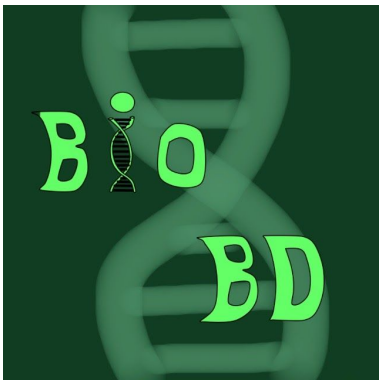
TUTORIAL DE UTILIZAÇÃO DO FLASK
BioBD

Última atualização: 09/04/2020 18h35min - versão 0.0



Este documento é destinado ao leitor que pretende começar um projeto *web* baseado em Flask junto ao BioBD. Aqui está descrito as etapas de configuração e utilização do *microframework* para a construção de uma aplicação em Python, especialmente da maneira que é feita nos projetos do laboratório de pesquisa BioBD na PUC-Rio.
Para o acompanhamento deste tutorial é necessário que o leitor tenha familiaridade com a linguagem Python e suas bibliotecas padrão.
Além disso, será um facilitador caso o leitor tenha conhecimento do Anaconda (Python) e das linguagens HTML, CSS (Bootstrap) e Javascript (ES6) junto a suas bibliotecas (como jQuery).

Sumário

1 - Pré-requisitos e configurações iniciais

Este capítulo explica e detalha o que é necessário para iniciar/contribuir em um projeto Flask do BioBD. Ao final do capítulo você estará com tudo preparado para começar a trabalhar em um projeto.

1.1 Anaconda

Anaconda é uma distribuição de Python e R que simplifica o gerenciamento de pacotes e ambientes virtuais de um projeto. O Anaconda possui muitas outras funcionalidades direcionadas ao processamento de dados em larga escala, ciência de dados, *machine learning* e afins, no entanto, tais funcionalidades não pertencem ao escopo deste tutorial e não serão aqui mencionadas. Além disso, a linguagem do Anaconda utilizada será evidentemente a distribuição em Python.

Para instalar o Anaconda basta entrar no link de download da distribuição em Python, selecionando o instalador da **versão 3.7** e, após baixado o instalador, seguir as instruções de instalação.

O Anaconda será utilizado para o gerenciamento do projeto por meio de um ambiente virtual Python isolado, que separa as dependências específicas do projeto em uma única "pasta", mantendo todos os pacotes e configurações específicas do projeto em um único lugar no computador. Portanto, sempre que o código do projeto for rodado, é necessário que o ambiente esteja ativo. Para a criação do ambiente, crie uma pasta onde os arquivos do projeto ficarão e em seguida, execute o seguinte código no **Anaconda Prompt**:

```
conda create -n meu-ambiente python=x.x.x
```

Sendo "x.x.x" a versão a ser utilizada do Python. Em seguida, para ativar o ambiente, execute o seguinte:

```
conda activate meu-ambiente
```

A partir de agora você está ativo no ambiente que criou. Caso queira desativar o ambiente, você deve executar:

```
conda deactivate
```

Isto é tudo necessário para utilizar a ferramenta de ambientes virtuais do Anaconda. Para maiores detalhes sobre os comandos ou consultas sobre outros comandos do Prompt, acesse esta referência do Anaconda.

1.2 Flask

Flask é um *microframework* baseado nos pacotes Werkzeug e Jinja2 utilizado para criar a aplicação *web* dos projetos. Ele ajudará no gerenciamento de *requests* ao site (Werkzeug) e na geração das páginas HTML (Jinja2). Para utilizá-lo no projeto será necessário usar o instalador **pip**:

```
pip install Flask
```

Junto com o módulo principal do Flask serão baixadas algumas outras dependências. Por uma questão de organização, é necessário que o seguinte código seja executado:

```
pip freeze > requirements.txt
```

Isto criará um arquivo texto que guarda todas as dependências do projeto, junto de

suas versões. Dessa forma, quando o código for clonado de algum repositório basta executar o mesmo *pip install* do início da seção, porém passando este arquivo como argumento, para que todas as dependências envolvidas no projeto sejam baixadas de uma vez.

Observação: caso você esteja entrando em um projeto Flask, você deverá fazer a instalação das dependências do projeto por meio do seguinte comando:

```
pip install -r requirements.txt
```

Posteriormente neste tutorial será explicado como ficará a estrutura do código do projeto. A partir do agora o Flask já está instalado no projeto.

