Flask Python

▼ Core concepts

▼ HTML escape: Khi trả về HTML (loại phản hồi mặc định trong Flask), mọi giá trị do người dùng cung cấp được hiển thị ở đầu ra phải được thoát (escape) để bảo vệ khỏi các cuộc tấn công injection. Các mẫu HTML được hiển thị bằng Jinja, sẽ tự động thực hiện việc này.

```
from markupsafe import escape

@app.route("/<name>")
def hello(name):
    return f"Hello, {escape(name)}!"
```

▼ Routing: Các ứng dụng web hiện đại sử dụng các URL có ý nghĩa để trợ giúp người dùng. Người dùng có nhiều khả năng thích một trang và quay lại nếu trang đó sử dụng một URL có ý nghĩa mà họ có thể ghi nhớ và sử dụng để truy cập trực tiếp vào một trang.

```
@app.route('/')
def index():
    return 'Index Page'

@app.route('/hello')
def hello():
    return 'Hello, World'
```

Variable Rule: Bạn có thể thêm các phần có thể thay đổi vào URL bằng cách đánh dấu các phần bằng <variable_name>. Sau đó, hàm của bạn sẽ nhận được <variable_name> làm đối số từ khóa. Theo tùy chọn, bạn có thể sử dụng trình chuyển đổi để chỉ định loại đối số.

```
from markupsafe import escape
```

```
@app.route('/user/<username>')
def show_user_profile(username):
    # show the user profile for that user
    return f'User {escape(username)}'

@app.route('/post/<int:post_id>')
def show_post(post_id):
    # show the post with the given id, the id is an intege
    return f'Post {post_id}'

@app.route('/path/<path:subpath>')
def show_subpath(subpath):
    # show the subpath after /path/
    return f'Subpath {escape(subpath)}
```

HTTP Method: Các ứng dụng web sử dụng các phương thức HTTP khác nhau khi truy cập URL. Bạn nên tự làm quen với các phương thức HTTP khi làm việc với Flask. Theo mặc định, tuyến đường chỉ trả lời các yêu cầu GET. Bạn có thể sử dụng đối số phương thức của trình trang trí Route() để xử lý các phương thức HTTP khác nhau.

```
from flask import request

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        return do_the_login()
    else:
        return show_the_login_form()
```

▼ Static File: Các ứng dụng web động cũng cần các tệp tĩnh. Đó thường là nơi các tệp CSS và JavaScript đến. Lý tưởng nhất là máy chủ web của bạn được cấu hình để phục vụ chúng cho bạn, nhưng trong quá trình phát triển, Flask có thể làm điều đó. Chỉ cần tạo một thư mục có tên static trong gói của bạn hoặc bên cạnh mô-đun của bạn và nó sẽ có sẵn tại /static trên ứng dụng.

```
url_for('static', filename='style.css')

# The file has to be stored on the filesystem as static/st
```

▼ Rendering templates: Việc tạo HTML từ bên trong Python khá cồng kềnh vì bạn phải tự mình thực hiện việc escape HTML để giữ an toàn cho ứng dụng. Do đó, Flask tự động định cấu hình công cụ tạo mẫu Jinja2 cho bạn.

Templates có thể được sử dụng để tạo bất kỳ loại tệp văn bản nào. Đối với các ứng dụng web, bạn chủ yếu sẽ tạo các trang HTML nhưng bạn cũng có thể tạo đánh dấu, văn bản thuần túy cho email và bất kỳ thứ gì khác.

Để hiển thị mẫu, bạn có thể sử dụng phương thức render_template(). Tất cả những gì bạn phải làm là cung cấp tên của mẫu và các biến bạn muốn chuyển đến công cụ tạo mẫu làm đối số từ khóa.

```
from flask import render_template

@app.route('/hello/')
@app.route('/hello/<name>')
def hello(name=None):
    return render_template('hello.html', name=name)
```

Bên trong các mẫu, bạn cũng có quyền truy cập vào các đối tượng config, request, session và g cũng như các hàm url_for() và get_flashed_messages().

Các mẫu đặc biệt hữu ích nếu sử dụng tính kế thừa. Về cơ bản, kế thừa mẫu giúp có thể giữ lại một số thành phần nhất định trên mỗi trang (như đầu trang, điều hướng và chân trang).

