Wesley Ângelo Ferreira / wesleyangeloferreira@gmail.com

Python Fundamentals

Python com PostgreSQL

Anotações

Python com PostgreSQL

Objetivos

- ✔ Conhecer o módulo para conexão.
- ✔ Fazer inserção, alteração, exclusão e consulta de dados.

Python com PostgreSQL

Nesta aula você aprenderá sobre o módulo de conexão com o PostgreSQL, como trabalhar, inserir dados e realizar consultas.

Anotações

Instalação do módulo

Instale o módulo que dá suporte ao PostgreSQL.

```
sudo apt-get install python3-psycopg2
```

- ✔ Este módulo em específico, deve ser instalado pelo apt-get.
- ✔ Postgres precisa de outras bibliotecas do sistema operacional que o pip não consegue resolver, já que foi criado para resolver dependências do python e não de Linux.

Anotações

Criando conexão

```
import psycopg2

con = psycopg2.connect("host=IP dbname=BANCO
                        user=USUARIO password=SENHA")
```

- ✔ Primeira, importa o módulo do postgres.
- ✔ Em seguida, abrimos uma conexão com o banco.
- ✔ Será utilizada para executar comandos dentro do banco.

Anotações

Executar operações

```
cur = con.cursor()
```

- ✔ A variável cur, recebe o cursor do banco de dados a partir da conexão.
- ✔ Este cursor possibilita efetuar as operações CRUD (Create, Retrieve, Update, Delete).

Anotações

Sintaxe

```
cur.execute(“insert into
TABELA(COLUNA1,COLUNA2)
            values ('VALOR1','VALOR2')”)
con.commit()
```

- ✔ Nesta código a variável cur recebeu o cursor para navegar no banco, executamos o comando insert.
- ✔ Logo abaixo, a variável com é empregada para fazer o commit.
- ✔ Enquanto não for efetuado o commit, não será gravado nenhum dado no banco.

Segue um exemplo de inserção no banco de dados:

```
#!/usr/bin/python3

import psycopg2

try:
    con = psycopg2.connect("host=127.0.0.1
                            dbname=projeto
                            user=admin
                            password=4linux")
    cur = con.cursor()
    cur.execute("insert into scripts(nome,conteudo) values('devops',
                                                'projeto de python')")
    con.commit()
    print("Registro criado com sucesso")
except Exception as e:
    print("Erro: {}".format(e))
    print("Fazendo rollback")
    con.rollback()

finally:
    print("Finalizando conexao com o banco de dados")
    cur.close()
    con.close()
```


O registro criado utilizando o DML Insert, ficará gravado na memória até que seja efetuado o

Sintaxe


```
#atualizar um registro
cur.execute("update TABELA set COLUNA=VALOR
where COLUNA=VALOR")
#deletar um registro
cur.execute("delete from TABELA where
COLUNA=VALOR")
```

- ✔ execute é utilizado para atualizar e deletar registros.
- ✔ Depende de commit para efetuar a modificação no banco.

Segue exemplo de update e delete no banco de dados:

```
#!/usr/bin/python3

import psycopg2

try:
    con = psycopg2.connect("host=127.0.0.1 dbname=projeto
                           user=admin password=4linux")
    cur = con.cursor()
    cur.execute("update scripts set nome='4linux' where id=1")
    print("Registro atualizado com sucesso")
    cur.execute("delete from scripts where id=2")
    print("Registro removido com sucesso")
    con.commit()
except Exception as e:
    print("Erro: {}".format(e))
    print("Fazendo rollback")
    con.rollback()

finally:
    print("Finalizando conexao com o banco de dados")
    cur.close()
    con.close()
```

Sintaxe

```
#voltar o último comando
cur.rollback()
#fecha a conexão
cur.close()
con.close()
```

- ✔ **cur.rollback()** volta a última transação caso ocorra algum erro.
- ✔ **cur.close()** e **con.close()** finalizam a conexão com o banco.

