

# Sistema de Restaurante Delivery — Pacote completo

Este documento contém todos os ficheiros essenciais para começar o projeto: esquema SQL, esqueleto de código do servidor e cliente (em Python), scripts de teste, `requirements.txt` e um `README` com instruções passo-a-passo para executar entre duas máquinas (servidor ↔ cliente).

**Nota:** o código está planeado como um esqueleto completo — fornece uma base sólida e executável para desenvolver funcionalidades adicionais. Foi dada prioridade à clareza, modularidade e aos requisitos do enunciado: **MySQL, sockets TCP, herança e módulos**.

## 1) README.md

```
# Sistema de Restaurante Delivery

## Conteúdo deste pacote

- `sql/schema.sql` – esquema MySQL para criar as tabelas.
- `server/` – código do servidor (socket TCP + integração MySQL).
- `client/` – código do cliente (CLI) que se conecta ao servidor.
- `tests/` – scripts de apoio e testes (popular BD, teste de concorrência).
- `requirements.txt` – dependências Python.

## Requisitos

- Python 3.10+
- MySQL Server (no servidor)
- Rede entre as duas máquinas (servidor e cliente)

## Instalação rápida (Servidor)

1. Instale MySQL e crie a base de dados:
   - Execute `mysql -u root -p` e depois rode o ficheiro `sql/schema.sql`.

2. Instale dependências Python (num virtualenv recomendado):

```bash
pip install -r server/requirements.txt
```

1. Configure credenciais do MySQL em `server/.env` (modelo incluído):

```
MYSQL_HOST=localhost
MYSQL_PORT=3306
MYSQL_USER=delivery_user
MYSQL_PASSWORD=yourpassword
```

```
MYSQL_DB=bd_delivery
SERVER_HOST=0.0.0.0
SERVER_PORT=5000
```

1. Execute o servidor:

```
python server/main.py --host 0.0.0.0 --port 5000
```

## Instalação (Cliente)

No computador cliente, instale dependências:

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

Edite `client/.env` para apontar para o IP do servidor ( `SERVER_IP` ) e porta.

Execute o cliente:

```
python client/main.py --server-ip 192.168.1.20 --port 5000
```

## Testes

- Popular BD com dados de exemplo:

```
python tests/populate_db.py --db-host 192.168.1.20
```

- Teste de concorrência (simula várias compras do mesmo produto):

```
python tests/test_concurrency.py --server-ip 192.168.1.20 --port 5000
```

## Observações de design

- Mensagens entre cliente e servidor usam JSON por linha ( `\n` como delimitador).
- Sessões: após `LOGIN`, o servidor devolve um `session_token` que o cliente inclui nas próximas mensagens.
- Operações que alteram stock usam transações MySQL.
- Monitor de stock envia alertas em tempo real ( `ALERTA_STOCK_BAIXO` ) a administradores online.

## 2) SQL — `sql/schema.sql`

```
-- Schema para bd_delivery
CREATE DATABASE IF NOT EXISTS bd_delivery;
USE bd_delivery;

CREATE TABLE IF NOT EXISTS users (
  id INT AUTO_INCREMENT PRIMARY KEY,
  username VARCHAR(50) UNIQUE NOT NULL,
  password_hash VARCHAR(255) NOT NULL,
  role ENUM('admin', 'entregador', 'cliente') NOT NULL,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE IF NOT EXISTS produtos (
  id INT AUTO_INCREMENT PRIMARY KEY,
  nome VARCHAR(100) NOT NULL,
  categoria VARCHAR(50),
  descricao TEXT,
  preco DECIMAL(10,2) NOT NULL DEFAULT 0.00,
  stock INT NOT NULL DEFAULT 0,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  UNIQUE (nome)
);

CREATE TABLE IF NOT EXISTS pedidos (
  id INT AUTO_INCREMENT PRIMARY KEY,
  cliente_id INT NOT NULL,
  total DECIMAL(10,2) NOT NULL DEFAULT 0.00,
  status ENUM('em_preparacao', 'pronto_entrega', 'entregue') NOT NULL DEFAULT
'em_preparacao',
  data_pedido DATETIME DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (cliente_id) REFERENCES users(id)
);

CREATE TABLE IF NOT EXISTS pedido_items (
  id INT AUTO_INCREMENT PRIMARY KEY,
  pedido_id INT NOT NULL,
  produto_id INT NOT NULL,
  quantidade INT NOT NULL,
  preco_unit DECIMAL(10,2) NOT NULL,
  FOREIGN KEY (pedido_id) REFERENCES pedidos(id) ON DELETE CASCADE,
  FOREIGN KEY (produto_id) REFERENCES produtos(id)
);

