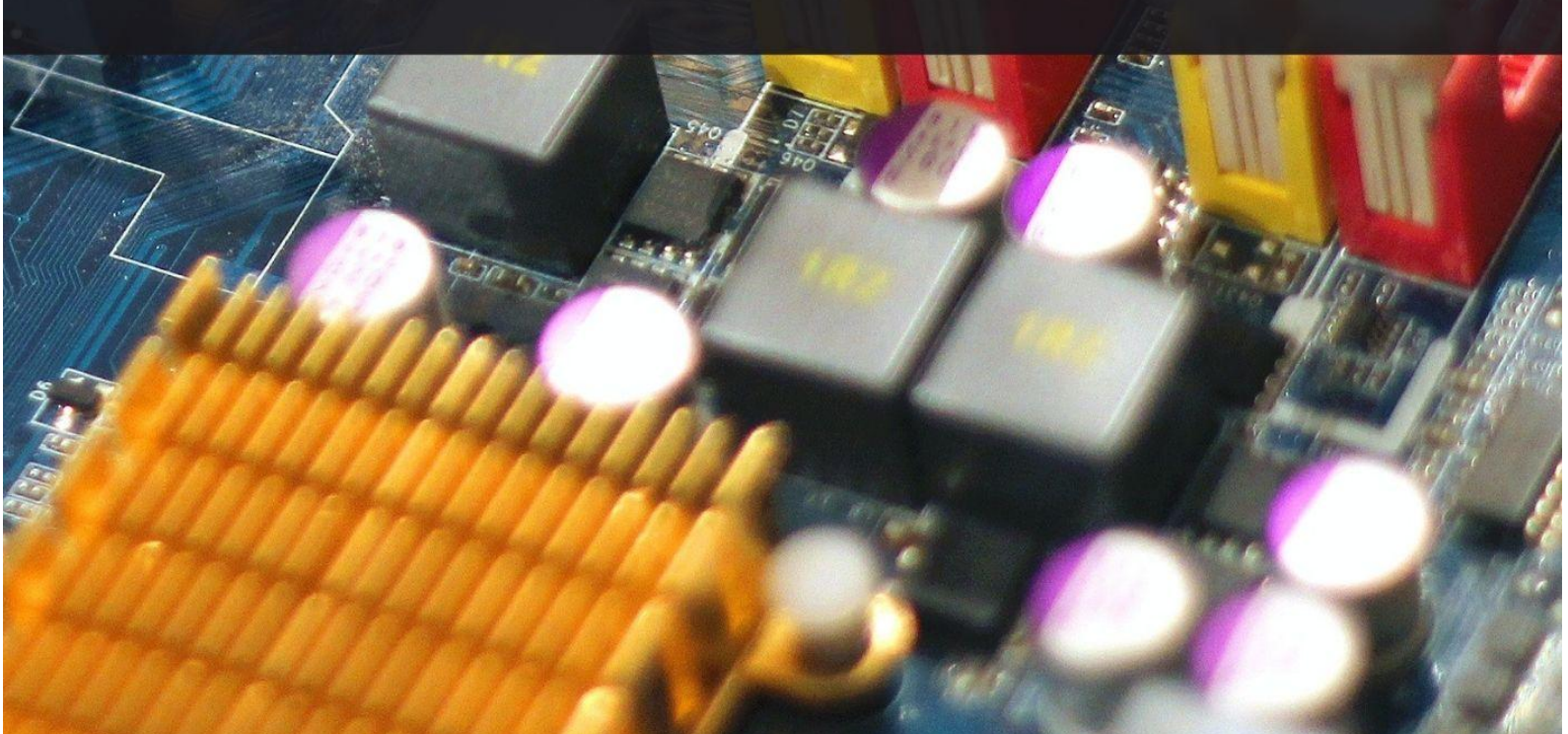


DESENVOLVIMENTO DE APIs E MICROSSERVIÇOS



3

Aprofundamento de SQL

Lucas Mendes Marques Gonçalves

Resumo

Você consegue criar bancos de dados consistentes e seguros?