▼ Accessing Request Data: Đối với các ứng dụng web, điều quan trọng là phải phản ứng với dữ liệu mà máy khách gửi đến máy chủ. Trong Flask, thông tin này được cung cấp bởi đối tượng yêu cầu chung Context Local

Giải pháp đơn giản nhất để kiểm tra đơn vị là sử dụng trình quản lý bối cảnh test_request_context(). Kết hợp với câu lệnh with, nó sẽ liên kết một yêu

cầu kiểm tra để bạn có thể tương tác với nó.

```
from flask import request

with app.test_request_context('/hello', method='POST'):
    # now you can do something with the request until the
    # end of the with block, such as basic assertions:
    assert request.path == '/hello'
    assert request.method == 'POST'
```

Request Object:

```
@app.route('/login', methods=['POST', 'GET'])
def login():
    error = None
    if request.method == 'POST':
        if valid_login(request.form['username'],
                       request.form['password']):
            return log_the_user_in(request.form['username'
        else:
            error = 'Invalid username/password'
    # the code below is executed if the request method
    # was GET or the credentials were invalid
    return render_template('login.html', error=error)
```

Upload file qua request:

```
@app.route('/upload', methods=['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        f = request.files['the_file']
        f.save('/var/www/uploads/uploaded_file.txt')
```

Sử dụng Cookie: Lưu ý rằng cookie được đặt trên các đối tượng respone. Vì thông thường chỉ trả về các chuỗi từ các hàm View nên Flask sẽ chuyển đổi

chúng thành đối tượng phản hồi cho bạn. Nếu muốn làm điều đó một cách rõ ràng, bạn có thể sử dụng hàm make_response() rồi sửa đổi nó.

```
from flask import make_response

@app.route('/')
def index():
    resp = make_response(render_template(...))
    resp.set_cookie('username', 'the username')
    return resp
```

▼ Redirects and Error: Để chuyển hướng người dùng đến điểm cuối khác, hãy sử dụng hàm redirect(); để hủy sớm một yêu cầu có mã lỗi, hãy sử dụng hàm abort()

```
from flask import abort, redirect, url_for

@app.route('/')
def index():
    return redirect(url_for('login'))

@app.route('/login')
def login():
    abort(401)
    this_is_never_executed()
```

Theo mặc định, một trang lỗi đen trắng được hiển thị cho mỗi mã lỗi. Nếu bạn muốn tùy chỉnh trang lỗi, bạn có thể sử dụng trình trang trí errorhandler().

```
from flask import render_template

@app.errorhandler(404)
def page_not_found(error):
    return render_template('page_not_found.html'), 404
```

▼ Sessions: Ngoài request object còn có một đối tượng thứ hai gọi là phiên cho phép bạn lưu trữ thông tin cụ thể cho người dùng từ yêu cầu này sang yêu cầu tiếp theo. Điều này được triển khai dựa trên cookie cho bạn và chữ ký cookie bằng mật mã. Điều này có nghĩa là người dùng có thể xem nội dung cookie của bạn nhưng không thể sửa đổi nó, trừ khi họ biết khóa bí mật được sử dụng để ký.

```
from flask import session

# Set the secret key to some random bytes. Keep this reall
app.secret_key = b'_5#y2L"F4Q8z\n\xec]/'

@app.route('/')
def index():
    if 'username' in session:
        return f'Logged in as {session["username"]}'
    return 'You are not logged in'

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        session['username'] = request.form['username']
        return redirect(url_for('index'))
    return '''
        <form method="post">
            <p><input type=text name=username>
            <p><input type=submit value=Login>
        </form>
    '''

@app.route('/logout')
def logout():
    # remove the username from the session if it's there
    session.pop('username', None)
    return redirect(url_for('index'))
```

▼ Logging: bạn có thể có một số mã phía máy khách gửi yêu cầu HTTP đến máy chủ nhưng không đúng định dạng. Điều này có thể do người dùng giả mạo dữ liệu hoặc mã máy khách bị lỗi. Trong hầu hết các trường hợp, bạn có thể trả lời 400 Yêu cầu Không hợp lệ nhưng đôi khi điều đó không hiệu quả và code phải tiếp tục hoạt động.

Đây là khi sử dụng loggers, bạn có thể sử dụng nó để ghi lại những bất thường xảy ra.

```
# Một số ví dụ
app.logger.debug('A value for debugging')
app.logger.warning('A warning occurred (%d apples)', 42)
app.logger.error('An error occurred')
```

▼ Hooking in WSGI Middleware: để thêm middleware WSGI vào ứng dụng, sử dụng wsgi_app. Ví dụ như áp dụng phần mềm ProxyFix vào sau Nginx

```
from werkzeug.middleware.proxy_fix import ProxyFix
app.wsgi_app = ProxyFix(app.wsgi_app)
```

▼ Docktest: là module cho phép import các kiểm tra vào code giúp kiểm tra đầu ra đã đúng mong đợi chưa

```
import doctest


def add(a, b):
    """
    Given two integers, return the sum.

    :param a: int
    :param b: int
    :return: int

    >>> add(2, 3)
    5
```

```
    """
    return a + b

doctest.testmod()

# Output
# TestResults(failed=0, attempted=1)
```

▼ Tuple: giống với list nhưng các phần tử có thể không cùng định dạng, các phần tử có thứ tự và giá trị cố định không thể thay đổi

```
coral = ('blue coral', 'staghorn coral', 'pillar coral')
print(coral)
```

