



AULA 05

FASTAPI

BANCO DE DADOS COM FASTAPI

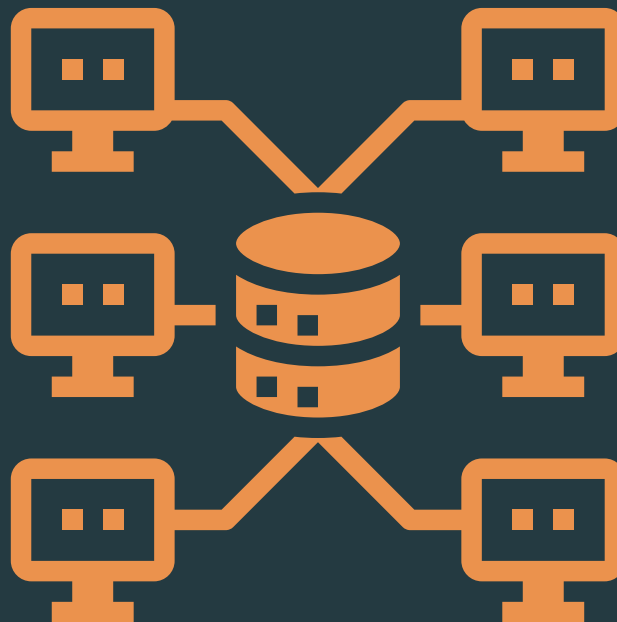
O QUE VEREMOS HOJE

- 01 BANCO DE DADOS
- 02 MYSQL
- 03 ORM E SQLMODEL
- 04 CONEXÃO DO FASTAPI COM MYSQL
- 05 CRUD USANDO O BANCO DE DADOS

BANCO DE DADOS

Um banco de dados é um sistema organizado de armazenamento e gerenciamento de dados, projetado para fornecer acesso eficiente e seguro a informações.

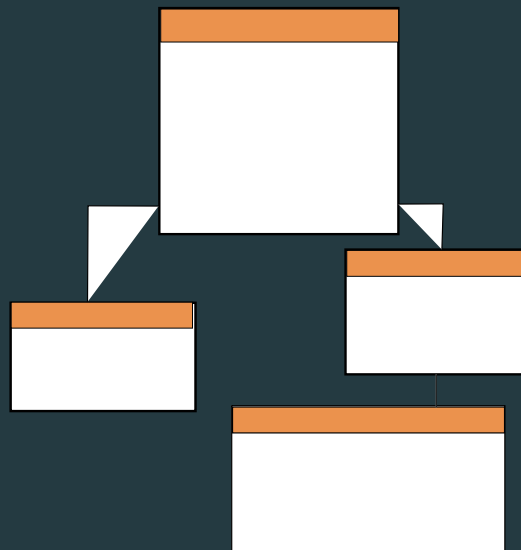
Ele permite que aplicações armazenem, recuperem, manipulem e gerenciem diversos tipos de dados de forma estruturada e padronizada.



BANCO DE DADOS RELACIONAL

Um banco de dados relacional é um tipo de sistema de gerenciamento de banco de dados (SGBD) que organiza os dados em **tabelas com linhas e colunas**. Cada tabela representa uma entidade, como clientes, produtos ou pedido, e as colunas representam os diferentes atributos dessa entidade.

Bancos de dados relacionais utilizam a linguagem SQL (Structured Query Language) para criar, modificar e consultar os dados de forma estruturada e padronizada.



MYSQL

O MySQL é um dos bancos de dados relacionais mais populares e amplamente usados no mundo. Ele é conhecido por sua rapidez, escalabilidade e eficiência.

Ele segue o modelo relacional, o que significa que os dados são organizados em tabelas compostas por linhas e colunas.




SQL (STRUCTURED QUERY LANGUAGE)

SQL é uma linguagem de programação específica para trabalhar com bancos de dados relacionais. Ela permite que você crie, manipule e gerencie dados de forma estruturada e eficiente.

Com o SQL, você pode realizar diversas operações, como selecionar, inserir, atualizar e excluir dados, além de criar tabelas, índices e visões no banco de dados.



