



Tutorial Flask Conf

Arquitetura de Projetos Flask

Olá seja bem vindo ao tutorial da #FlaskConf 2018 meu nome é **Bruno Rocha** no twitter, github e outras redes meu nickname é [@rochacbruno](https://twitter.com/rochacbruno) e meu site <http://brunorocha.org>

Quero te convidar para se inscrever no meu canal <http://YouTube.com/CursoDePython>

E entrar no grupo FlaskBrasil do telegram: <https://t.me/flaskbrasil>

Os códigos deste tutorial estão em <http://github.com/rochacbruno/talkshow>

Este trabalho está licenciado com uma Licença [Creative Commons - Atribuição-NãoComercial-Compartilha Igual 4.0 Internacional](https://creativecommons.org/licenses/by-nc-sa/4.0/).

Agradecimentos:

Scientific Electronic Library Online - SciELO, Associação Python Brasil - APyB - Python Software Foundation - PSF, Avidity, TopTal, HE:Labs, Equipe Python Sudeste, Equipe Flask Conf: Filipe Cifali, Jamil Atta, Juan Funez, Renne Rocha e Samuel Sampaio. Voluntários e palestrantes da FLaskConf, David Lord, Grupo FlaskBrasil, Todos os inscritos na FlaskConf e em especial a Karla Magueta Soler e Erik Rocha Soler.

Para começar vamos alinhar as expectativas!

O objetivo deste tutorial

- Apresentar sugestões para arquitetura de um Projeto Flask
- Mostrar erros comuns a serem evitados no Flask
- Praticar a utilização de Factories, Blueprints e Extensões

Desenvolveremos

- Aplicação Flask com factory pattern
- Interface CLI com o flask.cli
- Acesso a banco de dados NoSQL (MongoDB)
- Criação de Blueprints
- Web UI com formulários wtforms e templates Flask-bootstrap
- Interface Admin
- Autenticação
- Rest API
- Testes da REST API

O que não veremos: (importante para otimizar o tempo)

- Bancos de dados SQL ou ORM (SQLAlchemy, Peewee etc)
- Extensões complexas de autenticação
- Extensões complexas de REST API
- Deploy
- HTML/CSS/Frontend
- Uso avançado do Jinja

Notas importantes:

- Durante o tutorial responderei perguntas rápidas, mas se perceber que a pergunta foge do foco do tutorial não será prontamente respondida mas poderemos conversar a respeito no grupo <https://t.me/flaskbrasil> no Telegram.
- O objetivo deste tutorial é ensinar os conceitos e não desenvolver o aplicativo completo, portanto algumas coisas serão simplificadas e caso não dê tempo de terminar o desenvolvimento completo do aplicativo não tem problema! o importante é fixar os conceitos principais de arquitetura com Flask.

Ementa:

- Parte 1: Estruturando a base do projeto Flask
- Parte 2: A linha de comando e o `Flask` CLI
- Parte 3: Banco de dados NoSQL (MongoDB/TinyMongo)
- Parte 4: View e processamento de Formulário usando Blueprint
- Parte 5: A interface administrativa
- Parte 6: REST API
- Parte 7: Protegendo o app com login/senha
- Parte 8: Testando a API

Parte 1: Estruturando a base do projeto

Um aplicativo básico no flask pode ser feito de maneira bastante simples com apenas 5 linhas de código:

`app.py` é o nome de arquivo onde geralmente escrevemos o ponto de entrada da aplicação Flask.

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def index():
    return "Hello World"
```

O problema é que quando o projeto começa a crescer vamos nos deparar com uma série de problemas e precisamos pensar em como evitá-los.

Um desses problemas é conhecido como `**circular imports**` e acontece quando começamos a dividir nosso projeto em múltiplos arquivos e módulos que tornam-se interdependentes.

Exemplo:

No `app.py`	E então no `views.py`
<pre>from flask import Flask from views import index app = Flask(__name__)</pre>	<pre>from app import app @app.route('/') def index(): return "Hello World"</pre>

Já deu para notar que o problema acontece pois um módulo está tentando importar o outro e entrando em dependência cíclica, conhecida também como circular import ou deadlock.

Para resolver o problema de circular imports uma das abordagens adotadas é alterar a ordem dos imports no ``app.py``

```
from flask import Flask
app = Flask(__name__)
from views import index
```

Porém isto torna o código ``feio``, difícil de manter e fere a legibilidade.

A melhor forma de resolver este problema é com a aplicação do padrão ``application factory`` e **Blueprints** que será discutido adiante.

Flask Lifetime

Um outro problema comum encontrado ao interagir com aplicações Flask ocorre por conta da falta de entendimento dos contextos da aplicação Flask. E dominar o Flask está diretamente relacionado ao entendimento desses **3 estados** da aplicação Flask.

1) Contexto de Configuração:

Um objeto Flask ``app`` assim que é instanciado com ``app = Flask(__name__)`` entra no que chamamos de **contexto de configuração**, nesta fase o aplicativo ainda não tem informações suficientes a respeito do ambiente em que está sendo executado, ele não sabe se está rodando em um servidor web e não tem informações completas sobre quais configurações e extensões estão registradas e é o momento em que podemos construir a aplicação através do uso de factories.

Esta fase é importantíssima pois é nesse momento que podemos registrar rotas e views, registrar blueprints, carregar extensões e mudar parâmetros de configuração do Flask.

O que podemos fazer durante o contexto de configuração?

Adicionar configurações

```
app.config['F00'] = 'bar'
```

Registrar views

```
@app.route('/')
app.add_url_rule('/', ....)
```

Fazer composição e injetar dependências