▼ Set & List & Dictionary: 3 kiểu dữ liệu tập hợp còn lại

List cho phép lưu trữ các phần tử theo thứ tự, có thể thay đổi, có thể chứa dữ liệu trùng, có thể lưu các kiểu dữ liệu khác nhau

```
list1 = [1, 2, 3, 4, [5, 6, 7], "A", 'B']
print (list1) # --> [1, 2, 3, 4, [5, 6, 7], 'A', 'B']
```

Set lưu các phần tử không có thứ tự, không có chỉ mục, không chứa dữ liệu trùng

```
set1 = {4, 3, 5, 2, 1, 6, 2}
print (set1) # --> {1, 2, 3, 4, 5, 6}
```

Dictionary (hay còn gọi là HashTable) là tập hợp các cặp key-value không có thứ tự

```
dictCar = {
    "brand": "Honda",
    "model": "Honda Civic",
    "year": 1972
```

```
}
print(dictCar) # --> {'brand': 'Honda', 'model': 'Honda Ci
```

▼ Decorator: là một design pattern cho phép mở rộng chức năng của 1 đối tượng mà không cần can thiệp điều chỉnh code của nó (ví dụ: @classmethod,...)

```
def display_decorator(func):
    def wrapper(str):
        # logic trước khi chạy hàm func
        print(f'Log: The function {func.__name__} is execu
        func(str)
        # logic sau khi chạy hàm func
        print('Log: Execution completed.\n')
    return wrapper

# không dùng decorator
def display(str):
    print(str)

display = display_decorator(display)
display('Hello world')

# dùng decorator
@display_decorator
def say_hello(str):
    print(str)

say_hello('Hello, Donald')

# kq
Log: The function say_hello is executing ...
Hello, Donald
Log: Execution completed.
```

▼ Lambda: là hàm không có tên và phần thân chỉ chứa một lệnh duy nhất

```
    x = lambda a : a + 10 # gọi lệnh x(5)
    y = x(5) # y = 5 + 10 = 15
```

Thường kết hợp lambda với decorator để mở rộng hàm

```
def makebold(f):
    return lambda: "<b>" + f() + "</b>"

def makeitalic(f):
    return lambda: "<i>" + f() + "</i>"

@makebold
@makeitalic
def say():
    return "Hello"

print(say()) #// kết quả là <b><i>Hello</i></b>
```

▼ SQLALchemy

SQLALchemy là công cụ toàn diện để làm việc với cơ sở dữ liệu và Python. Gồm 2 phần chính:

- SQLALchemy ORM (object relational mapper)
- SQLALchemy Core

▼ Thiết lập kết nối: bắt đầu các ứng dụng SQLALchemy đều phải tạo kết nối đến database bằng URL

```
# Code hiện tại đang được sử dụng
from flask_sqlalchemy import SQLAlchemy
from flask import Flask

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///test.db
```

```
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)

# Demo một câu lệnh
db.session.add(object)
```

▼ Xây dựng database: sau khi đã có kết nối, có thể tạo bảng trong database

```
class Todo(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    content = db.Column(db.String(200), nullable=False)
    date_created = db.Column(db.DateTime, default=datetime
    def __repr__(self):
        return '<Task %r>' % self.id
```

Ở đây, ta tạo một lớp (ánh xạ vào database là 1 bảng) có tên User với các thuộc tính (các cột trong DB) là id, content và date_create

▼ Thêm/ xóa dữ liệu trong database

```
# Add thêm 1 hàng trong db và xác nhận
db.session.add(new_task)
db.session.commit()

# Truy vấn các dữ liệu trong DB
tasks = Todo.query.all()
```

▼ REST API

REST API (hay RESTful API) là một tiêu chuẩn dùng trong việc thiết kế API cho các ứng dụng web (thiết kế Web services) để tiện cho việc quản lý các resource. Nó chú trọng vào tài nguyên hệ thống (tệp văn bản, ảnh, âm thanh, video...), bao gồm các trạng thái tài nguyên được định dạng và được truyền tải qua HTTP.

Chức năng quan trọng nhất của REST là quy định cách sử dụng các HTTP method (như GET, POST, PUT, DELETE...) và cách định dạng các URL cho ứng dụng web để quản các resource.

REST hoạt động chủ yếu dựa vào giao thức HTTP. Các hoạt động cơ bản sẽ sử dụng những phương thức HTTP riêng:

- GET (SELECT): Trả về một Resource hoặc một danh sách Resource.
- POST (CREATE): Tạo mới một Resource.
- PUT (UPDATE): Cập nhật thông tin cho Resource.
- DELETE (DELETE): Xoá một Resource.