Segue exemplo forçando um rollback quando digitamos um update errado:

```
#!/usr/bin/python3

import psycopg2

try:
    con = psycopg2.connect("host=127.0.0.1
                            dbname=projeto
                            user=admin
                            password=4linux")
    cur = con.cursor()
    cur.execute("insert into projeto(nome,conteudo) values('devops',
                                                'projeto de python')")
    con.commit()
    print("Registro criado com sucesso")
except Exception as e:
    print("Erro: {}".format(e))
    print("Fazendo rollback")
    con.rollback()

finally:
    print("Finalizando conexao com o banco de dados")
    cur.close()
    con.close()
```

Sintaxe


```
#fazer a busca
cur.execute("select * from TABELA")
#retornar o primeiro registo
cur.fetchone()
#retornar todos os registros
cur.fetchall()
```

- ✔ **fetchone():** realiza busca, retorna só o primeiro registro do banco.
- ✔ **fetchall():** realiza busca retornando todos os registros do banco.

Abaixo temos um exemplo buscando o primeiro registro de uma tabela, na sequência todos os registros:

```
#!/usr/bin/python3

import psycopg2

try:
    con = psycopg2.connect("host=127.0.0.1 dbname=projeto
                           user=admin password=4linux")
    cur = con.cursor()
    cur.execute("select * from scripts")
    print("O primeiro registro é: ",cur.fetchone())
    print("Todos os registros: ",cur.fetchall())
except Exception as e:
    print("Erro: {}".format(e))
    print("Fazendo rollback")

finally:
    print("Finalizando conexao com o banco de dados")
    cur.close()
    con.close()
```

Note que o commit e o rollback foram removidos deste exemplo. No caso de querys, não são feitas alterações na base de dados, então eles não são necessários.

Python Fundamentals

Python com MySQL

Anotações

Python com MySQL

Objetivos da Aula

- ✔ Conhecer o módulo para conexão.
- ✔ Inserção, atualização, exclusão e consulta no banco.

Python com MySQL

Nesta aula apresentaremos o módulo de conexão com o banco de dados MySQL, mostrando como criar tabelas, inserir dados e realizar consultas através do Python.

Anotações

Instalação do Módulo

Instale módulo que dá suporte ao MySQL.

```
sudo apt-get install python3-mysqldb
```

- ✔ Este módulo em específico, será instalado pelo apt-get.
- ✔ MySQL precisa de outras bibliotecas do sistema operacional Linux que o pip não consegue resolver.

Anotações

Criando conexão

Efetuando a conexão com um banco de dados MySQL:

```
import MySQLdb
                        con =
MySQLdb.connect(host="127.0.0.1",user="usuario",
                        passwd="senha",db="banco")
```

Python não possui função nativa para realizar conexão com o MySQL. Por isso, instalamos o módulo MySQLdb. Este módulo, fornece todos os comandos necessários para interagir com o banco de dados.

Em funções, explicamos o conceito de tratamento de exceções, essenciais quando trabalhamos com bancos de dados.

Podemos usar o bloco do try, para efetuar a conexão com o banco de dados, o bloco do except para exibir o erro, caso não efetue a conexão com o banco e o bloco de finally para finalizar a conexão, após efetuar as operações.

É uma boa prática finalizar as conexões do MySQL após utilizá-las. Por padrão, o mysql deixa uma
conexão aberta durante 8 horas, então, caso não sejam fechadas após o seu uso, caso seu script tenha muitas conexões de banco, pode ocasionar o erro: MySQL server has gone away, causando a indisponibilidade do banco de dados.

Executando comandos

No MySQL também é necessário cursor para interagir com o banco de dados.

```
cur = con.cursor()
```

✔ Utilizamos a variável cur que recebeu o cursor para navegar no banco de dados.

Anotações

Executando comandos

con.commit(): grava os dados no banco.

```
cur.execute("insert into TABELA(COLUNA1,COLUNA2)
                values ('VALOR1','VALOR2')")
con.commit()
```

Segue um exemplo de inserção no banco de dados:

```
#!/usr/bin/python3

import MySQLdb

try:
    con = MySQLdb.connect(host="127.0.0.1",db="projetos",
                          user="admin",passwd="4linux")
    cur = con.cursor()
    cur.execute("insert into projetos(nome,cliente)
                values('Python','Dexter Courier')")
    con.commit()
    print("Registro criado com sucesso")
except Exception as e:
    print("Erro: {}".format(e))
    con.rollback()

finally:
    print("Finalizando conexao com o banco de dados")
    cur.close()
    con.close()
```

O registro criado utilizando a DML Insert, ficará gravado na memória até que seja efetuado o commit. Se o python não encontrar este comando, não haverá a persistência de dados ao término da execução do script .