```
app.add_template_global(function)
```

Registrar blueprints

```
app.register_blueprint(foobar)
```

Iniciar extensões

```
admin = Admin(app)
```

Registrar hooks

```
@app.before_request(...)
@app.error_handler(...)
```

Passar o `app` como parâmetro em factory functions

```
views.configure(app)
extensions.configure(app)
```

2) Contexto de aplicação

Com a aplicação executada (sendo servida por um web server) assim que uma requisição chega ao web server (ex: um usuário digita a URL no navegador ou clicar em um botão) o web server então direciona este pedido ao aplicativo que está em execução e uma nova thread de processamento é iniciada para ser utilizada especificamente neste request (**thread locals**) e o objeto Flask residente desta thread é uma app em estado de Aplicação **Application Context**.

Essa separação de contexto é bastante importante pois é apenas nessa fase que podemos usar objetos compartilhados bastante úteis como `current_app` e o objeto global `g`. Iremos interagir com o **Application Context** ao realizar testes, criar aplicações reutilizáveis com **Blueprints**, criar e instalar extensões e ao utilizar o `shell` ou ferramentas de **CLI** do Flask.

Ao tentar utilizar o objeto `g` fora do contexto veremos

```
RuntimeError: Working outside of application context.
```

O que podemos fazer durante o contexto de aplicação?

Usar o proxy ``current_app``

```
from flask import current_app as app
```

Usar o thread local ``g`` para compartilhar objetos globais.

```
from flask import g
db = g.db
```

3) Contexto de Request

Assim que o roteamento é feito (o servidor web recebe o request e então baseado no endereço da URL roteia a execução de uma função no Flask) a ``app`` entra em estado de request e isso significa que esta aplicação tem agora informações isoladas e específicas deste request como por exemplo a URL, os cookies e os headers HTTP, informações sobre o servidor e sobre o cliente que requisitou e a sessão do usuário. Nesta fase objetos como ``request`` e ``session`` estarão disponíveis.

Ao tentar utilizar ``request`` ou ``session`` fora deste contexto veremos

```
RuntimeError: Working outside of request context.
```

O que podemos fazer durante o request context?

Acessar valores da requisição (passados em GET, POST etc)

```
from flask import request
name = request.form.get('name')
```

Acessar a ``session`` do usuário e os `cookies`

```
from flask import session
session['foo'] = 'bar'
```

Projeto:

Estrutura inicial do projeto “**talkshow**” um sistema para call for papers (submissão de palestras), inspirado no speakerfight.com

As funcionalidades de nosso projeto:

- CLI: Novos eventos poderão ser criados através da linha de comando ``flask addevent``
- CLI: Novos usuários admin poderão ser criados através da linha de comando ``flask adduser``
- UI: Admin poderá gerenciar palestras através de <http://localhost:500/admin>
- UI: Usuários poderão ver os eventos criados em uma lista em ``/`` (**index**)
- UI: Usuário poderá submeter proposta de palestra através de um formulário aberto em ``/<id_do_evento>``
- UI: Interface admin será protegida por usuário e senha em `/login`
- API: Usuários não logados poderão listar eventos em `/api/v1/event/` - **GET**
- API: Usuários logados como admin (BasicAuth) poderão incluir eventos em `/api/v1/event/` - **POST**
- API: Usuários não logados poderão listar propostas em `/api/v1/event/<id_do_evento>` - **GET**
- API: Usuários não logados poderão submeter propostas em `/api/v1/event/<id_do_evento>` - **POST**

A estrutura de arquivos do projeto ``talkshow``:

```
talkshow
├── .env
├── setup.py
├── talkshow/
│   ├── __init__.py
│   ├── app.py
│   ├── blueprints
│   │   └── __init__.py
│   ├── ext
│   │   └── __init__.py
│   └── templates/
│       └── base.html
└── tests/
```

O ambiente

```
cd talkshow
# directorio onde está o `setup.py`

python3.6 -m venv venv
source venv/bin/activate # windows venv/Scripts/activate

pip install --upgrade pip

pip install python-dotenv
pip install flask
```

Parte 2: A linha de comando e o `flask` CLI

Com o ambiente configurado agora podemos executar **flask** na linha de comando e ver a saída do flask CLI ref: <http://flask.pocoo.org/docs/1.0/cli/>

```
$ flask
flask.cli.NoAppException: Could not locate a Flask application. You did
not provide the "FLASK_APP" environment variable, and a "wsgi.py" or
"app.py" module was not found in the current directory.
Usage: flask [OPTIONS] COMMAND [ARGS]...
```

```
$ export FLASK_APP=hello.py
$ export FLASK_ENV=development
$ flask run
```

Options:

```
--version  Show the flask version
--help     Show this message and exit.
```

Commands:

```
routes  Show the routes for the app.
run      Runs a development server.
shell    Runs a shell in the app context.
```

Os comandos disponíveis por default são **run**, **routes** e **shell**.

Variáveis de ambiente.

A primeira coisa a fazer é configurar as variáveis de ambiente, isso pode ser feito manualmente ou incluindo no arquivo `.env`

```
export FLASK_APP=talkshow/app.py
export FLASK_ENV=development
```

No windows usamos `set FLASK_ENV=talkshow/app.py` ao invés de export

Agora tentamos rodar o `flask run`

```
$ flask run
* Serving Flask app "talkshow/app.py" (lazy loading)
* Environment: development
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 130-128-431
```

O servidor Flask irá executar com sucesso! mas pera aí.. não criamos a app ainda... repare no (**lazy loading**) o servidor irá procurar a existência de `talkshow/app.py` apenas quando chegar o primeiro request mas vamos resolver isso!

Editamos o `app.py` utilizando o conceito de application factory