1.3 VPN e conexão ao banco de dados

Para conectar-se ao banco quando em projetos do BioBD é necessário estabelecer a conexão via SSH a uma VPN disponibilizada para nós pela PUC-Rio. Em outras palavras, conectamo-nos remotamente a uma máquina virtual onde o banco (produção, testes e desenvolvimento) está. O meio de estabelecer essa conexão é conectando-se diretamente à porta da VM disponibilizada ao mesmo tempo que executa o arquivo .bat para se conectar usando o **OpenVPN.** Portanto, para realizar essa conexão é necessário que você tenha acesso à cloud-di (ter em mãos as credenciais de acesso, como porta a se conectar, senha do usuário, número da VM, etc.), além de ter baixado o programa OpenVPN (versão grátis).

Tendo tudo preparado para a conexão, clique duas vezes no arquivo **connect.bat**, se estiver utilizando Windows, ou **connect.sh** se estiver utilizando Linux. Uma tela do Command Prompt irá aparecer. Você deve esperar até que a mensagem **"Initialization Sequence Completed"** apareça. Em seguida abra o **Command Prompt** do Windows ou o **Terminal** do Linux e execute:

```
ssh -L 5001:localhost:XXXX cloud-di@vmYY.cloud.inf.puc-rio.br
```

Onde "XXXX" e "YY" representam o número da porta e código da máquina virtual cujo acesso fora concedido (Todos estes valores, arquivos e senha cedidos pelo DI são informados via *email*).

Caso a máquina virtual esteja disponível, um pedido de senha será feito. A senha pedida é a senha de acesso à máquina virtual que se encontra na mesma pasta do arquivo connect.bat utilizado anteriormente (nesse caso você deverá consultar a senha específica do projeto).

Ao informar a senha corretamente, o Command Prompt irá refletir o Terminal da máquina virtual (que é um sistema Ubuntu). A partir deste momento você já está devidamente conectado à máquina virtual e já pode realizar mudanças no banco.

Você também pode fazer a conexão SSH pelo Bitvise que fará a mesma coisa que o comando no Prompt, porém com suporte FTP, permitindo a transferência de arquivos.

Caso necessário, você pode entrar no banco e visualizar esquemas, tabelas, colunas, etc.. Para isso, existe duas maneiras:

1. **Shell do Postgresql:** Ao executar:

```
psql -h host -U user
```

Você entrará no Prompt do Postgresql, podendo visualizar ou modificar informações que estão no banco. Caso não saiba os comandos, digite **psql --help** ou consulte a documentação do terminal interativo. Note que os comandos para o banco são sempre em SQL.

2. **pgAdmin4:** após fazer o download do programa, mude o código do comando SSH devidamente e adicione o **localhost** e a porta **5001** para acessar o banco por meio desta interface gráfica. O resultado será idêntico ao comando no Prompt, porém com a facilidade de interagir com uma GUI.

Uma observação: é possível que seja necessário mudar o *path* para o arquivo .ovpn cedido para conexão (precisamente vpnclouddipuc-projeto.ovpn). Para isso, abra o arquivo connect.bat (ou .sh) em um editor de texto (VSCode, Notepad++, Text Editor do Linux, Vim, etc.) e altere a rota do arquivo para o local onde o arquivo se encontra no seu computador.

2 - Construção de um projeto CRUD

Neste capítulo você escreverá o código de um projeto CRUD (*Create, Read, Update and Delete resources*) do zero. O início trata de explicações teóricas e configurações iniciais para um projeto deste tipo, enquanto final trata da parte prática, como código das páginas de explicações sobre o **Jinja2** e o **Bootstrap.**

Terminando este capítulo você terá duas páginas (home e dashboard) operando sobre uma tabela Usuario no banco; sendo assim, estaremos cobrindo toda a parte da definição de *models*, *forms*, *views*, *Blueprints* e criação de páginas HTML.