Executando comandos

Podemos fazer update e delete:

```
#atualizar um registro
cur.execute("update TABELA set COLUNA=VALOR where
COLUNA=VALOR")
#deletar um registro
cur.execute("delete from TABELA where COLUNA=VALOR")
```

Segue exemplo de update e delete no banco de dados:

```
#!/usr/bin/python3

import MySQLdb

try:
    con = MySQLdb.connect(host="127.0.0.1",db="projetos",
                          user="admin",passwd="4linux")
    cur = con.cursor()
    cur.execute("update projetos set nome='4linux' where id=1")
    print('Registro atualizado com sucesso!')
    cur.execute("delete from projetos where id=2")
    print('Registro deletado com sucesso')
    con.commit()
    print("Registro criado com sucesso")
except Exception as e:
    print ("Erro: {}".format(e))
    con.rollback()

finally:
    print ("Finalizando conexao com o banco de dados")
    cur.close()
    con.close()
```

Sintaxe

```
#voltar o último comando
cur.rollback()
#fecha a conexão
cur.close()
con.close()
```

- **cur.rollback()** volta a última transação caso ocorra algum erro.
- **cur.close()** e **con.close()** finalizam a conexão com o banco de dados.

Segue exemplo forçando um rollback quando digitamos um update errado:

```
#!/usr/bin/python3

import MySQLdb

try:
    con = MySQLdb.connect(host="127.0.0.1",db="projetos",
                          user="admin",passwd="4linux")
    cur = con.cursor()
    cur.execute("insert into proj(nome,cliente)
                 values('Python','Dexter Courier')")
    con.commit()
    print("Registro criado com sucesso")
except Exception as e:
    print("Erro: {}".format(e))
    con.rollback()

finally:
    print("Finalizando conexao com o banco de dados")
    cur.close()
    con.close()
```

Neste exemplo foi digitado errado o nome da tabela.

Sintaxe

```
#fazer a busca
cur.execute("select * from TABELA")
#retornar o primeiro registo
cur.fetchone()
#retornar todos os registros
cur.fetchall()
```

- **fetchone():** realiza busca, retorna só o primeiro registro do banco.
- **fetchall():** realiza busca retornando todos os registros do banco.

Segue exemplo buscando o primeiro registro de uma tabela e, na sequência todos os registros:

```
#!/usr/bin/python3

import MySQLdb

try:
    con = MySQLdb.connect(host="127.0.0.1",db="projetos",
                          user="admin",passwd="4linux")
    cur = con.cursor()
    cur.execute("select * from projetos")
    print("Primeiro registro da tabela: ",cur.fetchone())
    print("Todos os registros: ",cur.fetchall())
except Exception as e:
    print("Erro: {}".format(e))

finally:
    print("Finalizando conexao com o banco de dados")
    cur.close()
    con.close()
```

Note que o commit e o rollback foram removidos deste exemplo. No caso de querys não são feitas alterações na base de dados, então eles podem ser omitidos.

Python Fundamentals

Python com MongoDB

Anotações

Objetivos da Aula

- ✔ Conhecer o módulo de conexão com o MongoDB.
- ✔ Trabalhar com Python e MongoDB.

Python com MongoDB

Nesta aula você conhecerá o módulo para conexão e integração do Python com MongoDB, aprenderá a fazer consultas, inserções, alterações e exclusões de documentos e subdocumentos.

Anotações

Instalação

Para conectar o MongoDB com scripts em Python, é necessário instalar o módulo PyMongo.

```
pip install pymongo
```

✔ Poucas coisas irão mudar na sintaxe shell do MongoDB para a sintaxe do Python.