```
from flask import Flask

def create_app():
    """Creates a new Flask app"""
    app = Flask(__name__)
    app.config['SECRET_KEY'] = 'secret'
    return app
```

Se tentarmos acessar / agora veremos o erro 404, mas isso é um bom sinal! significa que o servidor está processando a requisição porém não encontrou nada para servir, pois nós ainda não criamos nenhuma rota em nosso app!

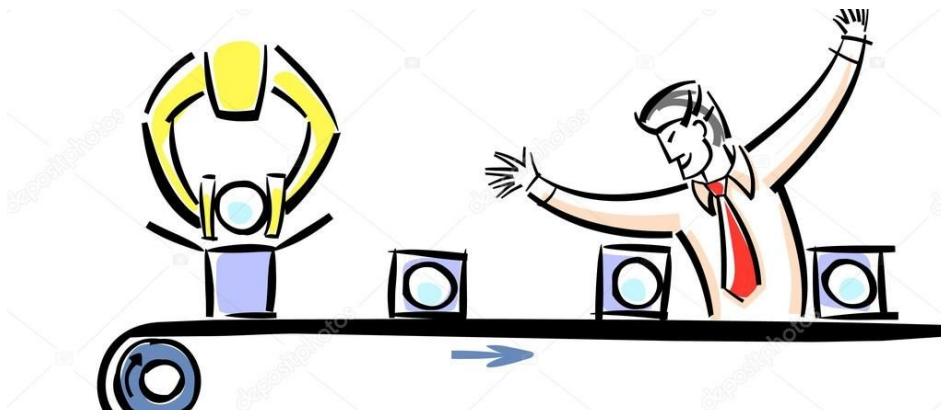
```
127.0.0.1 - - [21/Aug/2018 16:39:20] "GET / HTTP/1.1" 404 -
127.0.0.1 - - [21/Aug/2018 16:39:20] "GET /favicon.ico HTTP/1.1" 404 -
```

Adicionando comandos customizados.

Nosso objetivo agora é adicionar um comando que ao ser executado adicione um novo evento ao banco de dados:

```
flask addevent -n FlaskConf -d 2018-08-25
```

Para isso vamos estender o flask cli e a melhor maneira de estender aplicações Flask é utilizando o conceito de factories que aplicamos anteriormente.



Imagine que a criação do ``app`` é uma linha de produção (por isso o nome factory) e nessa esteira|pipeline passamos o ``app`` de ``setor em setor`` para que mais trabalho seja feito em cima deste objeto. ex:

```
app = create_app()  
# fase 1 criar o objeto app básico
```

```
ext.db.configure(app)  
# fase 2 passamos o app para o `ext.db` para que ele faça o que quiser  
com app ex: configurar o acesso a banco de dados
```

```
ext.cli.configure(app)  
# fase 3 passamos o app para o `ext.cli` para configurar os comandos  
customizados
```

E vamos repetindo até o término da nossa linha de produção/factory, vamos começar criando um comando customizado no CLI e depois partimos para o banco de dados.

Crie o arquivo `talkshow/ext/cli.py` com o seguinte conteúdo.

```
import click

def configure(app):
    """Attach new commands in to app"""

    @app.cli.command()
    @click.option('--name', '-n', required=True)
    @click.option('--date', '-d', required=True)
    def addevent(name, date):
        """Creates a new event entry"""
        click.echo(f"Name: {name} Date: {date}")
```

e agora alteramos o `app.py` incluindo este `configure()` factory

```
from flask import Flask
from talkshow.ext import cli

def create_app():
    """Creates a new Flask app"""
    app = Flask(__name__)
    app.config['SECRET_KEY'] = 'secret'
    # extensions
    cli.configure(app) # <-- `app` passado para o cli factory
    return app
```

Agora experimente rodar o novo comando no terminal

```
flask --help
flask addevent --help
flask addevent -n RustConf -d 2018-08-28
```

Por enquanto o comando ainda não faz muita coisa útil, **apenas imprime o nome e a data do evento** mas logo veremos como inserir esses dados no banco de dados!

O click também adiciona uma opção `--help` automaticamente:

```
$ flask addevent --help
Usage: flask addevent [OPTIONS]
```

Creates a new event entry

Options:

```
-n, --name TEXT [required]
-d, --date TEXT [required]
--help             Show this message and exit.
```

Parte 3: Banco de dados NoSQL (MongoDB/TinyMongo)

Nesta aplicação usaremos **MongoDB**, porém para evitar a instalação do mongodb server usaremos o **TinyMongo** que é uma versão file-based do **MongoDB** (apenas para uso durante o desenvolvimento)

Instalando as dependências

```
pip install pymongo
pip install tinymongo
```

Agora configuramos o cliente para o banco de dados **MongoDB** no arquivo `talkshow/ext/db.py` novamente utilizando o padrão factory.