COMANDOS SQL

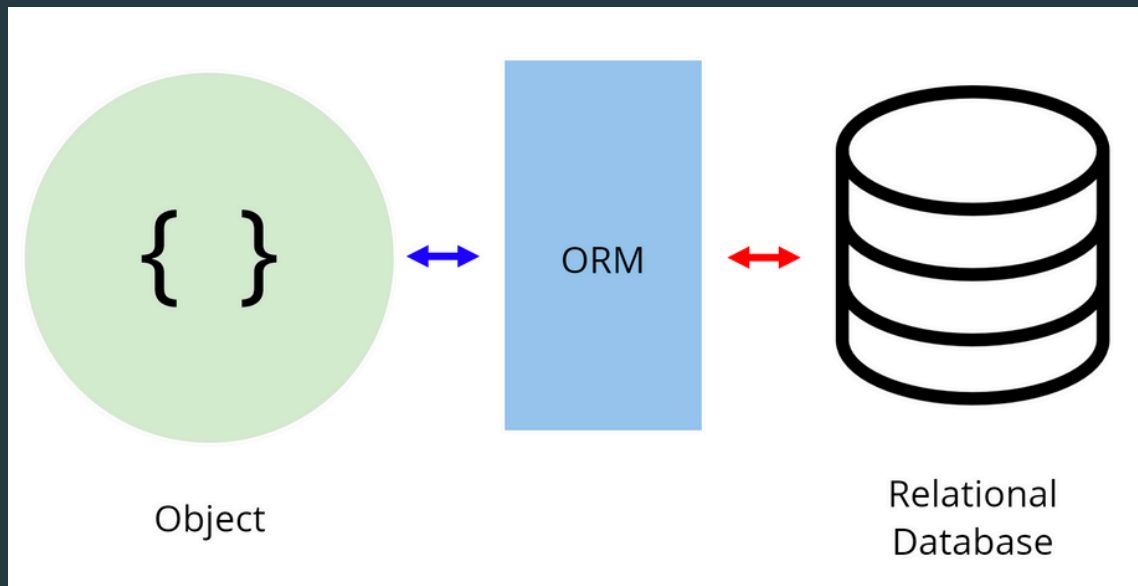


```
1  -- Cria uma nova tabela chamada usuarios --
2  CREATE TABLE usuarios (
3      id INT AUTO_INCREMENT PRIMARY KEY,
4      nome VARCHAR(100),
5      email VARCHAR(100)
6  );
7
8  -- Adiciona um novo registro na tabela --
9  INSERT INTO usuarios (nome, email)
10 VALUES ('João Silva', 'joao@email.com');
11
12 -- Seleciona todos os registros da tabela --
13 SELECT * FROM usuarios;
14
15 -- Atualiza o email do usuário com id 1 --
16 UPDATE usuarios
17 SET email = 'joaonovo@email.com'
18 WHERE id = 1;
19
20 -- Remove o usuário com id 1 --
21 DELETE FROM usuarios
22 WHERE id = 1;
```

ORM (OBJECT-RELATIONAL MAPPING)

ORM é uma técnica de mapear objetos de uma linguagem de programação orientada a objetos (como Python) diretamente para as tabelas de um banco de dados relacional (como MySQL, PostgreSQL, etc.).

Em vez de escrever consultas SQL diretamente, o ORM permite que você utilize classes Python para representar tabelas e colunas do banco de dados.




COMO IMPLEMENTAR O ORM

Para implementar o ORM vamos usar a biblioteca SQLAlchemy, essa biblioteca permite mapear os objetos Python para tabelas de banco de dados executando SQL por baixo dos panos.

Para utilizar o SQLAlchemy, precisamos instalar a biblioteca usando o comando:

pip install sqlalchemy

E para utilizar, precisamos importar a biblioteca.



```
1 from sqlalchemy import SQLAlchemy, Field
2 from typing import Optional
3
4 class User(SQLAlchemy, table=True):
5     id: int = Field(primary_key=True) # Autoincremento automático
6     name: str
7     email: str
8     password: str
9     age: Optional[int] = Field(default=None)
```

COMO IMPLEMENTAR O ORM

Além de criar o model, é importante garantir que a validação e formatação dos dados estejam conforme as regras definidas no model.