Anotações

Conexão

Crie um script, contendo instância destinada a objeto MongoClient para fazer a conexão com o banco de dados.

```
from pymongo import MongoClient
client = MongoClient('127.0.0.1')
db = client['dexterops']
```

Anotações

Trabalhar com comandos no MongoDB

Para inserir um novo documento, execute:

```
db.fila.insert({“_id”:1,
                “servico”:”Intranet”,
                “status”:0})
```

Para deletar todos os documentos de uma collections:

```
db.fila.remove()
```

Inserção de Dados

Podemos inserir os documentos e subdocumentos como demonstrado no script a seguir:

```
from pymongo import MongoClient
client = MongoClient('127.0.0.1')
db = client['dexterops']


def inserir_dados():
    try:
        db.fila.insert({"_id":1, "empresa":"4linux",
                        "cursos": [{"nome":"Python Fundamentals",
                                    "carga horaria":40},
                                   {"nome":"Linux Fundamentals",
                                    "carga horaria":40}]})
    except Exception as e:
        print("Erro: {}".format(e))

inserir_dados()
```

Trabalhar com comandos no MongoDB

Para realizar busca de dados utilize:

```
db.fila.find()
```

A busca nos retornará um objeto, para exibir informações, execute:

```
for r in db.fila.find():
    print("Serviço:{} ".format(r['servico']))
```

Busca de Dados

Para a busca de dados utilizando subdocumentos, faça com a seguir:

```
from pymongo import MongoClient
client = MongoClient('127.0.0.1')
db = client['dexterops']


def buscar_dados():

    for r in db.fila.find():
        print('Empresa: {}'.format(r['empresa'] ))
        for c in r['cursos']:
            print('==============')
            print('Nome: {} \n Carga Horária: {} \n'.format(
                            (c['nome'],c['carga horaria'])))

buscar_dados()
```

Trabalhar com comandos no MongoDB

Para adicionar um subdocumento:

```
db.fila.update({“_id”:1}, {$addToSet:
                    {“servidores”:
                        {“nome”:”dns”,
                        “endereco”:”192.168.0.40”}
                    }})
```

Adicionar subdocumentos

Para adicionar os instrutores disponíveis, veja o exemplo:

```
from pymongo import MongoClient
client = MongoClient('127.0.0.1')
db = client['dexterops']


def adicionar_sub():
     db.fila.update({"_id":1}, {"$addToSet":
                    {"instrutores":{'nome':'Mariana',
                     'email':'mariana.albano@4linux.com.br'}}})

adicionar_sub()
```

Trabalhar com comandos no MongoDB

Para update em subdocumento:

```
db.fila.update({“_id”:1,”servidores.nome”:”dns”},
                {“$set”{“servidores.$.endereco”:
                        “10.0.0.1”}}
```

Update em subdocumentos

Para update dentro do subdocumento instrutores, podemos fazer desta forma:

```
from pymongo import MongoClient
client = MongoClient('127.0.0.1')
db = client['dexterops']


def update_sub():
     db.fila.update({"_id":2, "instrutores.nome":"Mariana"},
                    {"$set":{"instrutores.$.nome":"Mari"}})

update_sub()
```

Trabalhar com comandos no MongoDB

Remover um subdocumento:

```
db.fila.update({“_id”:1,”servidores.nome”:”dns”},
                {“$pull”{“servidores.nome”:”dns”}}
```

Remover em subdocumentos

Para remover um objeto dentro do subdocumento instrutores, podemos fazer deste modo:

```
from pymongo import MongoClient
client = MongoClient('127.0.0.1')
db = client['dexterops']


def remove_sub():
    db.fila.update({"_id":2, "instrutores.nome":"Mari"},
                   {"$pull": {"instrutores":{"nome":"Mari"}}})

remove_sub()
```

Python Fundamentals

SQLAlchemy

Anotações

Objetivos da Aula

- ✔ Compreender o que é sqlalchemy.
- ✔ Por que usar.
- ✔ Conhecer tipos de dados.
- ✔ Executar Operações CRUD.

SQLAchemy

Nesta aula apresentaremos a definição de sqlalchemy, a razão da sua utilização, seus tipos de dados, como fazer operações crud

Anotações

O que é SQLAlchemy

- ✔ Trata-se de uma biblioteca, criada por Mike Bayer em 2005.
- ✔ Permite interagir com grande variedade de bancos de dados.
- ✔ Possibilita criar modelos de dados e consultas de maneira mais confortável com códigos Python.

Anotações

Por que usar SQLAlchemy

✔ Abstrair seu código de SQL, aproveitar declarações e tipos comuns de instruções SQL que sejam criadas de forma eficiente.
✔ Evitar problemas com ataques de SQLInject.
✔ SQLAlchemy consiste escrita diferente, constitui biblioteca extensível que trabalha com Oracle, MySQL, Postgres etc.
✔ Pode ser estendida facilmente para outros bancos relacionais, trabalha em dois modos diferentes (Core e ORM).
✔ SQLAlchemy Core: Uma maneira Pythonica de representar seus dados sem perder o sotaque do SQL.
✔ Focado diretamente no banco e seu esquema.