```
from pymongo import MongoClient
from tinymongo import TinyMongoClient

def configure(app):
    """Creates a new database connection"""

    if app.env == "production":
        client = MongoClient(host="localhost")
    else:
        client = TinyMongoClient('database')

    db_name = app.config.get('MONGODB_NAME', 'db')
    app.db = client[db_name]
```

NOTA: Caso venha a usar um banco de dados relacional (SQL) em conjunto com ORMs como o SQLAlchemy ou Peewee é recomendado utilizar extensões que provém controle de sessão em cima do global object ``g`` ou implementar o controle de pool de conexões utilizando ``g`` e request **hooks**.

E agora precisamos passar o ``app`` para o db factory alterando o ``talkshow/app.py``

```
from flask import Flask
from talkshow.ext import db
from talkshow.ext import cli

def create_app():
    """Creates a new Flask app"""
    app = Flask(__name__)
    app.config['SECRET_KEY'] = 'secret'
    # extensions
    db.configure(app) # <-- `app` passado para o db factory
    cli.configure(app)
    return app
```

IMPORTANTE: o ``db`` precisa ser inicializado antes de todas as outras extensões que o forem utilizar.

Já temos nossa estrutura de **CLI** e banco de dados mas para fazer o ``addevent`` funcionar gravando as informações no banco de dados.

Inserindo no banco de dados

Agora que temos o banco de dados podemos implementar o comando ``addevent`` no ``talkshow/ext/cli.py``

```
import click

def configure(app):
    """Attach new commands in to app"""

    @app.cli.command()
    @click.option('--name', '-n', required=True)
    @click.option('--date', '-d', required=True)
    def addevent(name, date):
        """Creates a new event entry"""
        event = app.db['events'].insert_one({'name': name, 'date': date})
        click.echo(f"{event.inserted_id} cadastrado com sucesso!")
```

A partir de agora podemos gravar novos eventos usando o CLI

```
$ flask addevent -n FlaskConf -d 2018-08-25  
cd50a7e4a59311e88889b46bfca38ce8 cadastrado com sucesso!
```

E para verificar se foi realmente gravado pode dar uma olhada no arquivo `databases/db.py` que foi criado pelo TinyMongo ou então através do **flask shell**

```
$ flask shell  
Python 3.6.6 (default, Jul 19 2018, 14:25:17)  
App: talkshow.app [development]  
Instance: /talkshow/instance  
>>> app.db['events'].find_one({'_id': 'cd50a7e4a59311e88889b46bfca38ce8'})  
{'name': 'FlaskConf', 'date': '2018-08-25', '_id':  
'cd50a7e4a59311e88889b46bfca38ce8'}  
>>>
```

Experimente listar todos os eventos cadastrados com:

```
>>> [item for item in app.db['events'].find()]  
>>> exit()
```

Cadastre alguns eventos usando o CLI

```
$ flask addevent -n "Python Brasil" -d 2018-10-19
```

Parte 4: View e processamento de Formulário usando Blueprint

Agora que temos um **banco de dados** e uma interface **CLI** vamos criar uma parte na nossa web **UI**, uma página com um formulário onde o usuário pode submeter uma proposta de palestra para nosso sistema. Usaremos o **flask-wtf** e o **flask-bootstrap** para ajudar nesse trabalho.

```
pip install flask-wtf  
# fornece estruturas para criação de formulários  
  
pip install flask-bootstrap  
# fornece templates básicos para uso com os formulários
```

Agora vamos inicializar a extensão Flask-Bootstrap

crie o arquivo ``talkshow/ext/bootstrap.py``

```
from flask_bootstrap import Bootstrap

def configure(app):
    """Configures the bootstrap extension"""
    Bootstrap(app)
```

E inicialize através do ``talkshow/app.py``

```
from flask import Flask
from talkshow.ext import db
from talkshow.ext import cli
from talkshow.ext import bootstrap

def create_app():
    """Creates a new Flask app"""
    app = Flask(__name__)
    app.config['SECRET_KEY'] = 'secret'
    # extensions
    db.configure(app)
    cli.configure(app)
    bootstrap.configure(app) # <-- `app` passado para o bootstrap factory
    return app
```

Para terminar a configuração do Flask-Bootstrap precisamos criar um template base que re-utilizamos em todos os nossos outros templates.

em ``talkshow/templates/base.html``

```
{% extends "bootstrap/base.html" %}
{% block content %}
<div class="container">
  <div class="row">
    {% block page %} {% endblock %}
  </div>
</div>
{% endblock %}
```

NOTA: Poderíamos ter chamado ``Bootstrap(app)`` diretamente no ``app.py`` mas isso não é recomendado pois foge do nosso padrão de arquitetura e dificulta caso no futuro queira configurar mais opções desta extensão.

Blueprint

Blueprint é um padrão de extensão de aplicações Flask e implementa quase todas as mesmas funcionalidades de uma instância de **Flask** como a nossa **app** porém o Blueprint não é um app e sim apenas um projeto, ou modelo, de como o app deve ser, e podemos registrar vários blueprints.

Em projetos pequenos podemos criar funções e registrar rotas com **@app.route** diretamente nos arquivos e módulos da aplicação sem se preocupar com a estrutura final e ordem de carregamento das coisas.

Mas em projetos grandes, ou que tendem a crescer é recomendado que **qualquer rota** (página/view) adicionada ao projeto seja adicionada na forma de **@blueprint.route** que é apenas uma receita de como a rota deve ser adicionada, para que ela seja finalmente registrada apenas quando o blueprint for registrado a uma app.

Adicionando UI ao nosso projeto

Já temos uma interface **CLI** e agora criaremos a nossa interface web **UI** (interface gráfica do usuário) chamaremos de **webui**

Vamos começar adicionando uma rota que ao ser requisitada em <http://localhost:5000/> irá listar todos os eventos que cadastramos no nosso app.

edite **talkshow/blueprints/webui.py**

```
from flask import Blueprint, render_template
from flask import current_app as app
bp = Blueprint("webui", __name__)

@bp.route('/')
def index():
    """View to list all registered events"""
    events = app.db['events'].find()
    return render_template('index.html', events=events)