2.1 Estrutura de diretórios

A construção de um projeto complexo em Flask exige uma determinada organização de pastas e módulos para que o código do programa não fique embaralhado e confuso em momentos de desenvolvimento ou manutenção da aplicação. O padrão de organização

utilizado na maioria dos projetos em Flask, inclusive os do BioBD, respeita a seguinte estruturação:

```
├─── projeto
       │
       ├─── app
       │      │
       │      ├─── blueprint-exemplo
       │      │          ├─── __init__.py
       │      │          ├─── forms.py
       │      │          └─── views.py
       │      │
       │      ├─── static
       │      │          ├─── css
       │      │          │      └─── style.css
       │      │          ├─── javascript
       │      │          └─── images
       │      │
       │      ├─── templates
       │      │          └─── blueprint-exemplo
       │      │                    └─── exemplo.html
       │      │
       │      ├─── __init__.py
       │      └─── models.py
       │
       ├─── instance
       │      └─── config.py
       │
       ├─── config.py
       ├─── requirements.txt
       └─── run.py
```

Apesar de complexa, esta estrutura permite organizar os módulos do arquivo de maneira lógica. Obviamente mais arquivos e pastas podem (e serão) adicionados ao diretório do projeto, mas sempre respeitando a lógica da estrutura, ou seja, arquivos que **não** sejam pertinentes ao **código de controle da aplicação** serão inseridos na pasta do **projeto**. Qualquer outro tipo de pasta ou arquivo relacionado ao projeto especificamente (ou seja, que **não seja uma pasta de logs, arquivo de configuração do projeto, arquivo para executar o projeto, que funcionam de maneira idêntica independentemente do projeto**) deverá ficar na pasta **app.**

A tabela a seguir explica o conteúdo de cada arquivo/pasta presente na estrutura.

run.py	Arquivo que, após carregar o objeto app do projeto (trazendo consigo todas as configurações da aplicação) executa a aplicação em um servidor de desenvolvimento (ou seja local). Serve apenas para desenvolvimento.
requirements.txt	Arquivo que contém todos os pacotes do python necessários para que a aplicação funcione. Apesar de não utilizado por nós é possível ter arquivos separados para produção e desenvolvimento.
config.py	Arquivo que contém todas as variáveis de configuração da aplicação.
/instance/config.py	Arquivo especial que contém variáveis de configuração que não devem ser incluídas no controle de versionamento. Portanto, qualquer informação que não deve ser passada entre versões (como senhas de API, URI do banco ou configurações pessoais de desenvolvimento) devem ser aqui inseridas. Vale ressaltar que este arquivo é lido após o arquivo de configuração padrão, ou seja, é possível sobrescrever variáveis do arquivo público de configuração caso seja necessário.
/app	Pasta que contém o material (código Python, Javascript, HTML, CSS, imagens...) especialmente relacionado à aplicação.
/app/__init__.py	Arquivo que inicializa um módulo (usado para aplicação geral e as blueprints que serão explicadas mais tarde).
/app/views.py	Arquivo onde são definidas as rotas (URLs) do projeto e o que acontece (em termos de código) quando elas são acessadas pelo usuário. Pode ser dividido em vários módulos quando existirem muitos assuntos (no nosso caso, a divisão será via Blueprints, que será explicado posteriormente)

/app/models.py	Arquivo que contém as declarações das tabelas do banco de dados em forma de classes. Isto é importante para que as *queries* do projeto sejam feitas por meio do ORM do SQLAlchemy, que é uma dependência do projeto que auxiliará na conexão entre código e banco.
/app/static/	Pasta que contém todo material público da aplicação, como código Javascript, CSS, fotos. Todos os arquivos da pasta são acessíveis pela rota "domínio.com/static".
/app/templates/	Pasta que contém todos os arquivos HTML (com códigos do Jinja2 e estilos do Bootstrap) de cada página do site, separados por Blueprint (caso existam).

Observações importantes sobre os arquivos:

1. **O arquivo onde a conexão entre banco e aplicação é definido fica em instance/config.py:** neste arquivo estão a **URI do banco**, junto da senha e usuário de acesso, além da variável de ambiente ***SECRET_KEY*****,** que guarda uma chave de criptografia caso o Flask precise criptografar algum tipo de informação.

2. **Uma Blueprint funciona como uma pasta "sub-app":** especificamente, uma Blueprint é um componente modular do projeto, logo com elas é possível separar as rotas correlacionadas em grupos de "assunto". Por exemplo, dentro da pasta **app,** você pode separar todas as rotas relacionadas à **homepage** (pasta home por exemplo) das rotas relacionadas à **administração** (pasta admin por exemplo) do *site*. Dessa forma cada *view* fica separada por assunto, ainda podendo compartilhar módulos entre si (como models.py) e separar outros tipos arquivos (como os presentes na pasta *static*, *templates*) por assunto/componente.