CREATE TABLE IF NOT EXISTS stock_alerts (
  id INT AUTO_INCREMENT PRIMARY KEY,
  produto_id INT NOT NULL,
  nivel INT NOT NULL,
  alerted_at DATETIME DEFAULT CURRENT_TIMESTAMP,
```

```

    FOREIGN KEY (produto_id) REFERENCES produtos(id)
);

CREATE TABLE IF NOT EXISTS logs (
    id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT,
    action VARCHAR(100),
    details TEXT,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);

-- Indexes úteis
CREATE INDEX idx_produtos_stock ON produtos(stock);
CREATE INDEX idx_pedidos_data ON pedidos(data_pedido);

```

### 3) server/requirements.txt

```

mysql-connector-python==8.0.*
python-dotenv
bcrypt

```

### 4) client/requirements.txt

```

python-dotenv

```

## 5) Servidor — ficheiros principais

server/.env.example

```

MYSQL_HOST=localhost
MYSQL_PORT=3306
MYSQL_USER=delivery_user
MYSQL_PASSWORD=yourpassword
MYSQL_DB=bd_delivery
SERVER_HOST=0.0.0.0
SERVER_PORT=5000

```

server/db.py

```

# server/db.py
import mysql.connector
from mysql.connector import Error

```

```

from dotenv import load_dotenv
import os

load_dotenv()

DB_CONFIG = {
    'host': os.getenv('MYSQL_HOST', 'localhost'),
    'port': int(os.getenv('MYSQL_PORT', 3306)),
    'user': os.getenv('MYSQL_USER', 'root'),
    'password': os.getenv('MYSQL_PASSWORD', ''),
    'database': os.getenv('MYSQL_DB', 'bd_delivery'),
}

def get_connection():
    return mysql.connector.connect(**DB_CONFIG)

# Helper functions used by handlers
def create_user(username, password_hash, role='cliente'):
    conn = get_connection()
    try:
        cur = conn.cursor()

        cur.execute("INSERT INTO users (username, password_hash, role) VALUES (%s,%s,%s)",
                    (username, password_hash, role))
        conn.commit()
        return cur.lastrowid
    finally:
        cur.close()
        conn.close()

# ... outras funções (get_user_by_username, CRUD de produtos, pedidos) serão
chamadas pelos handlers

```

server/models.py

```

# server/models.py
from dataclasses import dataclass

@dataclass
class User:
    id: int
    username: str
    role: str

class Admin(User):
    def __init__(self, id, username):
        super().__init__(id=id, username=username, role='admin')

```

```

# métodos administrativos (interfaces que os handlers usam)

class Entregador(User):
    def __init__(self, id, username):
        super().__init__(id=id, username=username, role='entregador')

class Cliente(User):
    def __init__(self, id, username):
        super().__init__(id=id, username=username, role='cliente')

@dataclass
class Produto:
    id: int
    nome: str
    categoria: str
    descricao: str
    preco: float
    stock: int

@dataclass
class PedidoItem:
    produto_id: int
    quantidade: int
    preco_unit: float

@dataclass
class Pedido:
    id: int
    cliente_id: int
    itens: list
    total: float
    status: str

```

server/protocol.py

```

# server/protocol.py
# Constantes e helpers para mensagens JSON

# códigos de retorno
CODES = {
    'ATUALIZACAO_OK': 'ATUALIZACAO_OK',
    'PRODUTO_NAO_ENCONTRADO': 'PRODUTO_NAO_ENCONTRADO',
    'PRODUTO_REMOVIDO': 'PRODUTO_REMOVIDO',
    'VENDA_CONFIRMADA': 'VENDA_CONFIRMADA',
    'STOCK_INSUFICIENTE': 'STOCK_INSUFICIENTE',
    'ALERTA_STOCK_BAIIXO': 'ALERTA_STOCK_BAIIXO',
    'PRODUTO_ADICIONADO': 'PRODUTO_ADICIONADO',
    'ERRO_DUPLICADO': 'ERRO_DUPLICADO',
}

```

```

}

# actions esperadas do cliente
ACTIONS = [
    'LOGIN', 'REGISTER',
    'LIST_PRATOS', 'ADD_PRATO', 'UPDATE_PRATO', 'REMOVE_PRATO',
    'REGISTRAR_PEDIDO', 'HISTORICO_PEDIDOS',
    'CONSULTAR_STOCK', 'ATUALIZAR_STOCK'
]

