_\_)

from app.errors import handlers

*Blueprint* constructor nhận vào tên của blueprint và tên của base module (thường được set là *\_\_name\_\_*). Sau đó ta import *handlers.py* module, với nội dung như sau:

from flask import render\_template

from app import db

from app.errors import bp

@bp.app\_errorhandler(404)

def not\_found\_error(error):

    return render\_template('errors/404.html'), 404

@bp.app\_errorhandler(500)

def internal\_error(error):

    db.session.rollback()

    return render\_template('errors/500.html'), 500

Tức là thay vì dùng @app.errorhandler decorator, ta sẽ dùng blueprint decorator *@bp.app\_errorhandler*.

Bước cuối cùng là register blueprint với application.

*app/\_\_init\_\_.py*: Register the errors blueprint with the application.

app = Flask(\_\_name\_\_)

# ...

from app.errors import bp as errors\_bp

app.register\_blueprint(errors\_bp)

# ...

from app import routes, models # <-- remove errors from this import!

**b) Authentication Blueprint**

Cấu trúc như sau:

app/

auth/ <-- blueprint package

\_\_init\_\_.py <-- blueprint creation

email.py <-- authentication emails

forms.py <-- authentication forms

routes.py <-- authentication routes

templates/

auth/ <-- blueprint templates

login.html

register.html

reset\_password\_request.html

reset\_password.html

\_\_init\_\_.py <-- blueprint registration

Khi define 1 route trong blueprint, ta dùng *@bp.route* thay vì *@app.route*. Và với function url\_for(), argument đầu mà nó nhận sẽ phải include tên của blueprint và tên của view function. Tức là VD từ *url\_for(‘login’)* thành *url\_for(‘auth.login’).*

Để register auth blueprint với application, ta dùng 1 format khác:

*app/\_\_init\_\_.py*: Register the authentication blueprint with the application.

# ...

from app.auth import bp as auth\_bp

app.register\_blueprint(auth\_bp, url\_prefix='/auth')

# ...

*url\_prefix* làoptional, khi đó mọi routes defined trong blueprint sẽ có prefix này trong url của nó. Bây giờ URL đén login sẽ là [*http://localhost:5000/auth/login*](http://localhost:5000/auth/login)*.*

**c) Main Application Blueprint**

Tương tự 2 phần kia, trong thư mục *main*

**2. The application Factory Pattern**

Có global variable là *application* sẽ làm phức tạp nhiều thứ, đặc biệt ở công đoạn testing. Nên ta sẽ tạo 1 function *create\_app()* giúp tạo ra Flask app instance, và loại bỏ global variable.

*app/\_\_init\_\_.py*: Application factory function.

# ...

db = SQLAlchemy()

migrate = Migrate()

login = LoginManager()

login.login\_view = 'auth.login'

login.login\_message = \_l('Please log in to access this page.')

mail = Mail()

bootstrap = Bootstrap()

moment = Moment()

babel = Babel()

def create\_app(config\_class=Config):

app = Flask(\_\_name\_\_)

app.config.from\_object(config\_class)

db.init\_app(app)

migrate.init\_app(app, db)

login.init\_app(app)

mail.init\_app(app)

bootstrap.init\_app(app)

moment.init\_app(app)

babel.init\_app(app)

# ... no changes to blueprint registration

if not app.debug and not app.testing:

# ... no changes to logging setup

return app

# **The end**