@bp.route('/<event_id>', methods=['GET', 'POST'])
def event(event_id):
    """A form to submit a talk to the selected event"""
    return "TODO: Implementar o form"

def configure(app):
    """Register the Blueprint to the app"""
    app.register_blueprint(bp)
```


Repare que a rota `/` precisa de um template que criaremos em `talkshow/templates/index.html`

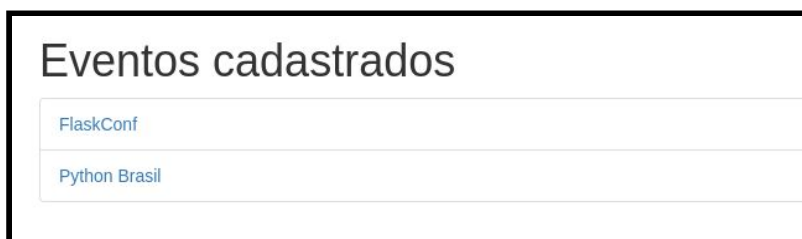
```
{% extends "base.html" %}
{% block page %}
<h1>Eventos cadastrados</h1>
  <ul class="list-group">
    {% for event in events %}
      <li class="list-group-item">
        <a href="{{ url_for('webui.event', event_id=event['_id']) }}">
          {{ event['name'] }}
        </a>
      </li>
    {% endfor %}
  </ul>
{% endblock %}
```

Agora para colocar este blueprint em ação precisamos registrar ele no `talkshow/app.py`

```
from flask import Flask
from talkshow.ext import db
from talkshow.ext import cli
from talkshow.ext import bootstrap
from talkshow.blueprints import webui

def create_app():
    """Creates a new Flask app"""
    app = Flask(__name__)
    app.config['SECRET_KEY'] = 'secret'
    # extensions
    db.configure(app)
    cli.configure(app)
    bootstrap.configure(app)
    # blueprints
    webui.configure(app) # <-- registro do webui
    return app
```

Acesse <http://localhost:5000/> e veja a lista de eventos cadastrados:



Formulários

Para processar formulários usaremos o `wtforms` biblioteca de formulários mais utilizada em projetos Flask, começaremos editando `talkshow/blueprints/webui.py`

```
import datetime
import wtforms as wtf
from flask_wtf import FlaskForm
from flask import Blueprint, request, render_template, abort
from flask import current_app as app

bp = Blueprint("webui", __name__)

@bp.route('/')
def index():
    ...

class ProposalForm(FlaskForm):
    """Form to register new proposals to events"""
    name = wtf.StringField('name', validators=[wtf.validators.DataRequired()])
    email = wtf.StringField('email', validators=[wtf.validators.Email()])
    title = wtf.StringField('title', validators=[wtf.validators.DataRequired()])
    description = wtf.TextAreaField(
        'description', validators=[wtf.validators.DataRequired()]
    )
    submit = wtf.SubmitField("Enviar")

@bp.route('/<event_id>', methods=['GET', 'POST'])
def event(event_id):
    """A form to submit a talk to the selected event"""
    event = app.db['events'].find_one({'_id': event_id})
    if not event:
        abort(404, 'Evento não encontrado')

    form = ProposalForm(request.form)

    if form.validate_on_submit():
        # Se estamos no meio de um submit válido preparamos os dados
        proposal = form.data.copy()
        proposal['event_id'] = event_id
        proposal['date'] = datetime.datetime.today().date().isoformat()
        proposal['approved'] = False
        # e gravamos no banco de dados
        app.db['talks'].insert_one(proposal)

        return render_template('thanks.html', proposal=proposal, event=event)

    # exibimos o formulário limpo
    return render_template('event.html', form=form, event=event)
```

```
def configure(app):  
    ...
```

Agora temos um formulário **ProposalForm** criado com o **wtforms** e seus fields e validadores, o código da view **event** ganhou um pouco mais de funcionalidades como validação de formulário e retornos condicionais. E além disso temos mais 2 templates que precisamos criar:

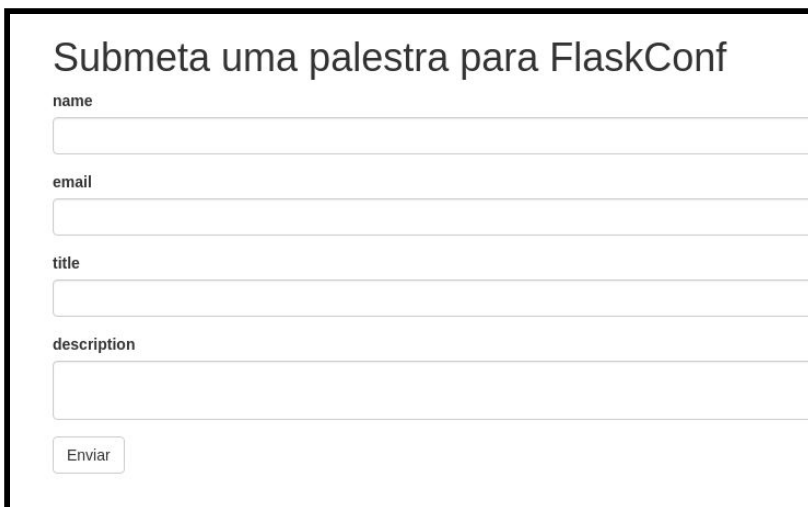
`talkshow/templates/event.html`

```
{% extends "base.html" %}  
{% import "bootstrap/wtf.html" as wtf %}  
{% block page %}  
    <h1>Submeta uma palestra para {{ event['name'] }}</h1>  
    {{ wtf.quick_form(form) }}  
{% endblock %}
```

`talkshow/templates/thanks.html`