Tạo REST API trong Flask:

```
# Ví dụ một API để thay đổi dữ liệu
@app.route('/update/<int:id>', methods=['GET', 'POST'])
def update(id):
    task = Todo.query.get_or_404(id)

    if request.method == 'POST':
        task.content = request.form['content']

        try:
            db.session.commit()
            return redirect('/')
        except:
            return 'There was an issue updating your task'

    else:
        return render_template('update.html', task=task)
```

▼ Alembic & Flask migrate

▼ Flask-migrate sẽ ánh xạ các mô hình được viết trong code sang cơ sở dữ liệu. Dùng để quản lý việc migrate db

```
# ví dụ sử dụng flask migrate
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate
```

```
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///app.db'

db = SQLAlchemy(app)
migrate = Migrate(app, db)

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(128))
```

Sau đó, có thể tạo repository migrate để lưu trữ db và migrate hay update khi cần

```
# tao repository migrate
flask db init

# migrate
flask db migrate -m "Initial migration."

# Ap dung thay doi vao db
flask db upgrade
```

Một số cấu hình Alembic khi sử dụng Flask Migrate:

- compare_type = True : cấu hình tự động migrate phát hiện thay đổi loại dữ liệu của cột
- render_as_batch = True : tóm tắt là bật option này sẽ không khác gì làm việc trực tiếp với db

```
# ví dụ
migrate = Migrate(app, db, render_as_batch=False)
```

Flask_migrate đưa ra 1 lớp là Migrate, lớp này chứa tất cả chức năng giúp làm việc với db của extension

```
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate

db = SQLAlchemy()
migrate = Migrate()

def create_app():
     """Application-factory pattern"""
     ...
     ...
     db.init_app(app)
     migrate.init_app(app, db)
     ...
     ...
     return app
```

▼ Alembic có thể xem như Git cho database, giúp theo dõi những thay đổi về dữ liệu trong db. Alembic dùng để quản lý version migration, tự động tạo migration script.

Cấu trúc của file Alembic khi cài đặt:

```
yourproject/
    alembic/
        env.py
        README
        script.py.mako
        versions/
            3512b954651e_add_account.py
            2b1ae634e5cd_add_order_id.py
            3adcc9a56557_rename_username_field.py
```

Cấu hình môi trường (là chỉnh sửa môi trường ảo mình vừa tạo bằng virtualenv sử dụng alembic pk ???)

```
cd /path/to/yourproject
source /path/to/yourproject/.venv/bin/activate   # kich ho
alembic init alembic
```

Để chỉnh sửa môi trường hay key, ta thay đổi file alembic.ini

Các bước tạo migrate script

```
# tao script
$ alembic revision -m "Add a column"
Generating /path/to/yourapp/alembic/versions/ae1027a6acf_a
done

# vao file vua duoc tao ra trong alembic/version va them p
revision = 'ae1027a6acf'
down_revision = '1975ea83b712'

from alembic import op
import sqlalchemy as sa

def upgrade():
    op.add_column('account', sa.Column('last_transaction_d

def downgrade():
    op.drop_column('account', 'last_transaction_date')

# chay script
$ alembic upgrade head
INFO  [alembic.context] Context class PostgresqlContext.
INFO  [alembic.context] Will assume transactional DDL.
INFO  [alembic.context] Running upgrade 1975ea83b712 -> ae
```

Những lần sau nếu muốn thêm cột trong db ta chỉ cần chạy script

```
$ alembic upgrade ae1
```

Thay đổi version nếu cần thiết

```
$ alembic upgrade +2

$ alembic downgrade -1

$ alembic upgrade ae10+2
```

Lấy thông tin phiên bản hiện tại

```
$ alembic current
INFO  [alembic.context] Context class PostgresqlContext.
INFO  [alembic.context] Will assume transactional DDL.
Current revision for postgresql://scott:XXXXX@localhost/te
```

▼ API Validation

API validation là quá trình kiểm tra và xác nhận tính đúng đắn của các yêu cầu và phản hồi trong ứng dụng lập trình giao diện.

API validation bao gồm kiểm tra xem các yêu cầu gửi đến API có đúng định dạng và dữ liệu không, xác nhận rằng các tham số và giá trị được gửi đến API đều hợp lệ và tuân theo các quy định, cũng như đảm bảo rằng các phản hồi từ API trả về các kết quả chính xác và theo mong muốn.

▼ Flask-RESTful (extension)

> Flask-RESTful — Flask-RESTful 0.3.10 documentation
> Flask-RESTful is an extension for Flask that adds support for quickly building REST APIs. It is a lightweight abstraction that works with your existing ORM/libraries. Flask-RESTful encourages best practices with minimal
> https://flask-restful.readthedocs.io/en/latest/

Đây là một extension phổ biến cho Flask, cung cấp các tính năng mạnh mẽ để phát triển các API RESTful. Flask-RESTful hỗ trợ validation payload thông qua các trình phân tích cú pháp như `reqparse`. Bạn có thể định nghĩa các quy tắc validation cho các trường dữ liệu được truyền trong payload.