Como maximizar a performance na carga dos dados?

Esses são os objetivos dessa aula, em que debatemos algumas das decisões tomadas quando se cria bancos de dados e quando se os acessa.

3.1. Constraints (restrições) em bancos de dados

Na função abaixo, podemos observar a criação de duas tabelas no banco de dados. Nessas tabelas, impomos algumas constraints, para garantir a consistência dos dados.

Codificação 3.1. Criação de tabelas

```
def criar_tabelas():  
  
    with engine.connect() as con:  
  
        create_tabela_aluno = """  
  
        CREATE TABLE IF NOT EXISTS Aluno (  
  
            id INTEGER PRIMARY KEY,  
  
            nome TEXT NOT NULL,  
  
            email TEXT NOT NULL UNIQUE  
  
        )
```



```
"""

rs = con.execute(create_tabela_aluno)

create_tabela_livro = """

CREATE TABLE IF NOT EXISTS Livro (

    id_livro INTEGER PRIMARY KEY,

    id_aluno INTEGER,

    descricao TEXT NOT NULL,

    FOREIGN KEY(id_aluno) REFERENCES Aluno(id)

)

"""

rs = con.execute(create_tabela_livro)
```

Fonte: do autor, 2021

A ideia de usar constraints é evitar que um erro de programação cause a inserção de dados inválidos no nosso banco.

Por exemplo, a constraint UNIQUE impõe que não pode haver valores repetidos. Ou seja, o código `email TEXT NOT NULL UNIQUE`, que pode ser visto na tabela “aluno”, diz que não pode haver dois alunos com o mesmo email. Se o programador tentar usar o mesmo email duas vezes, receberá um erro do banco de dados.

Já a constraint NOT NULL especifica que nunca podemos salvar valores NULL naquela coluna. Na tabela livro, vemos `descricao TEXT NOT NULL`, o que força todos os registros de livros a terem descrição - violações novamente geraram erros. Na mesma tabela livros, note que `id_aluno` pode ser NULL. Está especificada como `id_aluno INTEGER`. Isso foi feito porque essa coluna identifica o aluno que está com o livro emprestado - e é perfeitamente possível que nenhum aluno esteja com o livro emprestado.

A constraint PRIMARY KEY (como em `id_livro INTEGER PRIMARY KEY`), significa que aquela coluna identifica a linha de forma única. Ou seja, basta saber o valor da coluna, e conseguimos achar uma única linha que tenha esse valor. Quando aplicamos PRIMARY KEY a uma coluna, estamos automaticamente dizendo que ela é UNIQUE e NOT NULL.

A constraint FOREIGN KEY serve para marcar um relacionamento. Ao escrever `FOREIGN KEY(id_aluno) REFERENCES Aluno(id)` estamos informando ao banco de dados que uma `id_aluno` tem que ser uma id válida, que aparece na tabela `aluno`. Apenas alunos existentes podem pegar livros.

3.2. Carregar dados para memória: como fazer para otimizar a performance?

Nos códigos abaixo, temos duas funções que fazem a mesma coisa: retornam uma lista de alunos.

Codificação 3.2. Lista de alunos

```
def todos_alunos():  
  
    with engine.connect() as con:  
  
        sql_consulta = text("SELECT * FROM aluno")  
  
        rs = con.execute(sql_consulta)  
  
        resultados = []  
  
        while True:  
  
            result = rs.fetchone()  
  
            if result == None:  
  
                break  
  
            d_result = dict(result)  
  
            resultados.append(d_result)  
  
        return resultados  
  
def todos_alunos_versao2():  
  
    with engine.connect() as con:  
  
        sql_consulta = text("SELECT * FROM aluno")  
  
        rs = con.execute(sql_consulta)
```

```
resultados_sujo = rs.fetchall()

resultados_limpo = []

for resultado in resultados_sujo:

    resultados_limpo.append(dict(resultado))

return resultados_limpo
```

Fonte: do autor, 2021

A principal diferença é que **todos_alunos** carrega uma entrada do resultado por vez, usando **fetchone**. Já **todos_alunos_versao2** carrega todas as entradas de uma vez, usando **fetchall**. Ou seja, **todos_alunos_versao2** pode facilmente esgotar a RAM de nosso servidor, se houver muitos resultados para carregar.