```
{% extends "base.html" %}  
{% block page %}  
<h1>  
    Olá {{ proposal['name'] }},  
    Obrigado por submeter {{ proposal['title'] }}  
    para o evento {{ event['name'] }}  
</h1>  
{% endblock %}
```

Agora já é possível acessar o formulário e submeter palestras!



Submeta uma palestra para FlaskConf

name

email

title

description

Enviar

Parte 5: Interface Administrativa

O próximo passo é permitir que o usuário admin consiga visualizar e selecionar as palestras aprovadas através de uma interface administrativa.

Para isto utilizaremos o **flask-admin** e para começar vamos instalar a extensão

```
pip install flask-admin
```

Agora basta criarmos a factory para o admin em `talkshow/ext/admin.py`

```
from flask_admin import Admin
from flask_admin.base import AdminIndexView # noqa
from flask_admin.contrib.pymongo import ModelView # noqa

def configure(app):
    """Adds admin extension to app"""
    app.admin = Admin(app, "TalkShow", template_mode="bootstrap2")
```

e inicializar no `talkshow/app.py`

```
from flask import Flask
from talkshow.ext import db
from talkshow.ext import cli
from talkshow.ext import bootstrap
from talkshow.ext import admin
from talkshow.blueprints import webui

def create_app():
    """Creates a new Flask app"""
    app = Flask(__name__)
    app.config['SECRET_KEY'] = 'secret'
    app.config['FLASK_ADMIN_SWATCH'] = 'cerulean' # ← tema do admin
    # extensions
    db.configure(app)
    cli.configure(app)
    bootstrap.configure(app)
    admin.configure(app) # ← `app` passado para o admin factory
    # blueprints
    webui.configure(app)
    return app
```

Com isso já é possível acessar <http://localhost:500/admin> mas você verá apenas uma tela em branco com a barra de menus do Flask-Admin então o próximo passo é adicionar novas funcionalidades ao **Flask-Admin**.

NOTA: os temas disponíveis para serem usados em FLASK_ADMIN_SWATCH podem ser encontrados em <http://bootswatch.com>

Vamos adicionar ao flask admin uma view que permitirá a administração das propostas submetidas, para isso adicionaremos esta funcionalidade ao blueprint **webui**

`talkshow/blueprints/webui.py`

```
from talkshow.ext.admin import ModelView # base Admin model view

bp = Blueprint("webui", __name__)

...

# Flask Admin models
class ProposalAdminForm(ProposalForm):
    """Extends Proposal form to use in admin interface"""
    event_id = wtfl.StringField()
    approved = wtfl.BooleanField()
    submit = None

def format_event(self, request, obj, fieldname, *args, **kwargs):
    """Returns the name for the event (see also get_list)"""
    return app.db['events'].find_one({'_id': obj['event_id']})['name']

class AdminProposal(ModelView):
    """The proposal admin item"""
    can_create = False # Admins não podem criar registros apenas editar
    column_list = ('event_id', 'name', 'title', 'approved')
    form = ProposalAdminForm
    # o event_id exibe apenas o _id do registro, precisamos formatar.
    column_formatters = {'event_id': format_event}

def configure(app):
    """Register the Blueprint to the app"""
    app.register_blueprint(bp)
    # register the new admin model
    app.admin.add_view(AdminProposal(app.db['proposal'], 'Proposals'))
```

Agora ao visitar o admin veremos no menu o item “Proposals”:

TalkShow Home Proposals				
List (1)	With selected ▾			
	Event Id	Name	Title	Approved
	FlaskConf	Hello	Hello	

NOTA: A maneira mais eficaz de exibir o ``name`` de event ao invés de ``event_id`` seria sobrescrevendo o método ``get_list`` do ``ModelView`` pois desta forma apenas uma query seria feita ao banco de dados, utilizando formatters cada registro é processado individualmente.

EXTRA: é possível adicionar ``actions`` ao Admin, por exemplo, selecionar várias propostas e mudar o estado de aprovação delas ao mesmo tempo, para isso basta usar o código:

```
from flask import flash # suporte a mensagens de alerta
from talkshow.ext.admin import ModelView # base Admin model view
from flask_admin.actions import action # suporte a actions

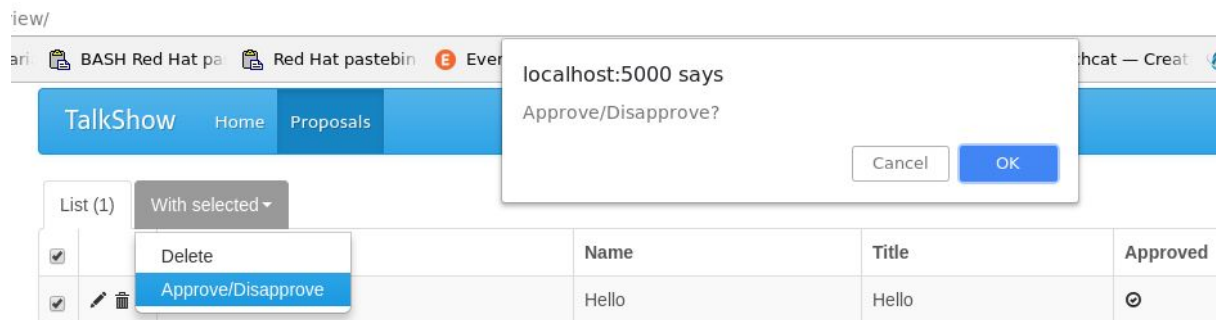
....

class AdminProposal(ModelView):
    ...