3. **Um arquivo forms.py de um módulo define todos os formulários da Blueprint em questão:** se um determinado componente do projeto exige a manipulação de formulários na página, os campos do form serão especificados neste arquivo especial de cada Blueprint com auxílio do pacote **Flask-WTF** que será explicado mais tarde neste tutorial

2.2 Conectando-se ao banco

Neste tutorial supõe-se que o banco já tenha sido criado é seja necessária apenas a conexão.

A conexão será feita a uma banco PostgreSQL por meio do SQLAlchemy (cuja referência está no bloco anterior). Resumidamente, o SQLAlchemy permite que utilizemos as tabelas do banco de uma maneira mais clara no código, sem uso de *queries* regulares de SQL. As queries são feitas por métodos definidos no pacote que será baixado e atuam

sobre as classes definidas no arquivo ***models.py***, que representam as tabelas e colunas. SQLAlchemy também possui inclusive uma proteção contra *SQLInjecion*. Para instalá-lo:

```
pip install flask-sqlalchemy
```

Para finalizar a conexão, o arquivo ***instance/config.py*** deve ser editado com o seguinte:

```
# instance/config.py

# neste arquivo ficarão configurações privadas relacionadas à aplicação,
como senhas e endereços secretos


SECRET_KEY = "senha_dificil"
SQLALCHEMY_DATABASE_URI = "postgresql://user:password@host:port/db_path"
```

Onde **SECRET_KEY** representa uma chave criptográfica para a aplicação (ou seja, **deve ser bem escondida e com um padrão difícil de se adivinhar**, como por exemplo 'Sm9obiBTY2hyb20ga2lja3MgYXNz'); e **SQLALCHEMY_DATABASE_URI** guarda a URI para a base de dados do Postgres (com respectivos usuário e senha de acesso ao banco, além do nome do host e porta, com o caminho especificado para a base desejada).

Com esse arquivo, caso a conexão VPN esteja estabelecida, você está possibilitado a conectar-se ao banco.

2.3 Arquivo de configuração

Como boa prática, as configurações **públicas** do projeto ficarão em um único arquivo dividido por classes de uso, como abaixo:

```python
# config.py

class Config(object):
    """
    Configurações gerais
    """
    # Aqui ficam configurações comuns a todos os tipos de execução


class DevelopmentConfig(Config):
    """
    Configurações de desenvolvimento
    """
    # Aqui ficam configurações comuns a todos os tipos de execução

    DEBUG = True
    SQLALCHEMY_ECHO = True


class ProductionConfig(Config):
    """
    Configurações de produção
    """
    # Aqui ficam configurações comuns a todos os tipos de execução

    DEBUG = False

app_config = {
    'development': DevelopmentConfig,
    'production': ProductionConfig
}
```

Dessa forma, variáveis de ambiente relacionadas a configuração podem ser definidas de acordo com o método de execução escolhido. Vale ressaltar que as classes não são padrão do Flask e podem ser criadas por você (por exemplo, classes para **testes** ou ***bugfixes***).

O dicionário abaixo das classes relaciona uma string a uma classe de configuração e basicamente serve para que a variável de ambiente guardando a configuração escolhida seja uma ponte entre a execução do comando no Prompt e as definições da configuração no código (mais a frente isso será mostrado).

2.4 Arquivo de inicialização

O arquivo de inicialização (***__init__.py***) contém a função que carrega a configuração escolhida no prompt, retornando o objeto **app**, que será utilizado pelo projeto. Além disso, o este arquivo contempla as variáveis e objetos globais do projeto, como o objeto **db** que corresponde ao objeto em que as operações do banco estão. Através deste objeto operações como *queries* no banco serão feitas. O código **mínimo** que deve estar no arquivo segue abaixo:

```
# app/__init__.py

# imports de pacotes
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

# imports locais
from config import app_config

# inicialização do objeto db
db = SQLAlchemy()


def create_app(config_name):
    app = Flask(__name__, instance_relative_config=True)
    app.config.from_object(app_config[config_name])
    app.config.from_pyfile('config.py')
    db.init_app(app)

    return app
```

2.5 Arquivo *run.py*

Este é o arquivo associado a variável de ambiente **FLASK_APP** que quando executado, inicializa todo o projeto. Para executar o projeto, é preciso que, além deste arquivo estar escrito conforme abaixo, o seguinte código seja executado no **Prompt** estejam definidas como:

```
SET FLASK_APP = run.py
SET FLASK_CONFIG = 'escolha'
flask run
```

