Trabalhando com dados externos e criando comandos para inserção de dados

DEVinHouse

Parcerias para desenvolver a sua carreira

SENAI

<LAB365>

AGENDA

- Instalação das dependências
- Atualizando modelo de dados
- Trabalhando com dados externos
- Criando com comandos próprios

Instalação das dependências

- **poetry add flasgger requests bcrypt**

Atualizando modelo de dados

Para começarmos, iremos atualizar o modelo de dados que fizemos, para popularmos dinamicamente o banco de dados.

Atualizando modelo de dados - COUNTRY

```
#Dentro da classe Country, iremos adicionar esse bloco
  @classmethod
  def seed(cls, name, language):
    country = Country(
      name = name,
      language = language
    )
    country.save()


  def save(self):
    DB.session.add(self)
    DB.session.commit()
```

Atualizando modelo de dados - STATE

```
#Dentro da classe State, iremos adicionar esse bloco
  @classmethod
  def seed(cls, country_id, name, initials):
    state = State(
      country_id = country_id,
      name = name,
      initials = initials
    )
    state.save()


  def save(self):
    DB.session.add(self)
    DB.session.commit()
```

Atualizando modelo de dados - CITY

```
#Dentro da classe City, iremos adicionar esse bloco
@classmethod
  def seed(cls, state_id, name):
    city = City(
      state_id = state_id,
      name = name
    )
    city.save()
  def save(self):
    DB.session.add(self)
    DB.session.commit()
```

Atualizando modelo de dados - USER

```
import bcrypt


#Dentro da classe User, iremos adicionar esse bloco
@classmethod
  def seed(cls, state_id, name):
    city = City(
      state_id = state_id,
      name = name
    )
    city.save()
  def save(self):
    DB.session.add(self)
    DB.session.commit()
```

Atualizando modelo de dados - USER

```
import bcrypt #Adicionar nas importações

#Dentro da classe User, iremos adicionar esse bloco

  @classmethod

  def seed(cls, city_id, name, age, email, password):

    user = User(

      city_id = city_id,

      name = name,

      age = age,

      email = email,

      password = cls.encrypt_password(password.encode("utf-8")),

    )

    user.save()
```

Atualizando modelo de dados - USER

```
#Dentro da classe User, iremos adicionar esse bloco
  @staticmethod
  def encrypt_password(password):
      return bcrypt.hashpw(password, bcrypt.gensalt()).decode('utf-8')


  def save(self):
     DB.session.add(self)
     DB.session.commit()
```

Atualizando modelo de dados - TECHNOLOGY

```
#Dentro da classe Technology, iremos adicionar esse bloco
  @classmethod
  def seed(cls, name):
     tech = Technology(
        name = name
     )
     tech.save()


  def save(self):
     DB.session.add(self)
     DB.session.commit()
```

Atualizando modelo de dados - DEVELOPER

Nesta etapa, iremos excluir o arquivo developer_technology.py, pois iremos adicionar as informações dessa tabela satélite (join table) dentro da tabela de developer.

```
developer_technologies = DB.Table('developer_technologies',
                    DB.Column('developer_id', DB.Integer,
DB.ForeignKey('developers.id')),
                    DB.Column('technology_id', DB.Integer,
DB.ForeignKey('technologies.id'))
                    )
```

Atualizando modelo de dados - DEVELOPER

```
  #Atualizar os valores do modelo de dados de Developer e da def __init__
  id = DB.Column(DB.Integer, autoincrement=True, primary_key=True)
  months_experience = DB.Column(DB.Integer, nullable = True)
  accepted_remote_work = DB.Column(DB.Boolean, nullable = False, default = True)
  user_id = DB.Column(DB.Integer, DB.ForeignKey(User.id), nullable = False)
  technologies = DB.relationship('Technology', secondary=developer_technologies,
backref='developers')
  def __init__(self, months_experience, accepted_remote_work, user_id, technologies):
    self.months_experience = months_experience
    self.accepted_remote_work = accepted_remote_work
    self.user_id = user_id
    self.technologies = technologies
```

Atualizando modelo de dados - DEVELOPER

```
#Dentro da classe Developer, iremos adicionar esse bloco
  @classmethod
  def seed(cls, months_experience, accepted_remote_work, user_id, technologies):
    developer = Developer(
      months_experience = months_experience,
      accepted_remote_work = accepted_remote_work,
      user_id = user_id,
      technologies = technologies
    )
    developer.save()
  def save(self):
    DB.session.add(self)
    DB.session.commit()
```

Trabalhando com dados externos

Agora, iremos inserir uma grande massa de dados de alguma API externa para trabalhar em nosso banco de dados. Nesse caso, iremos criar na pasta src/app/db no arquivo __init__.py, uma função que será responsável por criar os dados no banco.