# Mensagem JSON por linha (delimitador '\n')

```

#### server/handlers.py

```

# server/handlers.py
import json
from db import get_connection
from protocol import CODES
import bcrypt
from uuid import uuid4

# sessão simples em memória: session_token -> user dict
SESSIONS = {}

def handle_register(payload):
    username = payload.get('username')
    password = payload.get('password')
    role = payload.get('role', 'cliente')
    # hash password
    pw_hash = bcrypt.hashpw(password.encode(), bcrypt.gensalt()).decode()
    conn = get_connection()
    try:
        cur = conn.cursor()

        cur.execute("INSERT INTO users (username, password_hash, role) VALUES (%s,%s,%s)",
                    (username, pw_hash, role))
        conn.commit()
        return {'status': 'OK', 'code': 'REGISTER_OK', 'payload': {'user_id':
cur.lastrowid}}
    except Exception as e:
        return {'status': 'ERROR', 'code': 'REGISTER_ERRO', 'message': str(e)}
    finally:
        cur.close(); conn.close()

def handle_login(payload):
    username = payload.get('username')
    password = payload.get('password')

```

```

conn = get_connection()
try:
    cur = conn.cursor(dictionary=True)
    cur.execute("SELECT * FROM users WHERE username=%s", (username,))
    row = cur.fetchone()
    if not row:
        return {'status': 'ERROR', 'code': 'AUTH_FAILED'}
    if not bcrypt.checkpw(password.encode(),
row['password_hash'].encode()):
        return {'status': 'ERROR', 'code': 'AUTH_FAILED'}
    token = str(uuid4())
    SESSIONS[token] = {'user_id': row['id'], 'username': row['username'],
'role': row['role']}
    return {'status': 'OK', 'code': 'AUTH_OK', 'payload':
{'session_token': token, 'role': row['role'], 'user_id': row['id']}}
finally:
    cur.close(); conn.close()

# placeholder para outros handlers: add_produto, list_produtos,
registrar_pedido, etc.
# cada handler deve verificar permissões usando session_token

```

server/stock\_monitor.py

```

# server/stock_monitor.py
# Thread que verifica stock e envia alertas para admins conectados
import threading
import time
from db import get_connection

CHECK_INTERVAL = 10 # segundos (ajuste conforme necessário)

class StockMonitor(threading.Thread):
    def __init__(self, send_alert_to_admins_fn):
        super().__init__(daemon=True)
        self.send_alert = send_alert_to_admins_fn
        self.running = True

    def run(self):
        while self.running:
            conn = get_connection()
            try:
                cur = conn.cursor(dictionary=True)

cur.execute("SELECT id,nome,stock FROM produtos WHERE stock < 5")
                rows = cur.fetchall()
                for r in rows:
                    # enviar alerta
                    self.send_alert({'produto_id': r['id'], 'nome':

```

```

r['nome'], 'stock': r['stock']})
    finally:
        cur.close(); conn.close()
        time.sleep(CHECK_INTERVAL)

def stop(self):
    self.running = False

```

server/main.py

```

# server/main.py
import socket
import threading
import json
from handlers import handle_login, handle_register, SESSIONS
from stock_monitor import StockMonitor
from protocol import CODES
import argparse
import os
from dotenv import load_dotenv

load_dotenv()

HOST = os.getenv('SERVER_HOST', '0.0.0.0')
PORT = int(os.getenv('SERVER_PORT', 5000))

# Mapa de sockets de admins conectados para envio de alertas
ADMIN_SOCKETS = set()

def send_json(conn, obj):
    data = json.dumps(obj) + '\n'
    conn.sendall(data.encode())

def recv_json(conn):
    buffer = b''
    while True:
        chunk = conn.recv(4096)
        if not chunk:
            return None
        buffer += chunk
        if b'\n' in buffer:
            line, rest = buffer.split(b'\n', 1)
            # Note: rest leftover ignored for simplicity; in production
            # buffer leftover must be handled
            return json.loads(line.decode())