▼ Resource Routing

Khối xây dựng chính do Flask-RESTful cung cấp là các tài nguyên. Tài nguyên được xây dựng dựa trên các view plugin của Flask, giúp bạn dễ dàng truy cập vào nhiều phương thức HTTP chỉ bằng cách xác định các phương thức trên tài nguyên của mình.

```
from flask import Flask, request
from flask_restful import Resource, Api

app = Flask(__name__)
api = Api(app)

todos = {}

class TodoSimple(Resource):
    def get(self, todo_id):
        return {todo_id: todos[todo_id]}

    def put(self, todo_id):
        todos[todo_id] = request.form['data']
        return {todo_id: todos[todo_id]}

api.add_resource(TodoSimple, '/<string:todo_id>')

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

▼ Endpoints

Nhiều khi trong một API, tài nguyên của bạn sẽ có nhiều URL. Bạn có thể chuyển nhiều URL tới phương thức add_resource() trên đối tượng Api. Mỗi cái sẽ được chuyển đến Tài nguyên của bạn

```
api.add_resource(Todo,'/todo/<int:todo_id>', endpoint='
```

▼ Argument Parsing

Mặc dù Flask cung cấp quyền truy cập dễ dàng vào dữ liệu yêu cầu (tức là dữ liệu được mã hóa chuỗi truy vấn hoặc biểu mẫu POST), nhưng việc xác thực dữ liệu biểu mẫu vẫn là một điều khó khăn. Flask-RESTful có hỗ trợ tích hợp để xác thực dữ liệu yêu cầu bằng thư viện tương tự như argparse.

Không giống như module argparse, reqparse.RequestParser.parse_args() trả về một từ điển Python thay vì cấu trúc dữ liệu tùy chỉnh.

```
from flask_restful import reqparse

parser = reqparse.RequestParser()
parser.add_argument('rate', type=int, help='Rate to cha
args = parser.parse_args()
```

▼ Data Formating

Theo mặc định, tất cả các trường trong vòng lặp trả về của bạn sẽ được hiển thị nguyên trạng. Mặc dù điều này hoạt động tốt khi bạn chỉ xử lý các cấu trúc dữ liệu Python, nhưng nó có thể trở nên rất khó chịu khi làm việc với các đối tượng. Để giải quyết vấn đề này, Flask-RESTful cung cấp field module và decorator marshal_with(). Tương tự như Django ORM và WTForm, bạn sử dụng mô-đun trường để mô tả cấu trúc phản hồi của mình.

```
from flask_restful import fields, marshal_with

resource_fields = {
    'task':   fields.String,
    'uri':    fields.Url('todo_ep')
}

class TodoDao(object):
    def __init__(self, todo_id, task):
```

```
        self.todo_id = todo_id
        self.task = task

        # This field will not be sent in the response
        self.status = 'active'

class Todo(Resource):
    @marshal_with(resource_fields)
    def get(self, **kwargs):
        return TodoDao(todo_id='my_todo', task='Remembe
```

▼ Marshmallow (lib)

> marshmallow: simplified object serialization — marshmallow 3.21.1 documentation
> marshmallow is an ORM/ODM/framework-agnostic library for converting complex datatypes, such as objects, to and from native Python datatypes.
> https://marshmallow.readthedocs.io/en/stable/

Marshmallow là một thư viện serialization và validation cho Python, phổ biến trong việc xử lý dữ liệu JSON. Bạn có thể sử dụng Marshmallow để định nghĩa các schema cho dữ liệu và thực hiện validation trước khi xử lý các yêu cầu. Flask-Marshmallow là một extension của Flask kết hợp Marshmallow với Flask để hỗ trợ tích hợp dễ dàng hơn.

▼ Schema

```
# Create schema from dictionaries
from marshmallow import Schema, fields

UserSchema = Schema.from_dict(
    {"name": fields.Str(), "email": fields.Email(), "cr
)
```

```
# Create default schema
from marshmallow import Schema, fields
```

```
class UserSchema(Schema):
    name = fields.Str()
    email = fields.Email()
    created_at = fields.DateTime()
```

▼ Serializing Objects (“Dumping”)

Chuẩn hóa các đối tượng bằng cách chuyển chúng tới phương thức dump của lược đồ của bạn, phương thức này trả về kết quả được định dạng.

```
from pprint import pprint

user = User(name="Monty", email="monty@python.org")
schema = UserSchema()
result = schema.dump(user)
pprint(result)
# {"name": "Monty",
#  "email": "monty@python.org",
#  "created_at": "2014-08-17T14:54:16.049594+00:00"}
```

▼ Filtering Output

Bạn có thể không cần xuất tất cả các trường đã khai báo mỗi khi sử dụng lược đồ. Bạn có thể chỉ định trường nào sẽ xuất bằng tham số duy nhất.