Onde a string '**escolha**' deve ser substituída por um dos nomes definidos no dicionário com os nomes das configurações, que está no arquivo ***config.py*****.** Abaixo segue o arquivo ***run.py*****.**

```
 # run.py

import os

from app import create_app

config_name = os.getenv('FLASK_CONFIG')
app = create_app(config_name)


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

2.6 Definição de *models*

Antes de criar os *models* das tabelas do banco, será necessário instalar o pacote **flask-login** para lidar com as sessões dos usuários no CRUD. Para isso execute no Prompt:

```
pip install flask-login
```

Para utilizá-lo, será necessário também modificar o arquivo ***__init__.py*****,** de forma que o arquivo inicialize o objeto **LoginManager**, que contém os métodos necessários para este controle:

```
# app/__init__.py

# após os outros imports

# adicionar o import do constructor
from flask_login import LoginManager

# inicializar o objeto citado
login_manager = LoginManager()
```

```
def create_app(config_name):
    # mantém-se o resto do código

    login_manager.init_app(app)
    # mensagem que aparece caso o usuário não esteja logado em uma
página que necessita de login
    login_manager.login_message = "Você precisa estar logado para
 ver esta página"
     # página a qual o usuário é redirecionado caso acesse uma
página sem estar logado
    login_manager.login_view = "auth.login"

    return app
```

Agora, definindo o os *models* em ***models.py***:

```
# app/models.py

from flask_login import UserMixin
from werkzeug.security import generate_password_hash,
check_password_hash

from app import db, login_manager

class Usuario(UserMixin, db.Model):
    """
    Define uma tabela Usuario
    """

    # Definição do nome da tabela (Este eh o nome que ficará no
banco em caso de migration e representa o nome da tabela no banco)
    __tablename__ = 'usuario'

    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(60), index=True, unique=True)
    username = db.Column(db.String(60), index=True, unique=True)
    first_name = db.Column(db.String(60), index=True)
    last_name = db.Column(db.String(60), index=True)
    password_hash = db.Column(db.String(128))
    is_admin = db.Column(db.Boolean, default=False)
```

```
    @property
    def password(self):
        """
        Prevent pasword from being accessed
        """
        raise AttributeError('password is not a readable
attribute.')

    @password.setter
    def password(self, password):
        """
        Set password to a hashed password
        """
        self.password_hash = generate_password_hash(password)

    def verify_password(self, password):
        """
        Check if hashed password matches actual password
        """
        return check_password_hash(self.password_hash, password)

    def __repr__(self):
        return '<Employee: {}>'.format(self.username)


# Set up user_loader
@login_manager.user_loader
def load_user(user_id):
    return Employee.query.get(int(user_id))
```

Note que existem alguma funções especiais no código. Para a geração de senhas com *hash* e a verificação delas quando em tentativas de login, os métodos importados do **Werkzeug**, **generate_password_hash** e **check_password_hash** estão sendo utilizados. Além disso, na última linha é possível ver a definição da função que recupera um id de usuário, em que nela existe um exemplo de como as *queries* são realizada neste modelo **ORM** (que foi comentado em seções anteriores).

Estando tudo feito, o projeto já possui classes que representam as tabelas do banco e operações sobre elas já podem ser feitas ao longo dos códigos do projeto.

2.7 Criando *Blueprints*

Como explicado anteriormente, um *Blueprint* é um contexto da aplicação cercado em um determinado componente/funcionalidade específica (definido pelo programador) que

modulariza os arquivos ***views.py***, ***forms.py***, e ***__init__.py***, ou seja, separa as definições e rotas que estão nesses arquivos em assuntos específicos. No nosso caso, criaremos uma *Blueprint* para a página inicial (**Home**) e outra para gerenciar os processos relacionados a autorização (**Auth** - cadastro e login).

Como no exemplo na seção **2.1**, as *Blueprint*s devem ser divididas por pastas (a que exemplifica é a pasta **blueprint-exemplo**). Portanto, crie as pastas **Home** e **Auth** seguindo aquele exemplo e adicione os códigos:

- Aos arquivos **nome_blueprint/__init__.py**:

```
# app/nome_blueprint/__init__.py

# import do construtor
from flask import Blueprint

# definição do objeto
nome_blueprint = ('nome_blueprint', __name__)

# importa views do módulo
from . import views
```

- E na pasta principal do projeto, o mesmo arquivo **app/__init__.py:**