Trabalhando com dados externos

Agora, iremos inserir uma grande massa de dados de alguma API externa para trabalhar em nosso banco de dados. Nesse caso, iremos criar na pasta src/app/db no arquivo __init__.py, uma função que será responsável por criar os dados no banco.

Trabalhando com dados externos

```
from sqlalchemy.sql.expression import func
import requests
from src.app.models.city import City, cities_share_schema
from src.app.models.developer import Developer
from src.app.models.state import State, states_share_schema
from src.app.models.country import Country, country_share_schema
from src.app.models.user import User, users_share_schema
from src.app.models.technology import Technology
```

Trabalhando com dados externos

```
def populate_db():
  country = Country.query.first() # Query para verificar se existe dados salvos
  if country != None:
    print('Já existe dados populados.')
    return
  brasil_code = 76
  #Requisições para pegar dados de país, estado e cidade respectivamente
  countries_data =
requests.get(f"https://servicodados.ibge.gov.br/api/v1/localidades/paises/{brasil_code}")
  states_data = requests.get("https://servicodados.ibge.gov.br/api/v1/localidades/estados")
  cities_data = requests.get("https://servicodados.ibge.gov.br/api/v1/localidades/municipios")
```

Trabalhando com dados externos

```
  country_name = countries_data.json()[0]['nome']
  Country.seed(country_name, 'Português') # Seed utilizada para salvar dados
  country_dict = country_share_schema.dump(country) # Método para serializar o dado
  for stateObject in states_data.json(): # For para criar dados em massa dos estados
    State.seed(
      country_dict['id'],
      stateObject['nome'],
      stateObject['sigla']
    )
```

Trabalhando com dados externos

```
  state = State.query.order_by(State.name.asc()).all() # Query para ordenar de forma ascendente
  state_dict = states_share_schema.dump(state)
  for city_object in cities_data.json(): # for para salvar dados da cidade
    state_id = 0
    for state_object in state_dict:# for para identificar em qual estado a cidade pertence
      if state_object['initials'] == city_object['microrregiao']['mesorregiao']['UF']['sigla']:
        state_id = state_object['id']
    City.seed(
      state_id,
      city_object['nome']
    )
```

Trabalhando com dados externos

```
  cities = City.query.order_by(City.name.asc()).all()# Query para ordenar de forma ascendente
  cities_dict = cities_share_schema.dump(cities)
  users = requests.get('https://randomuser.me/api?nat=br&results=100')
  techs =
requests.get('https://lit-citadel-12163.herokuapp.com/technologies/get_all_technologies')
  for tech_object in techs.json(): # for para salvar a lista de tecnologias
    Technology.seed(
      tech_object['name']
    )
```

Trabalhando com dados externos

```
  for user in users.json()['results']: # for criado para salvar lista de usuários
    city_id = 2 #inicialmente setado com 2, pois nem sempre é encontrado um valor definido
    for city_object in cities_dict: # for criado identificar a cidade do usuário
      if user['location']['city'] == city_object['name']:
        city_id = city_object['id']
    User.seed(
      city_id,
      user['name']['first'] + ' ' + user['name']['last'],
      user['registered']['age'],
      user['email'],
      user['login']['password']
    )
```

Trabalhando com dados externos

```
  users = User.query.order_by(User.name.asc()).all()
  users_dict = users_share_schema.dump(users)
  for index, user_object in enumerate(users_dict):
    if index % 2 == 0: # if para verificar se o index é par
      techs = Technology.query.order_by(func.random()).limit(10).all() # query que retorna
aleatóriamente 10 tecnologias
      Developer.seed(
        None,
        index % 2 == 0,
        user_object['id'],
        techs
      )
  print("Dados inseridos com sucesso.")
  return
```

Criando com comandos próprios

Para executarmos algumas funcionalidades mais rapidamente, iremos utilizar um comando próprio para criar o banco de dados baseado na função que fizemos abaixo e um para apagar as tabelas quando necessário. Para começar, iremos importar no arquivo app.py:

```
from flask.cli import with_appcontext
import click
from src.app.db import populate_db
from src.app import DB
```

Criando com comandos próprios

```
#Nessa parte, iremos colocar esse bloco abaixo da instância da routes


@click.command(name='populate_db')#Comando para popular o banco de dados
@with_appcontext #Verificação no flask, que identifica o comando que executará a função abaixo.
def call_command():
  populate_db()


@click.command(name='delete_tables'))#Comando para deletar as tabelas do banco de dados
@with_appcontext #Verificação no flask, que identifica o comando que executará a função abaixo.
def delete_tables():
  DB.drop_all()
app.cli.add_command(call_command) #Adicionar na instanciação do Flask para encontrar o comando.
app.cli.add_command(delete_tables)#Adicionar na instanciação do Flask para encontrar o comando.
```

DEVinHouse

Parcerias para desenvolver a sua carreira

OBRIGADO!

SENAI

<LAB365>