```
summary_schema = UserSchema(only=("name", "email"))
summary_schema.dump(user)
# {"name": "Monty", "email": "monty@python.org"}
```

▼ Deserializing Objects (“Loading”)

Ngược lại với phương thức dump là load, phương thức này xác thực và giải tuần tự hóa một từ điển đầu vào thành cấu trúc dữ liệu cấp ứng dụng.

Theo mặc định, load sẽ trả về một từ điển gồm các tên trường được ánh xạ tới các giá trị đã được giải tuần tự hóa (hoặc đưa ra một ValidationError với một từ điển các lỗi xác thực).

```
from pprint import pprint

user_data = {
    "created_at": "2014-08-11T05:26:03.869245",
    "email": "ken@yahoo.com",
    "name": "Ken",
}
schema = UserSchema()
result = schema.load(user_data)
pprint(result)
# {'name': 'Ken',
#  'email': 'ken@yahoo.com',
#  'created_at': datetime.datetime(2014, 8, 11, 5, 26,
```

▼ Validation

chema.load() (và bản sao giải mã JSON của nó, Schema.loads()) gây ra lỗi ValidationError khi dữ liệu không hợp lệ được truyền vào. Bạn có thể truy cập từ điển các lỗi xác thực từ thuộc tính ValidationError.messages. Dữ liệu đã được giải tuần tự hóa chính xác có thể truy cập được trong ValidationError.valid_data. Một số trường, chẳng hạn như trường Email và URL, có xác thực tích hợp.

```
from marshmallow import ValidationError

try:
    result = UserSchema().load({"name": "John", "email"
except ValidationError as err:
    print(err.messages)  # => {"email": ['"foo" is not
    print(err.valid_data)  # => {"name": "John"}
```

Khi xác thực một bộ sưu tập, từ điển lỗi sẽ được khóa theo chỉ mục của các mục không hợp lệ.

```
from pprint import pprint
from marshmallow import Schema, fields, ValidationError

class BandMemberSchema(Schema):
    name = fields.String(required=True)
    email = fields.Email()

user_data = [
    {"email": "mick@stones.com", "name": "Mick"},
    {"email": "invalid", "name": "Invalid"},  # invalid
    {"email": "keith@stones.com", "name": "Keith"},
    {"email": "charlie@stones.com"},  # missing "name"
]

try:
    BandMemberSchema(many=True).load(user_data)
except ValidationError as err:
    pprint(err.messages)
    # {1: {'email': ['Not a valid email address.']},
    #  3: {'name': ['Missing data for required field.']
```

Bạn có thể thực hiện xác thực bổ sung cho một trường bằng cách chuyển đổi số xác thực. Có một số trình xác nhận được tích hợp sẵn trong mô-đun marshmallow.validate.

```
class UserSchema(Schema):
    name = fields.Str(validate=validate.Length(min=1))
    permission = fields.Str(validate=validate.OneOf(["r
    age = fields.Int(validate=validate.Range(min=18, ma


in_data = {"name": "", "permission": "invalid", "age":
try:
```

```
    UserSchema().load(in_data)
except ValidationError as err:
    pprint(err.messages)
    # {'age': ['Must be greater than or equal to 18 and
    #  'name': ['Shorter than minimum length 1.'],
    #  'permission': ['Must be one of: read, write, adm
```

Bạn có thể triển khai trình xác thực của riêng mình. Trình xác nhận là một lệnh gọi có thể chấp nhận một đối số duy nhất, giá trị để xác thực. Nếu xác thực không thành công, lệnh gọi sẽ đưa ra ValidationError kèm theo thông báo lỗi hữu ích hoặc trả về Sai (đối với thông báo lỗi chung).

```
def validate_quantity(n):
    if n < 0:
        raise ValidationError("Quantity must be greater
    if n > 30:
        raise ValidationError("Quantity must not be gre


class ItemSchema(Schema):
    quantity = fields.Integer(validate=validate_quantit


in_data = {"quantity": 31}
try:
    result = ItemSchema().load(in_data)
except ValidationError as err:
    print(err.messages)  # => {'quantity': ['Quantity m
```

Lưu ý: Validate xảy ra khi deserialization (đưa dữ liệu vào) nhưng không xảy ra khi serialization (đưa dữ liệu ra). Để cải thiện hiệu suất tuần tự hóa, dữ liệu được chuyển tới Schema.dump() được coi là hợp lệ.

▼ Field Validators as Method

Đôi khi việc viết trình xác nhận dưới dạng phương thức sẽ thuận tiện hơn. Sử dụng trình xác thực decorator để đăng ký các phương thức xác

thực trường.