```
# app/__init__.py

# mantém código anterior


def create_app(config_name):
    # mantém código anterior

    # importa classes do arquivo models
    from app import models

    # definições das blueprints
    from .blueprint_exemplo import blueprint_exemplo as
blueprint_exemplo_blueprint
    app.register_blueprint(blueprint_exemplo_blueprint)

    return app
```

Dessa forma, o Blueprint é inicializado em seu módulo e registrado dentro do contexto da aplicação. Estes dois códigos devem ser copiados para cada Blueprint criada. Caso exista a necessidade de criar prefixos de URL especiais, utilize o argumento ***url_prefix=' '*** em **app.register_blueprint( )** para isso, definindo a string desejada para o prefixo.

Agora, definiremos um exemplo de **views.py** e **forms.py** para a *Blueprint* **Auth.** Perceba que o padrão será o mesmo para quaisquer outras *Blueprint*, salvo casos em que não existam formulários a serem controlados na página. Nesse caso, não há necessidade de criar esse arquivo.

De qualquer forma, um bom exemplo desses arquivos para **Auth** são:

- **forms.py**

```
# app/auth/forms.py

from flask_wtf import FlaskForm
from wtforms import PasswordField, StringField, SubmitField,
ValidationError
from wtforms.validators import DataRequired, Email, EqualTo

from ..models import Usuario


class RegistrationForm(FlaskForm):
    """
    Definição do form para cadastro de usuário
    """
    email = StringField('Email', validators=[DataRequired(),
Email()])
    username = StringField('Nome de usuário',
validators=[DataRequired()])
    first_name = StringField('Nome, validators=[DataRequired()])
    last_name = StringField('Sobrenome',
validators=[DataRequired()])
    password = PasswordField('Senha, validators=[DataRequired(),
EqualTo('confirm_password')])
    confirm_password = PasswordField('Confirmar senha')
    submit = SubmitField('Cadastrar')

    def validate_email(self, field):
        if Employee.query.filter_by(email=field.data).first():
            raise ValidationError('Email já em uso')
```

```
    def validate_username(self, field):
        if Employee.query.filter_by(username=field.data).first():
            raise ValidationError('Nome de usuário já em uso')


class LoginForm(FlaskForm):
    """
    Definição do form de login
    """
    email = StringField('Email', validators=[DataRequired(),
Email()])
    password = PasswordField('Senha', validators=[DataRequired()])
    submit = SubmitField('Entrar')
```

Perceba que cada campo do formulário é definido por meio de um construtor que vem do pacote **wtforms,** diferenciando os campos por tipos de dados de entrada (ou seja, **StringField()** espera dados do tipo **string, PasswordField()** espera um dado string também, no entanto trata o input para que ele seja automaticamente **censurado**, e **SubmitField()** automaticamente **gera um botão** de *submit* para o form, sendo o parâmetro informado o texto que aparece no botão, assim como nos outros construtores de campo).

Além disso, os construtores de campo também possuem um parâmetro **validators**, que recebem outros construtores para gerenciar as validações de campo, ou seja, obrigar, por exemplo, que algum campo esteja devidamente preenchido quando os dados forem enviados (que é o caso de **DataRequired( )**)

- **views.py**

```
# app/auth/views.py

from flask import flash, redirect, render_template, url_for
from flask_login import login_required, login_user, logout_user

from . import auth
from forms import LoginForm, RegistrationForm
from .. import db
from ..models import Usuario


@auth.route('/register', methods=['GET', 'POST'])
def register():
    """
    Gerencía requests na rota de URL /register
    Adiciona um usuário no banco pelo form de cadastro
```

```
    """
    form = RegistrationForm()
    if form.validate_on_submit():
        usuario = Usuario(email=form.email.data,
                          username=form.username.data,
                          first_name=form.first_name.data,
                          last_name=form.last_name.data,
                          password=form.password.data)

        # add employee to the database
        db.session.add(usuario)
        db.session.commit()
        flash('Seu cadastro foi efetuado com sucesso')

        # redirect to the login page
        return redirect(url_for('auth.login'))

    # load registration template
    return render_template('auth/register.html', form=form,
title='Register')


@auth.route('/login', methods=['GET', 'POST'])
def login():
    """
    Gerencía requests na rota de URL /login
    Loga um usuário pelo form de login
    """
    form = LoginForm()
    if form.validate_on_submit():

        # checa se o usuário existe e/ou sua senha informada
corresponde com a cadastrada
usuario = Usuario.query.filter_by(email=form.email.data).first()
        if usuario is not None and usuario.verify_password(
                form.password.data):
            # loga o usuário
            login_user(usuário)

            # redireciona o usuário para a página principal
            return redirect(url_for('home.dashboard'))
```

```
        # caso as credenciais sejam inválidas
        else:
            flash('Email ou senha inválidos')

    # carrega o html da página de login
    return render_template('auth/login.html', form=form,
title='Login')


@auth.route('/logout')
@login_required
def logout():
    """
    Gerencía requests na rota de URL /logout
    Desloga um usuário pelo link da rota
    """
    logout_user()
    flash('Você saiu com sucesso.')