    @action(
        'toggle_approval',
        'Approve/Disapprove',
        'Approve/Disapprove?'
    )
    def action_toggle_publish(self, ids):
        for _id in ids:
            model = self.coll.find_one({'_id': _id})
            model['approved'] = not model['approved']
            self.coll.update({'_id': _id}, model)

        flash(f'{len(ids)} items published/Unpublished.', 'success')
```

e então teremos agora a action no menu ``with selected``



Tem muito mais coisas que podem ser customizadas no Flask Admin

<https://flask-admin.readthedocs.io/en/latest/>

Parte 6: Rest API

Agora que já temos a interface CLI e a interface web UI (com admin) falta incluir a terceira interface a REST API!

Para isso vamos usar a extensão `'flask_restful'`

```
pip install flask-restful
```

Começaremos incluindo um novo arquivo em `'talkshow/blueprints/restapi.py'`

```
from flask import Blueprint
from flask import current_app as app
from flask_restful import Api, Resource, reqparse

bp = Blueprint('restapi', __name__, url_prefix='/api/v1')
api = Api(bp)

event_post_parser = reqparse.RequestParser()
event_post_parser.add_argument('name', required=True)
event_post_parser.add_argument('date', required=True)

class Event(Resource):
    def get(self):
        return {'events': list(app.db['events'].find())}

    def post(self):
        event = event_post_parser.parse_args()
        new = app.db['events'].insert({'name': event.name, 'date': event.date})
        return {'event created': new.inserted_id}, 201

proposal_post_parser = reqparse.RequestParser()
proposal_post_parser.add_argument('name', required=True)
proposal_post_parser.add_argument('email', required=True)
proposal_post_parser.add_argument('title', required=True)
proposal_post_parser.add_argument('description', required=True)
```

```

class EventItem(Resource):
    def get(self, event_id):
        event = app.db['events'].find_one({'_id': event_id})
        proposals = app.db['proposal'].find({'event_id': event_id})
        return {'event': event, 'proposals': list(proposals)}

    def post(self, event_id):
        event = app.db['events'].find_one({'_id': event_id})
        prop = proposal_post_parser.parse_args()
        new = app.db['proposal'].insert({
            'event_id': event['_id'],
            'name': prop.name,
            'email': prop.email,
            'title': prop.title,
            'description': prop.description,
            'approved': False
        })
        return {'proposal created': new.inserted_id}, 201

def configure(app):
    """Initialize API and register blueprint"""
    api.add_resource(Event, '/event/')
    api.add_resource(EventItem, '/event/<event_id>')
    app.register_blueprint(bp)

```

Utilizando a API

Criar um novo evento: com **httpie**

```

$ http --form POST localhost:5000/api/v1/event/ name="Cobol Conf"
HTTP/1.0 400 BAD REQUEST
{
  "message": {
    "date": "Missing required parameter"
  }
}

```

O `RequestParser` não vai deixar processar um POST sem campos requeridos, precisamos informar o campo `date`

```

$ http --form POST localhost:5000/api/v1/event/ name="Cobol Conf"
date="2018-08-25"

```

ou utilizando o client **curl**:


```
$ curl --request POST --header "Content-Type: application/json" -d
'{"name": "Cobol Conf", "date": "2019-01-01"}'
http://localhost:5000/api/v1/event/
```

resultado

```
HTTP/1.0 201 CREATED
{
  "event created": "e96c66d2a68511e8b50fb46bfca38ce8"
}
```

Criando uma nova proposta de palestra para o evento **Cobol Conf**:

```
$ http --form POST
localhost:5000/api/v1/event/cd50a7e4a59311e88889b46bfca38ce8 name=Bruno
email=hi@ho.com title='Cobol is Great' description='Cobol is the best'

HTTP/1.0 201 CREATED
{
  "proposal created": "223cb592a68711e8874cb46bfca38ce8"
}
```

Listando todas as propostas de um determinado evento:

```
$ http --form GET
localhost:5000/api/v1/event/cd50a7e4a59311e88889b46bfca38ce8

HTTP/1.0 200 OK
{
  "event": {
    "_id": "cd50a7e4a59311e88889b46bfca38ce8",
    "date": "2019-01-01",
    "name": "Cobol Conf"
  },
  "proposals": [
    {
      "_id": "223cb592a68711e8874cb46bfca38ce8",
      "description": "Cobol is the best",
      "email": "hi@ho.com",
      "event_id": "cd50a7e4a59311e88889b46bfca38ce8",
      "name": "Bruno",
      "title": "Cobol is Great"
    }
  ]
}
```

EXTRA: Podemos adicionar documentação da API com OpenAPISpec

```
pip install flasgger
```

`talkshow/ext/apidocs.py`

```
from flasgger import Flasgger

def configure(app):
    """Starts openapispec"""
    Flasgger(app)
```

`talkshow/app.py`

```
from talkshow.ext import apidocs
...

def create_app():
    ...
    apidocs.configure(app) # <-- `app` passado para o admin factory
    # blueprints
```

`talkshow/blueprints/restapi.py`

```
class Event(Resource):
    ...
    def post(self):
        """Creates new event
        ---
        parameters:
          - in: body
            name: body
            schema:
              id: Event
              properties:
                name:
                  type: string
                date:
                  type: string
        responses:
          201:
            description: Success or failure message
            schema:
              properties:
                event_created:
                  type: string
                description: The id of the created event
        """
```

Agora acesse <http://localhost:5000/apidocs>

Parte 7: Protegendo o app com login/senha

Agora está quase tudo pronto! temos **CLI**, **web UI** e **REST API**, as 3 interfaces importantes estão implementadas! mas estão inseguras! precisamos proteger com login e senha!