Já **fetchone** pode, dependendo da implementação do banco de dados, ser extremamente lenta. Isso porque pode fazer muitos acessos ao disco rígido, e cada acesso desses é muito custoso.

Um meio termo que pode resolver o problema é usar **fetchmany**, que pega vários resultados de uma vez, mas não todos. Veja o código a seguir, que pega 20 resultados por vez.

Codificação 3.3. Lista de alunos - 2

```
def todos_alunos_versao3():

    with engine.connect() as con:

        sql_consulta = text("SELECT * FROM aluno")

        rs = con.execute(sql_consulta)

        resultados_limpo = []

        while True:

            resultados_sujo = rs.fetchmany(20)

            if resultados_sujo == []:

                break

            for resultado in resultados_sujo:

                resultados_limpo.append(dict(resultado))

        return resultados_limpo
```

Fonte: do autor, 2021

Ele é um pouco mais complexo, mas resolve ambos os problemas.

3.3. Segurança contra SQL Injections usando prepared statements.

Para entender SQL Injections, vamos usar um exemplo. Veja a criação de um banco de dados abaixo.

Codificação 3.4. Criação de banco de dados

```
from sqlalchemy import create_engine
from sqlalchemy.sql import text

#quero usar o banco de dados nesse arquivo, usando o formato sqlite
engine = create_engine('sqlite:///autentica.db')

def criar_usuarios():
    with engine.connect() as con:
        create_tabela_senha = """
        CREATE TABLE IF NOT EXISTS users (
            login TEXT NOT NULL,
            senha TEXT NOT NULL
        )
        """
        rs = con.execute(create_tabela_senhas)

def criar_usuarios():
    with engine.connect() as con:
        add_usaria = "INSERT INTO users (login, senha) VALUES
('Minerva','123') "
        con.execute(add_usuaria)
        add_usuario = "INSERT INTO users (login, senha) VALUES
('José','Carabina') "
```

```
con.execute(add_usuario)

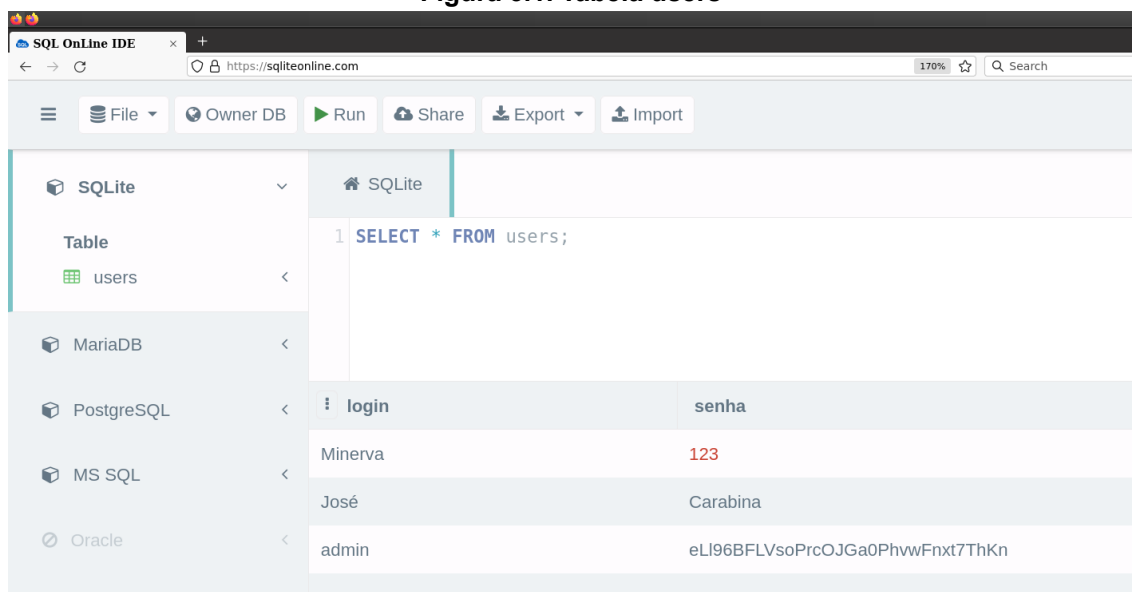
add_admin = "INSERT INTO users (login, senha) VALUES
('admin', 'eLl96BFLVsoPrcOJGa0PhvwFnxt7ThKn') "

con.execute(add_admin)
```

Fonte: do autor, 2021.