    # redireciona para a página de login
    return redirect(url_for('auth.login'))
```

Como explicado anteriormente, o arquivo determina exatamente o que acontece (em termos de código) quando cada **URL** é acessada. Vale ressaltar que as funções são associadas a uma rota por meio dos **decorators @app.route(‘ ’)**. É possível associar quantos decorator desejar a uma função, como é feito na função logout com @login_required. (decorator é uma funcionalidade padrão do Python, portanto você pode criá-los também se desejar).

Observação:

2.8 Criando *Templates*

Os templates dos projetos são as estruturas HTML de cada página do site. Cada HTML pode (e deve) ser misturado com código **Jinja2**, que pode adicionar certa “dinâmica” às páginas, e será estilizado por **CSS** e **Bootstrap**. Antes, será necessário instalar o módulo **flask-bootstrap** que conta com alguns arquivos que ajudam na construção de páginas com Bootstrap, além de atualizar o arquivo **app/__init__.py**:

```
pip install flask-bootstrap
```

```
# app/__init__.py
```

```
# after existing third-party imports
from flask_bootstrap import Bootstrap

# existing code remains


def create_app(config_name):
    # existing code remains

    Bootstrap(app)

    from app import models

    # blueprint registration remains here

    return app
```

Com isto feito, podemos iniciar os códigos HTML das páginas de cadastro e login.

- **register.html:**

```
<!-- app/templates/auth/register.html -->

{% import "bootstrap/wtf.html" as wtf %}
{% extends "base.html" %}
{% block title %}Register{% endblock %}
{% block body %}
<div class="content-section">
  <div class="center">
    <h1>Register for an account</h1>
    <br/>
    {{ wtf.quick_form(form) }}
  </div>
</div>
{% endblock %}
```

- **login.html**

```
<!-- app/templates/auth/login.html -->
```

```
{% import "bootstrap/utils.html" as utils %}
{% import "bootstrap/wtf.html" as wtf %}
{% extends "base.html" %}
{% block title %}Login{% endblock %}
{% block body %}
<div class="content-section">
  <br/>
  {{ utils.flashed_messages() }}
  <br/>
  <div class="center">
    <h1>Login to your account</h1>
    <br/>
    {{ wtf.quick_form(form) }}
  </div>
</div>
{% endblock %}
```

Os códigos do Jinja2 que inserem elementos na página são envoltos de **{{ }}** enquanto os códigos de procedimentos (como *if statements*) são envoltos de **{% %}.**
Existem muitas funcionalidades que o site pode usar do Jinja, no entanto, o que é mais importante de ser explicado são os "procedimentos" **extends** e **block**. Extends permite fazer o semelhante a um **#include em C**, copiando um arquivo inteiro no HTML em que é executado; A diferença é que o código da página em questão é **"inserido" ao invés de copiado** no HTML que será estendido por meio do elemento **block**. Sendo assim, separa-se um espaço no HTML "pai" e o HTML "filho" é envolto sobre as tags **{% block body %}, montando assim um HTML completo que herda código HTML do informando no comando extends** (observe que ainda não montamos o código "base.html", mas falaremos dele mais tarde)**.** O funcionamento de outros código pode ser visto na documentação do Jinja2.

O HTML utilizado é da versão **5** e cada elemento, atributo, categoria de conteúdo pode ser facilmente consultado na documentação do MDN.

Um código HTML base (**base.html**), por convenção, deve ser algo que você deseja que apareça em todas as outras telas do site. É conveniente, portanto, definir elemento como **footer** e **navbar** neste arquivo. Observe este exemplo:

```
<!-- app/templates/base.html -->

<!DOCTYPE html>
<html lang="en">
<head>
    <title>{{ title }} | Project Dream Team</title>
    <link
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstra
```