```
from marshmallow import fields, Schema, validates, Vali


class ItemSchema(Schema):
    quantity = fields.Integer()

    @validates("quantity")
    def validate_quantity(self, value):
        if value < 0:
            raise ValidationError("Quantity must be gre
        if value > 30:
            raise ValidationError("Quantity must not be
```

▼ **Required Fields**

Tạo một trường bắt buộc bằng cách chuyển required=True. Sẽ xảy ra lỗi nếu giá trị đầu vào của Schema.load() bị thiếu.

Để tùy chỉnh thông báo lỗi cho các trường bắt buộc, hãy chuyển một lệnh có khóa bắt buộc làm đối số error_messages cho trường.

```
from pprint import pprint

from marshmallow import Schema, fields, ValidationError

class UserSchema(Schema):
    name = fields.String(required=True)
    age = fields.Integer(required=True, error_messages=
    city = fields.String(
        required=True,
        error_messages={"required": {"message": "City r
    )
    email = fields.Email()
```

```
try:
    result = UserSchema().load({"email": "foo@bar.com"}
except ValidationError as err:
    pprint(err.messages)
    # {'age': ['Age is required.'],
    # 'city': {'code': 400, 'message': 'City required'}
    # 'name': ['Missing data for required field.']}
```

▼ **Partial Loading**

Khi sử dụng cùng một lược đồ ở nhiều nơi, bạn có thể bỏ qua xác thực bắt buộc bằng cách chuyển một phần dữ liệu.

```
class UserSchema(Schema):
    name = fields.String(required=True)
    age = fields.Integer(required=True)


result = UserSchema().load({"age": 42}, partial=("name"
# OR UserSchema(partial=('name',)).load({'age': 42})
print(result)  # => {'age': 42}
```

▼ **Validation Without Deserialization**

Nếu bạn chỉ cần xác thực dữ liệu đầu vào (không cần giải tuần tự hóa thành một đối tượng), bạn có thể sử dụng Schema.validate().

```
errors = UserSchema().validate({"name": "Ronnie", "emai
print(errors)  # {'email': ['Not a valid email address.
```

▼ **“Read-only” and “Write-only” Fields**

Trong ngữ cảnh của API web, các tham số dump_only và Load_only về mặt khái niệm tương đương với các trường “chỉ đọc” và “chỉ ghi”.

```
class UserSchema(Schema):
    name = fields.Str()
```

```
    # password is "write-only"
    password = fields.Str(load_only=True)
    # created_at is "read-only"
    created_at = fields.DateTime(dump_only=True)
```

▼ **Handling Unknown Fields**

Theo mặc định, load sẽ đưa ra lỗi xác thực (validation error) nếu nó gặp một khóa không có Trường khớp trong lược đồ.

Có thể thay đổi hành vi này với option unknown:

- RAISE (default): đưa ra lỗi Xác thực nếu có bất kỳ trường nào không xác định
- EXCLUDE: loại trừ các trường không xác định
- INCLUDE: chấp nhận và bao gồm các trường không xác định

```
# Chỉ định trong class Meta hoặc Schema
from marshmallow import Schema, INCLUDE

class UserSchema(Schema):
    class Meta:
        unknown = INCLUDE

# cách 2
schema = UserSchema(unknown=INCLUDE)
```

▼ WTForms (lib)

Flask-WTF — Flask-WTF Documentation (1.2.x)
Simple integration of Flask and WTForms, including CSRF, file upload, and reCAPTCHA.
https://flask-wtf.readthedocs.io/en/1.2.x/

WTForms là một thư viện form validation cho Flask (và cũng cho các framework khác). Mặc dù chủ yếu được thiết kế để xử lý web forms, nhưng

nó cũng có thể được sử dụng để validate JSON payload trong các ứng dụng Flask.

▼ Validating Form

Xác thực request trong khi xử lý view

```
@app.route('/submit', methods=['GET', 'POST'])
def submit():
    form = MyForm()
    if form.validate_on_submit():
        return redirect('/success')
    return render_template('submit.html', form=form)
```

Lưu ý rằng bạn không phải chuyển request.form tới Flask-WTF; nó sẽ tự động tải. Và valid_on_submit sẽ kiểm tra xem đó có phải là yêu cầu POST hay không và nó có hợp lệ hay không.

Nếu biểu mẫu của bạn bao gồm xác thực, bạn sẽ cần thêm vào template của mình để hiển thị bất kỳ thông báo lỗi nào.

```
{% if form.name.errors %}
    <ul class="errors">
    {% for error in form.name.errors %}
        <li>{{ error }}</li>
    {% endfor %}
    </ul>
{% endif %}
```

▼ SQLALchemy kết hợp WTForms (chưa tìm được docs)

▼ jsonschema (lib)

jsonschema là một thư viện Python cho validation JSON payload dựa trên JSON Schema. Bạn có thể sử dụng nó để định nghĩa các schema cho dữ liệu JSON và thực hiện validation trước khi xử lý yêu cầu.

▼ The validator protocol

jsonschema định nghĩa một giao thức mà tất cả các validator class tuân thủ.