Nossa tabela de usuários fica da seguinte forma.

Figura 3.1. Tabela users



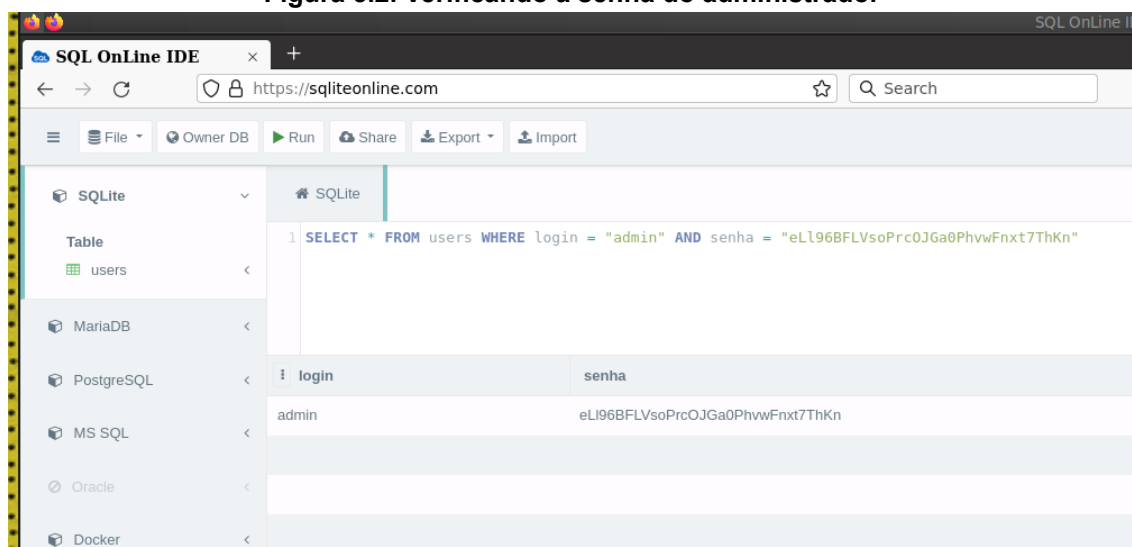
The screenshot shows the SQLite Online IDE interface. On the left, a sidebar lists databases: SQLite, MariaDB, PostgreSQL, MS SQL, and Oracle. The 'SQLite' database is selected, and the 'users' table is highlighted. The main area displays the SQL query: `SELECT * FROM users;`. Below the query, the results of the table are shown in a table format.

login	senha
Minerva	123
José	Carabina
admin	eLl96BFLVsoPrcOJGa0PhvwFnxt7ThKn

Fonte: sqliteonline.com, 2021

Esse banco de dados nos permite fazer o login de um usuário, verificando sua senha. Veja o select que nos permite fazer isso.

Figura 3.2. Verificando a senha do administrador



The screenshot shows the SQLite Online IDE interface. The 'users' table is selected. The main area displays the SQL query: `SELECT * FROM users WHERE login = "admin" AND senha = "eLl96BFLVsoPrcOJGa0PhvwFnxt7ThKn"`. Below the query, the results of the table are shown in a table format.

login	senha
admin	eLl96BFLVsoPrcOJGa0PhvwFnxt7ThKn

Fonte: sqliteonline.com, 2021

A ideia é que, se tivermos o nome de usuário e a senha corretos, a query retorna uma única linha. Se não tivermos, não retorna nenhuma linha, e podemos dizer que o usuário não foi autorizado.