def client_thread(conn, addr):

```

```

print('Conexão de', addr)
try:
    while True:
        msg = recv_json(conn)
        if msg is None:
            break
        action = msg.get('action')
        payload = msg.get('payload', {})
        # dispatch básico
        if action == 'LOGIN':
            resp = handle_login(payload)
            send_json(conn, resp)
            # se admin, adicionar socket a ADMIN_SOCKETS (após sucesso)
            if resp.get('status') == 'OK' and resp['payload']
['role'] == 'admin':
                ADMIN_SOCKETS.add(conn)
            elif action == 'REGISTER':
                resp = handle_register(payload)
                send_json(conn, resp)
            else:
                send_json(conn, {'status': 'ERROR', 'code': 'UNKNOWN_ACTION'})
finally:
    print('Fechando conexão', addr)
    try:
        ADMIN_SOCKETS.discard(conn)
        conn.close()
    except:
        pass

def start_server(host, port):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind((host, port))
    sock.listen(5)
    print(f"Servidor a ouvir em {host}:{port}")

    def send_alert_to_admins(alert):
        for s in list(ADMIN_SOCKETS):
            try:
                data =
json.dumps({'status': 'ALERT', 'code': CODES['ALERTA_STOCK_BAIXO'], 'payload': alert})
+ '\n'
                s.sendall(data.encode())
            except Exception:
                ADMIN_SOCKETS.discard(s)

    monitor = StockMonitor(send_alert_to_admins)
    monitor.start()

try:

```

```

        while True:
            conn, addr = sock.accept()
            threading.Thread(target=client_thread, args=(conn,addr),
daemon=True).start()
        except KeyboardInterrupt:
            print('Servidor a encerrar...')
            monitor.stop()
            sock.close()

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--host', default=HOST)
    parser.add_argument('--port', default=PORT, type=int)
    args = parser.parse_args()
    start_server(args.host, args.port)

```

## 6) Cliente — ficheiros principais

client/.env.example

```

SERVER_IP=127.0.0.1
SERVER_PORT=5000

```

client/protocol.py

```

# client/protocol.py
# Constantes que espelham server/protocol.py
ACTIONS = [

    'LOGIN', 'REGISTER', 'LIST_PRATOS', 'ADD_PRATO', 'UPDATE_PRATO', 'REMOVE_PRATO',

    'REGISTRAR_PEDIDO', 'HISTORICO_PEDIDOS', 'CONSULTAR_STOCK', 'ATUALIZAR_STOCK'
]

# Para enviar mensagens JSON por linha

```

client/main.py

```

# client/main.py
import socket
import json
import argparse
import os
from dotenv import load_dotenv

```

```

load_dotenv()

SERVER_IP = os.getenv('SERVER_IP')
SERVER_PORT = int(os.getenv('SERVER_PORT', 5000))

def send_rcv(server_ip, port, message):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((server_ip, port))
    sock.sendall((json.dumps(message) + '\n').encode())
    # rcv single line
    buffer = b''
    while True:
        chunk = sock.recv(4096)
        if not chunk:
            break
        buffer += chunk
        if b'\n' in buffer:
            line, _ = buffer.split(b'\n', 1)
            return json.loads(line.decode())
    return None

def interactive_loop(server_ip, port):
    print('Cliente CLI - conectar a', server_ip, port)
    session_token = None
    role = None
    while True:
        print('\nMenu:\n1) Login\n2) Register\n3) Listar pratos\n4) Sair')
        opt = input('> ').strip()
        if opt == '1':
            username = input('Username: ')
            password = input('Password: ')
            msg = {'action': 'LOGIN', 'payload':
{'username': username, 'password': password}}
            resp = send_rcv(server_ip, port, msg)
            print('Resp:', resp)
            if resp and resp.get('status') == 'OK':
                session_token = resp['payload']['session_token']
                role = resp['payload']['role']
        elif opt == '2':
            username = input('Username: ')
            password = input('Password: ')
            role_in = input('Role (admin/entregador/cliente) [cliente]: ') or
'cliente'
            msg = {'action': 'REGISTER', 'payload':
{'username': username, 'password': password, 'role': role_in}}
            resp = send_rcv(server_ip, port, msg)
            print('Resp:', resp)
        elif opt == '3':
            msg = {'action': 'LIST_PRATOS', 'payload': {}}

```

```

        resp = send_recv(server_ip, port, msg)
        print('Resp:', resp)
    elif opt == '4':
        break
    else:
        print('Opção inválida')

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--server-ip', default=SERVER_IP)
    parser.add_argument('--port', default=SERVER_PORT, type=int)
    args = parser.parse_args()
    interactive_loop(args.server_ip, args.port)

```

#### client/ui.py (opcional)