```
from flask import Flask, request, jsonify
from jsonschema import validate

app = Flask(__name__)

# JSON Schema for todo item
todo_schema = {
    "type": "object",
    "properties": {
        "id": {"type": "integer"},
        "task": {"type": "string"}
    },
    "required": ["id", "task"]
}

# Route for adding a new todo item
@app.route('/todo', methods=['POST'])
def add_todo():
    # Get JSON data from the request
    json_data = request.get_json()

    # Validate JSON data against the schema
    try:
        validate(instance=json_data, schema=todo_schema
    except Exception as e:
        return jsonify({"error": str(e)}), 400

    # If validation passed, add the todo item
    # For demonstration purposes, just return the added
    return jsonify(json_data), 201
```

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

- ▼ Validating Formats
- ▼ Versioned Validators
- ▼ Validating with Additional Types
- ▼ Type checking

- ▼ pydantic (lib)

  Pydantic là một thư viện Python được sử dụng để kiểm tra và validate dữ liệu. Nó cung cấp các công cụ để xác định các class có các trường dữ liệu và các ràng buộc liên quan đến dữ liệu, sau đó Pydantic sẽ tự động kiểm tra và chuyển đổi dữ liệu theo các quy tắc đã xác định.

```
from datetime import datetime

from pydantic import BaseModel, PositiveInt, ValidationErr

class User(BaseModel):
    id: int
    name: str
    signup_ts: datetime | None
    tastes: dict[str, PositiveInt]

external_data = {
    'id': 123,
    'str': 'John Doe',
    'signup_ts': '2019-06-01 12:22',
    'tastes': {
        'wine': 9,
        'cheese': 7,
        'cabbage': '1'
    },
}
```

```
user = User(**external_data)
print(user.model_dump())
"""
{
    'id': 123,
    'name': 'John Doe',
    'signup_ts': datetime.datetime(2019, 6, 1, 12, 22),
    'tastes': {'wine': 9, 'cheese': 7, 'cabbage': 1},
}
"""
external_data = {'id': 'not an int', 'tastes': {}}

try:
    User(**external_data)
except ValidationError as e:
    print(e.errors())
    """
    [
        {
            'type': 'int_parsing',
            'loc': ('id',),
            'msg': 'Input should be a valid integer, unabl
            'input': 'not an int',
            'url': 'https://errors.pydantic.dev/2/v/int_pa
        },
        {
            'type': 'missing',
            'loc': ('signup_ts',),
            'msg': 'Field required',
            'input': {'id': 'not an int', 'tastes': {}},
            'url': 'https://errors.pydantic.dev/2/v/missin
        },
    ]
    """
```

```
#kq
The input data is wrong here – id is not a valid integer,
```

▼ Khó khăn

jsonschema: biết là giống schema trong marshmallow nhưng đọc doc dùng code hơi lạ nhưng không hiểu

SQLALchemy with WTForms chưa tìm được doc

▼ Cấu hình môi trường (là chỉnh sửa môi trường ảo mình vừa tạo bằng virtualenv sử dụng alembic pk ???) - ở đoạn alembic

Alembic có thể quản lý môi trường của project nhưng ko phải là quản lý virtual environment

File .env trong alembic bao gồm tập lệnh Python được chạy mỗi khi migrate được gọi. Nó chứa các hướng dẫn để cấu hình và tạo ra đối tượng engine của SQLALchemy

Tập lệnh này là một phần của môi trường được tạo ra để run migrate có thể tùy chỉnh theo nhu cầu. Tập lệnh có thể được sửa đổi để nhiều engine có thể được vận hành, các đối số tùy chỉnh có thể được truyền vào môi trường di chuyển, các thư viện và mô hình cụ thể của ứng dụng có thể được tải vào và làm sẵn có.

Alembic bao gồm một tập hợp các mẫu khởi tạo có các loại khác nhau để sử dụng cho các trường hợp sử dụng khác nhau.

▼ Bổ sung 1

- Bluesprint là 1 phần của flask, là cách tổ chức flask thành các phần độc lập → để scale project

▼ Bổ sung 2

1. Interface và abstract khác nhau

   Interface có thể abstract nhiều class con nhưng chỉ liệt kê các method chứ ko implement và khi thêm method thì class con phải định nghĩa method

   Abstract class abstract ít hơn nhưng các class con dùng được luôn

2. Static method và class method và instance method

   Static method

3. Tuple, set, list khác nhau
4. Decorator là gì, ORM ưu nhược điểm, cách sd
5. File py_cache là gì

▼ Bổ sung 3

Khi project phát triển lớn và sử dụng blueprints có thể bị gặp lỗi 308 vòng lặp vô hạn

Nguyên nhân có thể do app.route và blueprint.route khác nhau

Cách sửa: thêm `strict_slashes=False` vào sau mỗi route của blueprint

```
@blueprint.route('/', strict_slashes=False)
def view_users():
    users = User.query.all()
    return render_template("users/view_users.html", users=use
```