```
pip install flask-simplelogin
```

Agora criaremos `talkshow/ext/login.py`

```
from flask import current_app as app
from flask_simplelogin import SimpleLogin
from werkzeug.security import check_password_hash, generate_password_hash

def login_checker(user):
    """Valida o usuário e senha para efetuar o login"""
    username = user.get('username')
    password = user.get('password')
    if not username or not password:
        return False
    existing_user = app.db['users'].find_one({'username': username})
    if not existing_user:
        return False
    if check_password_hash(existing_user.get('password'), password):
        return True
    return False

def create_user(username, password):
    """Registra um novo usuário caso não esteja cadastrado"""
    if app.db['users'].find_one({'username': username}):
        raise RuntimeError(f'{username} já está cadastrado')
    user = {'username': username,
            'password': generate_password_hash(password)}
    return app.db['users'].insert_one(user)

def configure(app):
    """Adds login control"""
    SimpleLogin(app, login_checker=login_checker)
```

Carregaremos no `talkshow/app.py`

```
from talkshow.ext import login
...
def create_app():
    ...
    login.configure(app) # <-- `app` passado para o login factory
    # blueprints
```

Adicionamos o comando `flask adduser` ao `talkshow/ext/cli.py`

```
...
from talkshow.ext.login import create_user

def configure(app):
    ...

    @app.cli.command()
    @click.option('--username', '-u', required=True)
    @click.option('--password', '-p', required=True)
    def adduser(username, password):
        """Creates a new admin user"""
        user = create_user(username, password)
        click.echo(f"{user.inserted_id} cadastrado com sucesso!")
```

criamos o usuário admin

```
$ flask adduser -u admin -p 1234
```

E então protegeremos o `talkshow/ext/admin.py`

```
from flask_admin import Admin
from flask_admin.base import AdminIndexView # noqa
from flask_admin.contrib.pymongo import ModelView # noqa

from flask_simplelogin import login_required

# decorate Flask-Admin view via Monkey Patching
AdminIndexView._handle_view = login_required(AdminIndexView._handle_view)
ModelView._handle_view = login_required(ModelView._handle_view)

def configure(app):
    """Adds admin extension to app"""
    app.admin = Admin(app, "TalkShow", template_mode="bootstrap2")
```

E os métodos POST da API em `talkshow/blueprint/restapi.py`

```

...
from flask_simplelogin import login_required
...

class Event(Resource):
    ...

    @login_required(basic=True)
    def post(self):
        ...

```

com a adição do decorator `login_required` com o parâmetro `basic=True` agora podemos garantir o login usando Basic Auth em nossa API

NOTA: O mais recomendado é usar JWT com `jwt=True` mas isso exige uma fase a mais na configuração dos tokens

```

$ http --json -a admin:senhaerrada --form POST
localhost:5000/api/v1/event/ name="New Event" date="2019-02-02"

```

```

HTTP/1.0 401 UNAUTHORIZED
Content-Type: application/json
WWW-Authenticate: Basic realm="Login Required"
"Invalid credentials"

```

e usando a senha correta

```

$ http --json -a admin:1234 --form POST localhost:5000/api/v1/event/
name="New Event" date="2019-02-02"

```

```

HTTP/1.0 201 CREATED

```

```

Set-Cookie:
session=.eJyrVirOzC3ISY1PSiz0TI5PLC3JULIqKSpN1YFJ50Snp6emxGfmoYmXFqcW5SX
mpipZKSWm5AKlawFW-hxu.Dl_S_w.8dzbdNorqXZQUFjcvSlUlqD-SRs; HttpOnly;
Path=/
Vary: Cookie

```

```

{
    "event_created": "b0e4345ea69211e8a3e8b46bfca38ce8"
}

```

Parte 8: Testando a API

Para finalizar o nosso projeto só falta criar alguns testes de API usando o **'flask web client'**

como a pasta de testes está fora do package precisamos ter o projeto adicionado ao PATH, a maneira mais fácil é exportando **'export PYTHONPATH=\$PWD'**

A mais correta é criando um **setup.py** e instalando o projeto

```
from setuptools import setup

requirements = [
    'flask',
    'pymongo',
    'tinymongo',
    'flask-simplelogin',
    'flask-admin',
    'flask-wtf',
    'flask-restful',
    'flask-pytest',
    'flask-bootstrap',
    'python-dotenv',
    'flasgger'
]

setup(
    name='talkshow',
    version='0.0.1',
    description="Call for papers system",
    packages=['talkshow'],
    include_package_data=True,
    install_requires=requirements
)
```

Com o setup.py configurado podemos então instalar o nosso projeto com:

```
$ pip install -e .
```

usaremos o flask-pytest

```
pip install flask-pytest
```

Criaremos o `tests/conftest.py`

```
import pytest
from base64 import b64encode
from talkshow.app import create_app

@pytest.fixture(scope="module")
def app():
    """Flask Pytest uses it"""
    return create_app()

@pytest.fixture(scope="module")
def auth():
    """The Basic Auth credentials for testing"""
    credentials = b64encode(bytes("admin:1234", 'ascii')).decode('ascii')
    data = {'Authorization': 'Basic ' + credentials}
    return data
```

E então nossos testes em `tests/test_api.py`

```
def test_can_create_event(app, auth):
    """Asserts an event can be created via API"""
    event_data = {"name": "Puggies Convention", "date": "2018-09-09"}
    with app.test_client() as client:
        created = client.post('/api/v1/event/', json=event_data, headers=auth)
        assert created.status_code == 201

def test_can_list_event(app):
    """Asserts event can be listed"""
    with app.test_client() as client:
        events = client.get('/api/v1/event/')
        assert events.status_code == 200
        assert 'events' in events.json
```

No terminal execute os testes

```
py.test tests -s -v
```

```
collected 2 items
tests/test_api.py::test_can_create_event PASSED
tests/test_api.py::test_can_list_event PASSED
```

FIM! - Bruno Rocha - @rochacbruno - #FlaskConf 2018