```

# client/ui.py
# Complemento: helpers para renderizar menus ou tratar mensagens recebidas do
servidor

def print_produtos(produtos):
    for p in produtos:
        print(f"{p['id']}: {p['nome']} - {p['preco']} (stock: {p['stock']})")

```

## 7) Scripts de Teste (tests/)

#### tests/populate\_db.py

```

# tests/populate_db.py
# Script para popular a BD com utilizadores e produtos de exemplo.
import mysql.connector
from dotenv import load_dotenv
import os
import argparse

load_dotenv('../server/.env')

DB_CONFIG = {
    'host': os.getenv('MYSQL_HOST', 'localhost'),
    'port': int(os.getenv('MYSQL_PORT', 3306)),
    'user': os.getenv('MYSQL_USER', 'root'),
    'password': os.getenv('MYSQL_PASSWORD', ''),
    'database': os.getenv('MYSQL_DB', 'bd_delivery')
}

EX_USERS = [

```

```

    ('admin','adminpass','admin'),
    ('entregador','entpass','entregador'),
    ('cliente','clientpass','cliente')
]

EX_PRODUTOS = [
    ('Pizza Margherita','Pizzas','Clássica com tomate e mozzarella',12.50,
10),
    ('Hambúrguer','Sandes','Carne bovina, alface, tomate',9.00,6),
    ('Salada César','Saladas','Alface, frango grelhado',7.50,3),
]

def main():
    conn = mysql.connector.connect(**DB_CONFIG)
    cur = conn.cursor()
    for u in EX_USERS:
        try:
            cur.execute("INSERT INTO users (username,password_hash,role)
VALUES (%s,%s,%s)", (u[0], u[1], u[2]))
        except Exception:
            pass
    for p in EX_PRODUTOS:
        try:
            cur.execute("INSERT INTO produtos
(nome,categoria,descricao,preco,stock) VALUES (%s,%s,%s,%s,%s)", p)
        except Exception:
            pass
    conn.commit()
    cur.close(); conn.close()
    print('População concluída')

if __name__ == '__main__':
    main()

```

*Nota:* neste script estamos a inserir `password_hash` em texto plano apenas para popular a BD; substitua por hash se quiser testar autenticação real.

`tests/test_concurrency.py`

```

# tests/test_concurrency.py
# Simula múltiplos clientes a pedir o mesmo produto ao mesmo tempo.
import threading
import socket
import json
import argparse

SERVER_IP = '127.0.0.1'
SERVER_PORT = 5000

```

```

NUM_CLIENTS = 6
PRODUTO_ID = 1
QUANTIDADE = 1

def make_order(client_idx):
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((SERVER_IP, SERVER_PORT))
        # Este script assume que o servidor aceita um REGISTRAR_PEDIDO sem
        # auth para testar race condition
        pedido = {
            'action': 'REGISTRAR_PEDIDO',
            'payload': {
                'cliente_id': 3,
                'itens': [{'produto_id': PRODUTO_ID, 'quantidade': QUANTIDADE}]
            }
        }
        s.sendall((json.dumps(pedido)+'\n').encode())
        data = b''
        while True:
            chunk = s.recv(4096)
            if not chunk:
                break
            data += chunk
            if b'\n' in data:
                line, _ = data.split(b'\n', 1)
                print(f'Client {client_idx} resposta:',
                      json.loads(line.decode()))
                break
        except Exception as e:
            print('Erro client', client_idx, e)

if __name__ == '__main__':
    threads = []
    for i in range(NUM_CLIENTS):
        t = threading.Thread(target=make_order, args=(i,))
        t.start()
        threads.append(t)
    for t in threads:
        t.join()
    print('Teste concorrência terminado')

```

## 8) Observações finais e próximos passos

- O código fornecido é um esqueleto: `handlers.py` e `db.py` precisam de mais funções CRUD completas para produtos, pedidos, histórico, etc. No entanto todas as peças fundamentais já

estão: sockets, mensagens JSON, gestão de sessões, monitor de stock, transacções MySQL (a implementar nas funções de pedido), herança via `models.py` e separação por módulos.

- Se quiser, eu posso agora:

- Gerar versões completas e testadas das funções CRUD ( `add_produto`, `update_produto`, `registrar_pedido` com transacções e `SELECT ... FOR UPDATE` ).
- Escrever um cliente com interface mais rica (por exemplo, curses) ou um GUI mínima.
- Criar Dockerfiles para servidor/cliente e uma `docker-compose` para testar localmente.

Diz o que preferes que eu faça a seguir: "Completar handlers & db (pedido transaccional)" / "Gerar Dockerfiles" / "Criar GUI" / outra coisa. ``