```
p.min.css" rel="stylesheet">
    <link href="{{ url_for('static', filename='css/style.css') }}"
rel="stylesheet">
    <link rel="shortcut icon" href="{{ url_for('static',
filename='img/favicon.ico') }}">
</head>
<body>
    <nav class="navbar navbar-default navbar-fixed-top topnav"
role="navigation">
        <div class="container topnav">
          <div class="navbar-header">
              <button type="button" class="navbar-toggle"
data-toggle="collapse"
data-target="#bs-example-navbar-collapse-1">
                  <span class="sr-only">Toggle navigation</span>
                  <span class="icon-bar"></span>
                  <span class="icon-bar"></span>
                  <span class="icon-bar"></span>
              </button>
              <a class="navbar-brand topnav" href="{{
url_for('home.homepage') }}">Project Dream Team</a>
          </div>
          <div class="collapse navbar-collapse"
id="bs-example-navbar-collapse-1">
              <ul class="nav navbar-nav navbar-right">
                  <li><a href="{{ url_for('home.homepage')
}}">Home</a></li>
                  <li><a href="#">Register</a></li>
                  <li><a href="#">Login</a></li>
              </ul>
          </div>
        </div>
    </nav>
    <div class="wrapper">
      {% block body %}
      {% endblock %}
      <div class="push"></div>
    </div>
    <footer>
        <div class="container">
            <div class="row">
```

```
                <div class="col-lg-12">
                    <ul class="list-inline">
                        <li><a href="{{ url_for('home.homepage')
}}">Home</a></li>
                        <li class="footer-menu-divider">·</li>
                        <li><a href="#">Register</a></li>
                        <li class="footer-menu-divider">·</li>
                        <li><a href="#">Login</a></li>
                    </ul>
                    <p class="copyright text-muted
small">Copyright (c) 2016. All Rights Reserved</p>
                </div>
            </div>
        </div>
    </footer>
</body>
</html>
```

O elemento **footer** representa o rodapé do site, enquanto o item **navbar** contém os itens do navegador (*top* ou *side*) para "andar" pelos links do site.

Um item importante das páginas são os estilos de cada uma. A estilização pode ser feita tanto pelo Bootstrap (que é inserido na página por meio das classes de HTML caso você já tenha baixado o *script* do Bootstrap - **classes row, col-lg-12, etc**) quanto por uma folha de estilos .css exemplificada abaixo:

```
/* app/static/css/style.css */

body, html {
    width: 100%;
    height: 100%;
}

body, h1, h2, h3 {
    font-family: "Lato", "Helvetica Neue", Helvetica, Arial,
sans-serif;
    font-weight: 700;
}

a, .navbar-default .navbar-brand, .navbar-default .navbar-nav>li>a
{
  color: #aec251;
```

```
}

a:hover, .navbar-default .navbar-brand:hover, .navbar-default
.navbar-nav>li>a:hover {
  color: #687430;
}

footer {
    padding: 50px 0;
    background-color: #f8f8f8;
}

p.copyright {
    margin: 15px 0 0;
}

.alert-info {
    width: 50%;
    margin: auto;
    color: #687430;
    background-color: #e6ecca;
    border-color: #aec251;
}

.btn-default {
    border-color: #aec251;
    color: #aec251;
}

.btn-default:hover {
    background-color: #aec251;
}

.center {
    margin: auto;
    width: 50%;
    padding: 10px;
}

.content-section {
    padding: 50px 0;
```

```
    border-top: 1px solid #e7e7e7;
}

.footer, .push {
  clear: both;
  height: 4em;
}

.intro-divider {
    width: 400px;
    border-top: 1px solid #f8f8f8;
    border-bottom: 1px solid rgba(0,0,0,0.2);
}

.intro-header {
    padding-top: 50px;
    padding-bottom: 50px;
    text-align: center;
    color: #f8f8f8;
    background: url(../img/intro-bg.jpg) no-repeat center center;
    background-size: cover;
    height: 100%;
}

.intro-message {
    position: relative;
    padding-top: 20%;
    padding-bottom: 20%;
}

.intro-message > h1 {
    margin: 0;
    text-shadow: 2px 2px 3px rgba(0,0,0,0.6);
    font-size: 5em;
}

.intro-message > h3 {
    text-shadow: 2px 2px 3px rgba(0,0,0,0.6);
}

.lead {
```

```
    font-size: 18px;
    font-weight: 400;
}

.topnav {
    font-size: 14px;
}

.wrapper {
  min-height: 100%;
  height: auto !important;
  height: 100%;
  margin: 0 auto -4em;
}
```

Em caso de dúvidas, consulte a documentação do Bootstrap 3 e a documentação do CSS 3 para adicionar estilos diferentes em sua página ou encontrar a definição de alguma classe/opção desconhecida.