Anotações

Tipos

Metadados: facilitam a indexação e busca a tabela.

Table: tabela.

Column: coluna da tabela.

Index: Acelera uma busca em um field especifico.

Anotações

Tipos

SQLALchemy	Python	SQL
BigInteger	Int	BIGINT
Boolean	bool	BOOLEAN OR SMALLINT
Date	datetime.datetime	DATE(SQLite:STRING)
Date Time	datetime.datetime	DATETIME(SQLite:STRING)
Enum	str	ENUM OR VARCHAR
Float	Float or Decimal	FLOAT OR REAL
Integer	int	INTEGER
Interval	datetime.timedelta	INTERVAL OR DATE from epoch
Large Binary	byte	BLOB OR BYTEA
Numeric	decimal.Decimal	NUMERIC or DECIMAL
Unicode	unicode	UNICODE or VARCHAR
Text	str	CLOB or TEXT
Time	datetime.time	DATETIME

Anotações

Configurando ambiente

1 Instale a biblioteca SQLALchemy:

```
# pip3 install sqlalchemy
```

2 Instale SQLite, execute:

```
# apt-get update
# apt-get install sqlite3
# apt-get install libsqlite3-dev
# apt-get install sqlitebrowser
```

Anotações

Criando Tabela com SQLAlchemy

Na estrutura de pasta, crie um arquivo **core.py** com o código:

```
1  from sqlalchemy import (create_engine, MetaData, Column,
2                          Table, Integer,
3                          String, DateTime)
4  from datetime import datetime
5
6
7  engine = create_engine('sqlite:///teste.db', echo=True)
8  metadata = MetaData(bind=engine)
9
10 user_table = Table('usuarios',
11                     metadata,
12                         Column('id', Integer, primary_key=True),
13                         Column('nome', String(40), index=True),
14                         Column('idade', Integer, nullable=False),
15                         Column('senha', String),
16                         Column('criado_em', DateTime, default=datetime.now),
17                         Column('atualizado_em', DateTime,
18                                 default=datetime.now, onupdate=datetime.now )
19                     )
20
21 metadata.create_all(engine)
```

Criando tabelas com sqlalchemy

In [1]: do modulo sqlamchemy importamos create_engine, o método de conexão Metadata que define a base de dados, Column, o método que define colunas, Table, método que estabelece as tabelas, Integer, String e DateTime que são tipos de dados como apresentado anteriormente.

In [4]: importamos o datetime, módulo nativo do python para trabalhar com data e hora.
In [7]: definimos a variável de conexão que recebe os parâmetros de conexão com o banco e, o parâmetro echo, logging que visualiza os processos realizados no terminal, este parâmetro pode receber falso para ter uma saída mais limpa no terminal.
In [8]: estabelecemos a variável que cria a base de dados recebendo o parâmetro da variável de conexão.
In [10]: determinamos a variável de criação da tabela que usa os seguintes parâmetros, nome da tabela, variável base de dados, colunas, cada coluna recebe os seguintes parâmetros, nome da coluna, tipo de dado, especificação da coluna.
In [21] : usamos o método da base de dados para fazer commit da criação da tabela, passando o parâmetro de conexão.

Inserindo dados com SQLAlchemy

Crie um arquivo **core_insert.py** com o código:

```
1   from core import user_table, engine
2
3   con = engine.connect()
4   ins = user_table.insert()
5   # insert pessoa
6   new = ins.values(idade=24, nome='teste', senha='testando')
7   con.execute(new)
8   # insert pesoas
9   con.execute(user_table.insert(),[
10              {'nome':'marcio', 'idade':20, 'senha':'semsenha'},
11              {'nome':'gustavo', 'idade':18, 'senha':'abacaxi123'},
12              {'nome':'guilherme', 'idade':22, 'senha':'goiaba123 '}
13  ])
```

Inserindo dados na tabela com sqlalchemy

In [1]: do arquivo core, importamos as variáveis, tabela e conexão.
In [3]: definimos uma variável de conexão que recebe o método connect.
In [4]: estabelecemos uma variável tabela, que recebe o método de inserção.
In [6]: determinamos uma variável de inserção que recebe os valores a serem inseridos.
In [7]: utilizamos o método execute da conexão, com parâmetro, variável de inserção para fazer commit dos valores na tabela.
In [9]: usamos o método execute da conexão, com os seguintes parâmetros, variável tabela, com método insert, uma lista de dicionário , na qual cada dicionário é equivalente a um insert na tabela.