Para fazer esse select no python, poderíamos usar a função a seguir (mas ela tem um problema importante!).

Codificação 3.5. Validando usuário

```
def validar_admin(senha):  
    with engine.connect() as con:  
        sql_admin = text (f"SELECT * FROM users WHERE login='admin'  
and senha='{senha}'")  
        print(sql_admin)  
        rs = con.execute(sql_admin)  
        result = rs.fetchone()  
        if result == None: #nao retornou nenhuma linha  
            return 'nao autorizado'  
        return 'autorizado'
```

Fonte: do autor, 2021.

A função funciona para alguns casos simples.

Codificação 3.6. Validando usuário - 2

```
>>> validar_admin("senhaerrada")  
  
SELECT * FROM users WHERE login='admin' and senha='senhaerrada'  
  
'nao autorizado'  
  
>>> validar_admin("eLl96BFLVsoPrcOJGa0PhvwFnxt7ThKn")  
  
SELECT      *      FROM      users      WHERE      login='admin'      and  
senha='eLl96BFLVsoPrcOJGa0PhvwFnxt7ThKn'  
  
'autorizado'
```

Fonte: do autor, 2021.

Note que além do resultado da validação, estamos imprimindo também a query executada no banco de dados.

Se rodarmos a função com um input malicioso, teremos um resultado surpreendente!

Codificação 3.7. Validando usuário - 3

```
>>> validar_admin("nao sei' OR 'a'='a")

SELECT * FROM users WHERE login='admin' and senha='nao sei' OR
'a'='a'

'autorizado'
```

Fonte: do autor, 2021

O que aconteceu ??

O erro fundamental está nessa linha do código.

Codificação 3.8. Validando usuário - 4

```
sql_admin = text (f"SELECT * FROM users WHERE login='admin' and
senha='{senha}'")
```

Fonte: do autor, 2021

Aqui, estamos dizendo, essencialmente, “coloque o conteúdo da string senha na string SQL para executar”.

Por isso, quando o usuário manda uma “senha” estranha como `"nao sei' OR 'a'='a'"`, isso passa a fazer parte do nosso código SQL, que será executado. Permitimos ao atacante escrever uma query em que OU o nome de usuário e senha estão corretos, OU `'a'='a'`. Mas, como `'a'='a'` é sempre verdade, independente da linha, nossa query retorna todas as linhas, e acabamos por considerar o usuário válido.

Sofremos um ataque do tipo “sql injection” ou “injeção de SQL”.

A solução, felizmente, é bem simples.

Codificação 3.9. Validando usuário - 5

```
def validar_seguro(senha):

    with engine.connect() as con:

        sql_admin = text ("SELECT * FROM users WHERE login='admin'
and senha= :senha")

        print(sql_admin)

        rs = con.execute(sql_admin , senha=senha)

        result = rs.fetchone()

        if result == None:

            return 'nao autorizado'

        return 'autorizado'
```

Fonte: do autor, 2021

Basta informar ao banco de dados o que é código SQL válido e o que é string fornecida pelo usuário. Se ele souber a diferença, vai evitar o ataque automaticamente. Essa técnica é chamada de “prepared statement”.

Em resumo:

- O sql injection é uma vulnerabilidade de segurança que ocorre quando usamos input do usuário para montar a string SQL a ser executada
- O prepared statement é a proteção contra sql injection. Criamos strings sql que usam uma sintaxe adequada para informar à nossa biblioteca de acesso ao banco de dados qual parte da string é codificada por nós e qual parte é input do usuário
- No nosso caso, a sintaxe para prepared statement é como no exemplo:

Codificação 3.10. Validando usuário - 6

```
sql_admin = text ("SELECT * FROM users WHERE login='admin' and  
senha= :senha")  
  
rs = con.execute(sql_admin , senha=senha)
```

Fonte: do autor, 2021

Referências

SQLALCHEMY. **A high level view and getting set up**. SQLAlchemy, 21 jul. 2021. Disponível em: <<https://docs.sqlalchemy.org/en/14/>>. Acesso em: 27 jul. 2021.