Como estamos trabalhando com um ORM, também precisamos garantir que os objetos do model sejam convertidos em dicionários que o Pydantic consiga interpretar e manipular.

```
1 from pydantic import BaseModel
2 from typing import Optional
3
4 # Leitura de usuários
5 class UserRead(BaseModel):
6     id: int
7     name: str
8     email: str
9     age: Optional[int] = None
10
11     class Config:
12         from_attributes = True # Configuração para aceitar objetos ORM e transformar dicionários
13
14 # Criação de usuários
15 class UserCreate(BaseModel):
16     name: str
17     email: str
18     password: str
19     age: Optional[int] = None
20
```

COMO CONECTAR O MYSQL NO FASTAPI

Uma vez definido os modelos para representar, vamos fazer a integração do MySQL com o FastAPI.

Para isso, temos que instalar o PyMySQL usando o comando:

pip install pymysql

Depois disso, podemos fazer a configuração do banco de dados usando a URL do banco que vai ser como essa:

mysql+pymysql://usuario:senha@localhost:3306/nome_do_banco



```
1 from sqlmodel import SQLModel, create_engine, Session
2
3 # Configurando a URL de conexão com o MySQL
4 DATABASE_URL = "mysql+pymysql://root:@localhost:3306/test"
5
6 # Criando o engine de conexão e o echo permite visualizar as queries SQL no terminal
7 engine = create_engine(DATABASE_URL, echo=True)
```

DEPENDÊNCIAS

No FastAPI, dependências são **funções ou classes que podem ser injetadas em rotas** para realizar uma tarefa comum ou fornecer algum recurso necessário para a rota funcionar.

As dependências garantem que certas operações, como abrir e fechar uma **sessão de banco de dados**, sejam feitas de forma automática e segura.



DEPENDÊNCIA DE SESSÃO

Para conectar o FastAPI ao banco de dados, precisamos criar uma dependência de sessão. A função **get_session()** é responsável por abrir uma sessão com o banco de dados quando uma rota precisar e fechá-la automaticamente após a execução da requisição.


- **with Session(engine) as session:** Abre uma sessão com o banco de dados e garante que ela será fechada automaticamente ao final do processo, mesmo se houver erros.
- **yield session:** Pausa a execução da função e retorna a sessão para a rota. Quando a rota finaliza, o FastAPI continua a função e fecha a sessão

```
1 from sqlmodel import SQLModel, create_engine, Session
2
3 # Configurando a URL de conexão com o MySQL
4 DATABASE_URL = "mysql+pymysql://root:@localhost:3306/test"
5
6 # Criando o engine de conexão e o echo permite visualizar as queries SQL no terminal
7 engine = create_engine(DATABASE_URL, echo=True)
8
9 # Função de dependência para criar uma sessão no banco de dados
10 def get_session():
11     with Session(engine) as session:
12         yield session
13
14 # Função para criar as tabelas no banco
15 def create_tables():
16     SQLModel.metadata.create_all(engine)
```

USANDO A DEPENDÊNCIA NAS ROTAS

Com o modelo (User) e os schemas de validação (UserCreate e UserRead) criados, podemos utilizá-los diretamente nas rotas da API.

Para garantir que cada rota possa interagir com o banco de dados, utilizamos a dependência `get_session`, que abre e fecha uma sessão de banco de dados automaticamente.



```
1 from fastapi import APIRouter, HTTPException, Depends
2 from sqlmodel import Session, select
3 from schemas.user import UserCreate, User
4 from db import get_session
5
6 router = APIRouter()
7
8 @router.get('/users/', response_model=list[User])
9 def read_users(session: Session = Depends(get_session)):
10     # Inicializa a sessão e executa a query para buscar todos os usuários
11     users = session.exec(select(User)).all()
12     return users
```

CRUD COM BANCO DE DADOS

Agora, ao utilizar o banco de dados, a operação **Create** exigirá a injeção de uma dependência que gerencia a sessão do banco de dados. Essa sessão será usada para registrar o novo usuário de forma segura e eficiente, garantindo que as conexões sejam abertas e fechadas corretamente.

```
1 from fastapi import APIRouter, HTTPException, Depends
2 from sqlmodel import Session, select
3 from schemas.user import UserCreate, UserRead
4 from models.user import User
5 from db import get_session
6
7 router = APIRouter()
8
9 @router.post('/users/', response_model=UserRead)
10 def create_user(user_create: UserCreate, session: Session = Depends(get_session)):
11     # Converte o UserCreate (Pydantic) para o User (SQLModel) para salvar no banco
12     user_db = User(**user_create.model_dump())
13     session.add(user_db) # Adiciona o novo usuário com a senha no banco de dados
14     session.commit() # Salva as alterações no banco
15     session.refresh(user_db) # Atualiza com o ID gerado
16     return {
17         'message': 'User created successfully',
18         'user': user_db
19     }
20
```

CRUD COM BANCO DE DADOS

O **Read** também vai exigir a injeção de dependência e também vai usar a sessão para fazer as opção com banco de dados.

```
1 from fastapi import APIRouter, HTTPException, Depends
2 from sqlmodel import Session, select
3 from schemas.user import UserCreate, UserRead
4 from models.user import User
5 from db import get_session
6
7 router = APIRouter()
8
9 @router.get('/users/', response_model=list[UserRead])
10 def read_users(session: Session = Depends(get_session)):
11     # Inicializa a sessão e executa a query para buscar todos os usuários
12     users = session.exec(select(User)).all()
13     return users
14
15 @router.get('/users/{user_id}', response_model=UserRead)
16 def read_user(user_id: int, session: Session = Depends(get_session)):
17     # Inicializa a sessão e executa a query para buscar um usuário pelo ID
18     user = session.get(User, user_id)
19     if not user:
20         raise HTTPException(status_code=404, detail='User not found')
21     return user
22
```


CRUD COM BANCO DE DADOS

O **Update** vai exigir o ID e os dados que serão inseridos no banco de dados. Pode ser usado o schema de criação ou pode ser criado um novo schema só para atualização.

```
1 from fastapi import APIRouter, HTTPException, Depends
2 from sqlmodel import Session, select
3 from schemas.user import UserCreate, UserRead
4 from models.user import User
5 from db import get_session
6
7 router = APIRouter()
8
9 @router.put('/users/{user_id}', response_model=UserRead)
10 def update_user(user_id: int, user_update: UserCreate, session: Session = Depends(get_session)):
11
12     user_db = session.get(User, user_id)
13     if not user_db:
14         raise HTTPException(status_code=404, detail="User not found")
15
16     user_db.name = user_update.name
17     user_db.email = user_update.email
18     user_db.password = user_update.password
19     user_db.age = user_update.age
20
21     session.commit() # Salvando as alterações no banco de dados
22     session.refresh(user_db) # Atualiza o objeto com os novos dados do banco de dados
23
24     return {
25         'message': 'User updated successfully',
26         'user': user_db
27     }
28
```

CRUD COM BANCO DE DADOS

A operação **Delete** também vai exigir o ID e a busca do registro no banco de dados e será usada a sessão pra excluir o registro.

```
1 from fastapi import APIRouter, HTTPException, Depends
2 from sqlmodel import Session, select
3 from schemas.user import UserCreate, UserRead
4 from models.user import User
5 from db import get_session
6
7 router = APIRouter()
8
9 @router.delete('/users/{user_id}', response_model=UserRead)
10 def delete_user(user_id: int, session: Session = Depends(get_session)):
11
12     user_db = session.get(User, user_id)
13     if not user_db:
14         raise HTTPException(status_code=404, detail="User not found")
15
16     # Remover o usuário da sessão e do banco de dados
17     session.delete(user_db)
18     session.commit() # Confirmar a remoção
19
20     return {'message': 'User deleted successfully'}
```

PROJETO DO MÓDULO

Sistema de Gerenciamento de Tarefas

Detalhes do projeto: https://docs.google.com/document/d/1rhx31q9s0lokyaAh6e_q4g3OqD3knCO57EmdUgk7vEE/edit?usp=sharing.

ATIVIDADE PRÁTICA

Crie um CRUD de Projetos usando o banco de dados onde cada projeto deve conter o ID, nome, descrição (opcional), e um campo de usuário associado (`user_id`) que também será opcional nesta fase inicial.

O CRUD deve conter os seguintes endpoints:

- Criação de novo projeto.
- Lista de todos os projetos.
- Busca de um projeto específico pelo ID.
- Atualização de projeto com base no ID.
- Exclusão de um projeto com base no ID.

Observações:

- O campo `user_id` deve ser do tipo inteiro e opcional.

Utilize a modularização para incluir as rotas, schemas e models.