Anotações

Select com SQLAlchemy

Crie um arquivo **core_select.py** com o código :

```
1  from sqlalchemy import select
2  from core import user_table
3
4  selecione = select([user_table])
5  selecione_01 = select([user_table]).where(user_table.c.nome == 'teste')
6
7  print([x for x in selecione_01.execute()])
8  print([x for x in selecione.execute()])
```

Select de dados na tabela com sqlalchemy

In [1]: do modulo sqlachemy importamos select.
In [2]: do arquivo core, importamos a tabela.
In [4]: definimos a variável de select, que recebe uma lista com as tabelas em que serão realizados os selects.
In [5]: mesmo procedimento anterior, porém passando uma condição where, que recebe o parâmetro tabela.coluna.nomecoluna e a condição igual a ‘teste’.
In [7]: utilizamos o listcomprension para percorrer o cursor do objeto e imprimir.
In [8]: usamos o listcomprension para percorrer o cursor do objeto e imprimir.

Anotações

Update com SQLAlchemy

Crie um arquivo **core_update.py** :

```
1  from sqlalchemy import update
2  from core import user_table, engine
3
4  con = engine.connect()
5
6  atualizar = update(user_table).where(user_table.c.nome == 'teste')
7
8  atualizar = atualizar.values(nome='daniel')
9  result = con.execute(atualizar)
10 print(result.rowcount)
11
12 atualizar = update(user_table).where(user_table.c.nome == 'daniel')
13 atualizar = atualizar.values(idade=(user_table.c.idade - 1))
14 result = con.execute(atualizar)
15 print(result.rowcount)
```

Update de dados na tabela com sqlalchemy

In [1]: do modulo sqlachemy importamos update.
In [2]: do arquivo core, importamos a tabela e a conexão.
In [4]: definimos a variável conexão com o método connect.
In [6]: determinamos a variável de update com where, semelhante ao select, para definir o usuário que receberá a alteração.
In [8]: estabelecemos os valores da variável update que serão alterados.
In [9]: determinamos o commit na conexão com o update.
In [10]: verificamos quantas linhas sofreram alterações.
In [12]: realizamos o mesmo passo da linha [6].
In [13]: definimos os valores da variável update, passando uma operação no dado.
In [14]: realizamos o mesmo passo da linha [9].
In [15]: realizamos o mesmo passo da linha [15].

Anotações

Delete com SQLAlchemy

Crie um arquivo **core_delete.py** com o código :

```
1  from sqlalchemy import delete
2  from core import user_table, engine
3
4  con = engine.connect()
5  d = delete(user_table).where(user_table.c.nome == 'daniel')
6
7  result = con.execute(d)
8  print(result.rowcount)
```

Delete de dados na tabela com sqlalchemy

In [1]: do modulo sqlachemy importamos delete.
In [2]: do arquivo core, importamos a tabela e a conexão.
In [4]: definimos a variável conexão com o método connect.
In [5]: determinamos a variável de delete com where, semelhante ao select, para definir o usuário que será deletado.
In [7]: estabelecemos o commit na conexão com o delete.
In [8]: verificamos quantas linhas sofreram alterações.

Anotações

Tipos

SQLALchemy	Python	SQL
BigInteger	Int	BIGINT
Boolean	bool	BOOLEAN OR SMALLINT
Date	datetime.datetime	DATE(SQLite:STRING)
Date Time	datetime.datetime	DATETIME(SQLite:STRING)
Enum	str	ENUM OR VARCHAR
Float	Float or Decimal	FLOAT OR REAL
Integer	int	INTEGER
Interval	datetime.timedelta	INTERVAL OR DATE from epoch
Large Binary	byte	BLOB OR BYTEA
Numeric	decimal.Decimal	NUMERIC or DECIMAL
Unicode	unicode	UNICODE or VARCHAR
Text	str	CLOB or TEXT
Time	datetime.time	DATETIME

Anotações: