Curs 8
Programare Python pentru baze de date distribuite și mobile

8.1 Introducere în Python

Python este un limbaj de programare **puternic și ușor de învățat**, cu suport bogat pentru diferite paradigme (imperativ, orientat pe obiecte, funcțional). Sintaxa sa simplă, tipizarea dinamică și faptul că este interpretat îl fac ideal pentru **scriptare și dezvoltare rapidă de aplicații** pe majoritatea platformelor[1]. Python vine preinstalat cu un set vast de biblioteci standard și dispune de un ecosistem extins de pachete externe (gestionate de utilitarul pip), inclusiv biblioteci specializate pentru baze de date.

Caracteristici esențiale ale Python:

Python oferă structuri de date de nivel înalt (liste, dicționare, seturi etc.), suport bun pentru programare orientată pe obiecte și o comunitate uriașă care dezvoltă biblioteci în aproape orice domeniu. Codul Python este în general concis și ușor de citit, ceea ce accelerează dezvoltarea și mentenanța. Fiind un limbaj interpretat, executarea codului este interactivă – utilă la testare și experimentare – însă performanța brută poate fi mai redusă comparativ cu limbaje compilate; totuși, pentru multe aplicații de baze de date și web, Python oferă performanță suficientă și poate apela componente native pentru sarcini intense.

Python și bazele de date:

Limbajul Python se integrează foarte bine cu diverse sisteme de gestiune a bazelor de date (SGBD). Există drivere și biblioteci conforme cu specificația Python DB-API 2.0 (PEP 249) pentru majoritatea SGBD-urilor relaționale (de ex. MySQL, PostgreSQL, Oracle, SQL Server), precum și biblioteci pentru baze de date NoSQL (MongoDB, Redis etc.). Vom explora în secțiunile următoare câteva biblioteci-cheie: modulul standard sqlite3 (pentru SQLite), biblioteca **SQLAlchemy** (un ORM pentru baze relaționale) și **PyMongo** (driver pentru MongoDB). Aceste unelte permit dezvoltatorilor Python să acceseze și să manipuleze datele într-un mod simplificat, folosind atât interogări SQL, cât și operații în stil document.

Pregătirea mediului Python:

Pentru a utiliza Python în dezvoltarea aplicațiilor distribuite, asigurați-vă că aveți instalată o versiune 3.x actualizată. Managerul de pachete pip vă permite instalarea ușoară a bibliotecilor terțe, de exemplu: pip install sqlalchemy pymongo flask. De asemenea, este recomandat să folosiți un mediu virtual (venv) pentru a izola dependențele proiectului. Studenții familizarizați cu SQL și PL/SQL vor regăsi în Python un limbaj mai general-purpose, capabil să combine logica aplicației cu operațiile pe baza de date, în afara SGBD-ului. În continuare, vom trece la prezentarea bibliotecilor specifice pentru baze de date în Python, urmată de exemple de cod și studii de caz.

8.2 Biblioteci Python pentru baze de date (sqlite3, SQLAlchemy, PyMongo)

Python oferă suport excelent pentru lucrul cu baze de date, atât relaționale, cât și NoSQL. Vom discuta pe rând trei biblioteci reprezentative:

- **sqlite3** – modulul standard din Python pentru baze de date SQLite (SQL relațional, local).

- **SQLAlchemy** – bibliotecă ORM (Object-Relational Mapping) pentru baze de date SQL, care abstractizează interacțiunea prin obiecte Python.
- **PyMongo** – driver oficial pentru MongoDB, o bază de date NoSQL orientată pe documente.

8.2.1 Modulul sqlite3 (baze de date SQLite)

SQLite este o bază de date relațională ușoară, stocată într-un fișier, care **nu necesită un server dedicat**. Python include în biblioteca standard modulul sqlite3, permițând crearea și accesarea bazelor de date SQLite folosind sintaxă SQL standard. SQLite este ideal pentru **stocarea locală a datelor, prototipare și aplicații mobile** – de altfel, multe aplicații mobile (Android/iOS) folosesc SQLite ca stocare internă. Un avantaj major este simplitatea: copierea fișierului de bază de date copiază efectiv baza de date.
Python expune SQLite printr-o interfață conformă DB-API 2.0[2].
Iată un exemplu simplu de utilizare a modulului sqlite3:

```
import sqlite3

# 1. Creare/conectare la baza de date (fizic, fișier local)
conn = sqlite3.connect("exemplu.db")  # dacă nu există, va fi creat
cur = conn.cursor()

# 2. Executarea unui DDL: creare tabel
cur.execute("""
   CREATE TABLE IF NOT EXISTS produse (
      id INTEGER PRIMARY KEY,
      nume TEXT,
      pret REAL
   )
""")

# 3. Inserarea unor date
cur.execute("INSERT INTO produse (nume, pret) VALUES (?, ?)", ("Telefon", 1200.5))
cur.execute("INSERT INTO produse (nume, pret) VALUES (?, ?)", ("Laptop", 3400.0))
conn.commit()  # confirmă tranzacția, scrie pe disc

# 4. Interogarea datelor
cur.execute("SELECT id, nume, pret FROM produse")
for (pid, nume, pret) in cur.fetchall():
   print(f"Produs {pid}: {nume} - pret {pret} RON")
```

În exemplul de mai sus, am deschis o conexiune către baza de date exemplu.db (creând fișierul dacă nu exista), apoi am obținut un **cursor** pentru a executa comenzi SQL. Am creat un tabel denumit produse cu trei coloane și am inserat două înregistrări folosind parametri (semnele ? în interogare, legate de valori prin tuplu) – aceasta este cea mai bună practică pentru a evita injecțiile SQL. Operațiile de modificare sunt efectuate într-o tranzacție care trebuie confirmată prin

conn.commit(). În final, am rulat o interogare SELECT și am afișat rezultatele parcurse cu fetchall().

Observații:

- Modulul sqlite3 tratează automat tipurile de bază; de exemplu, valorile Python float și str sunt mapate la tipuri SQLite REAL și TEXT.
- SQLite suportă tranzacții ACID. Dacă nu se face commit(), modificările rămân neconfirmate (se pot face rollback). Conexiunea are mod autocommit dezactivat implicit, conform specificației DB-API.
- Aceeași bază de date SQLite poate fi accesată simultan din mai multe conexiuni, dar concurența este limitată de un mecanism de **blocare la nivel de fișier** – SQLite permite multiple citiri concurrente, însă scrierile sunt exclusiviste. Pentru scenarii cu mulți utilizatori sau cerințe ridicate de concurență, se recomandă un SGBD server client robust (PostgreSQL, MySQL etc.), SQLite fiind destinat uzului embedded sau la scară mică.

În concluzie, sqlite3 în Python oferă o soluție rapidă pentru stocarea de date locale și testare. Multe aplicații folosesc SQLite ca etapă inițială de dezvoltare, apoi portarea spre un SGBD mai mare se face ușor, codul Python rămânând similar[3]. Prin înlocuirea lanțului de conexiune (connection string) și folosirea unui driver adecvat, aplicația poate trece la Oracle, PostgreSQL etc. fără mari modificări logice.

8.2.2 SQLAlchemy – ORM pentru baze de date relaționale

SQLAlchemy este o bibliotecă Python puternică pentru lucrul cu baze de date relaționale, oferind atât un **nivel de abstracție tip ORM (Object-Relational Mapper)**, cât și un **nivel de SQL expresiv (Core)**. Cu SQLAlchemy, programatorii pot defini **modele Python** care sunt mapate la tabele din baza de date, permițând interogarea și manipularea datelor folosind obiecte în loc de SQL textual. Acest lucru crește productivitatea și siguranța (reduce erorile de sintaxă SQL și dependența de un anumit dialect de SGBD). SQLAlchemy suportă o gamă largă de SGBD-uri: SQLite, PostgreSQL, MySQL/MariaDB, Oracle, SQL Server ș.a., adaptând sintaxa pentru fiecare în mod transparent[4].

Conceptul ORM:

Obiectele Python dintr-o clasă mapată (Model) corespund rândurilor din tabelul asociat. Coloanele tabelului devin atribute ale obiectului. Operațiile CRUD (Creare, Citire, Actualizare, Ștergere) se realizează prin metode Python, iar SQLAlchemy se ocupă de generarea și execuția automată a instrucțiunilor SQL necesare (INSERT, SELECT, UPDATE, DELETE)[5][6]. De asemenea, SQLAlchemy facilitează definirea de **relații** (one-to-many, many-to-many etc.) între modele, folosind chei străine și acces prin atribute, ca în exemplul de mai jos.

Exemplu de utilizare SQLAlchemy (ORM): să presupunem că dorim să gestionăm utilizatori și articole într-o aplicație. Definim două clase Python care moștenesc un Base comun furnizat de SQLAlchemy, indicând tabelul și coloanele:

```
from sqlalchemy import create_engine, Column, Integer, String, ForeignKey
from sqlalchemy.orm import declarative_base, sessionmaker, relationship

# Configurare engine către o bază de date SQLite (pentru simplitate)
engine = create_engine('sqlite:///aplicatie.db')
```

```
Base = declarative_base()

# Definire modele ORM
class Utilizator(Base):
    __tablename__ = 'utilizatori'
    id = Column(Integer, primary_key=True)
    nume = Column(String, unique=True)
    # Relație One-to-Many cu Articol (un utilizator -> mai multe articole)
    articole = relationship("Articol", back_populates="autor")

class Articol(Base):
    __tablename__ = 'articole'
    id = Column(Integer, primary_key=True)
    titlu = Column(String)
    continut = Column(String)
    autor_id = Column(Integer, ForeignKey('utilizatori.id'))
    # Relație Many-to-One cu Utilizator
    autor = relationship("Utilizator", back_populates="articole")

# Creare efectivă a tabelelor în baza de date
Base.metadata.create_all(engine)

# Inițializare sesiune (unitate de lucru pentru operații cu BD)
Session = sessionmaker(bind=engine)
session = Session()

# Creare obiecte și salvare în baza de date
u = Utilizator(nume="ana.popescu")
art = Articol(titlu="Salut lume", continut="Primul meu articol", autor=u)
session.add(u)
session.add(art)
session.commit()

# Interogare: obținere articole postate de utilizatorul cu numele "ana.popescu"
rezultate = session.query(Articol).join(Articol.autor).filter(Utilizator.nume ==
"ana.popescu").all()
for art in rezultate:
    print(f"{art.titlu} - autor {art.autor.nume}")
```

În acest exemplu:
- Am configurat un *engine* către baza de date (folosind un URL – aici un fișier SQLite local). SQLAlchemy suportă dialecți specifice SGBD-urilor, de exemplu 'postgresql+psycopg2://user:pass@host:port/baza' pentru PostgreSQL, 'mysql+pymysql://...' pentru MySQL etc.
- Am creat o clasă de bază declarativă Base și două clase Utilizator și Articol care extind Base. Atributul special __tablename__ definește numele tabelului din BD. Fiecare coloană este definită

prin obiecte Column – specificăm tipul (Integer, String etc.) și opțiuni (ex. primary_key=True pentru cheia primară, ForeignKey pentru chei străine, unique=True pentru unicitate).
- Relațiile sunt declarate cu relationship: la Utilizator.articole definim relația One-to-Many către clasa Articol (SQLAlchemy va căuta automat cheia externă corespunzătoare, Articol.autor_id). Parametrul back_populates asigură legătura bidirecțională – în clasa Articol, atributul autor este perechea care referă înapoi la Utilizator. Astfel putem naviga ușor: art.autor.nume dă numele autorului unui articol, iar utilizator.articole este listă de articole scrise de acel utilizator[7][8].
- Apelul Base.metadata.create_all(engine) creează tabelele fizice în baza de date, pe baza definițiilor claselor (dacă ele nu există deja). Aceasta ne scutește de a scrie manual SQL de tip CREATE TABLE.
- Am inițializat o **sesiune** ORM – un obiect de lucru prin care efectuăm operații. Am creat instanțe Python pentru un utilizator și un articol, setând relațiile prin atribuire (autor=u). Când adăugăm obiectele în sesiune și facem commit, SQLAlchemy va genera și executa automat instrucțiuni SQL: un INSERT în utilizatori și unul în articole, având grijă de valorile cheilor primare/străine.
- Exemplul de interogare arată puterea ORM: folosim session.query(Articol).join(Articol.autor).filter(Utilizator.nume == "...") pentru a obține articolele filtrate după numele autorului. Acest query ORM va fi tradus într-un JOIN între tabelele articole și utilizatori cu condiția utilizatori.nume = 'ana.popescu'[6][9]. Rezultatul este o listă de obiecte Articol, din care putem accesa direct atributul legat autor (fără a scrie alt query pentru utilizator).

Beneficii SQLAlchemy/ORM: dezvoltatorul lucrează cu entități de nivel înalt (obiecte) fără a scrie SQL repetitiv. ORM-ul gestionează **sincronizarea cu baza de date**, incapsulează tranzacțiile (sesiunea poate grupa multiple operații înainte de commit)[10], oferă mecanisme de **rollback** în caz de eroare[11] și permite definirea de constrângeri și relații la nivel de model. De asemenea, este portabil – același cod ORM poate folosi diferite SGBD modificând doar configurația de conectare.

Studii de caz moderne:
SQLAlchemy este folosit pe scară largă în industrie, stând la baza multor framework-uri web Python (ex: Flask poate folosi extensia Flask-SQLAlchemy pentru integrare ușoară cu BD). De exemplu, o aplicație web Flask poate defini modele SQLAlchemy pentru entitățile sale și poate utiliza aceste modele direct în rutele web pentru a realiza operații CRUD[12][13].
Un alt exemplu: Instagram (o aplicație la scară foarte mare) folosea la început Python+Django – Django are un ORM similar conceptului SQLAlchemy – pentru a gestiona milioane de utilizatori și volume masive de date.
Chiar și în medii de analiză de date, ORM-ul poate ajuta la prototiparea accesului la BD relaționale într-un mod structurat. Totuși, e bine de știut că în cazuri de **cerințe de performanță extremă**, uneori se recurge la SQL "manual" optimizat; SQLAlchemy permite oricând execuția de SQL brut dacă este nevoie, sau combinarea cu procedural SQL (ex: apelarea unui Stored Procedure).

Limitări și bune practici: ORM-urile simplifică mult dezvoltarea, dar pot introduce suprasarcină. Este important să înțelegem cum produce SQL interogările complexe (profilarea query-urilor generate ajută la optimizare). De asemenea, pentru operații de masă (ex: inserarea a mii de rânduri), ORM-ul poate fi mai lent – SQLAlchemy oferă totuși mecanisme precum **bulk insert** și

posibilitatea de a folosi direct SQLAlchemy Core (expresii SQL) pentru astfel de situații. Per ansamblu, SQLAlchemy rămâne un instrument de bază pentru aplicațiile Python ce interacționează cu baze de date relaționale, accelerând dezvoltarea și permițând scrierea unui cod mai curat și mai sigur.

8.2.3 PyMongo – acces la MongoDB (baze de date NoSQL)

Pentru a lucra cu baze de date NoSQL în Python, exemplificăm folosind MongoDB – o bază de date orientată pe documente, foarte populară pentru aplicații web și big data. **MongoDB** stochează datele sub formă de documente **JSON** (intern, BSON – o variantă binară de JSON) în colecții, în loc de rânduri în tabele, ca în modelelor relaționale. Acest model flexibil înseamnă că nu avem un *schelet strict (schemă)* pentru date – documentele din aceeași colecție pot avea structuri diferite, câmpuri diferite, pot fi adăugate câmpuri noi oricând, fără migrarea întregii baze[14][15].

MongoDB se concentrează pe **scalare orizontală** (poate distribui datele pe mai multe noduri), replicare pentru disponibilitate și oferă un limbaj puternic de interogare, cu suport pentru indexare, agregare, căutare text, geo-spațială etc.[16]. Această flexibilitate și scalabilitate fac MongoDB foarte atractiv pentru **aplicații distribuite la scară largă**, care manipulează date semi-structurate sau nestructurate (de ex. date de jurnalizare, date generate de utilizatori, profiluri flexibile).

Biblioteca **PyMongo** este **driver-ul oficial** pentru interacțiunea Python cu MongoDB. PyMongo permite conectarea la un server MongoDB (local sau cluster), inserarea și interogarea documentelor, actualizări, ștergeri, administrarea bazelor de date și colecțiilor etc., totul prin apeluri Python. Spre deosebire de ORM, PyMongo este o interfață de nivel mai jos (low-level) care operează cu structuri de date primitive (dicționare Python, liste) ce corespund documentelor JSON. Există și biblioteci de nivel mai înalt, cum ar fi **MongoEngine** (un ODM – Object Document Mapper, similar unui ORM pentru documente), dar aici vom folosi PyMongo pentru claritate.

Exemplu de utilizare PyMongo: presupunem că avem MongoDB rulând local (pe portul implicit 27017).
Vom crea o bază de date pentru o aplicație de senzori IoT și o colecție de exemplu:

```
from pymongo import MongoClient

# Conectare la serverul MongoDB local (localhost:27017)
client = MongoClient("mongodb://localhost:27017/")

# Crearea/referința la o bază de date și o colecție
db = client["monitorizare_senzori"]
col = db["temperaturi"]

# Inserarea unui document în colecție
sensor_data = {"sensor_id": 101, "locatie": "Laborator 1", "valoare": 23.5, "unitate": "C"}
result = col.insert_one(sensor_data)
print(f"Document inserat cu _id = {result.inserted_id}")
```

```
# Inserarea multiplă de documente
vals = [
   {"sensor_id": 102, "locatie": "Laborator 1", "valoare": 22.3, "unitate": "C"},
   {"sensor_id": 101, "locatie": "Laborator 1", "valoare": 24.1, "unitate": "C"}
]
col.insert_many(vals)

# Interogare: găsește toate măsurătorile senzorului 101
for doc in col.find({"sensor_id": 101}):
   print(doc)
```

Explicații pentru codul de mai sus:
- Ne conectăm la serverul MongoDB prin crearea unui MongoClient. PyMongo acceptă URL-ul de conectare în format MongoDB; dacă nu specificăm, folosește implicit localhost:27017. Opțional, putem specifica parametri precum autentificare, replicaset etc.
- Accesăm baza de date numită "monitorizare_senzori". În MongoDB, bazele de date se creează automat la prima utilizare (când inserăm un prim document). Similar, accesăm colecția "temperaturi" din această bază de date. Nu este necesar un pas explicit de creare a bazei sau colecției – **MongoDB va crea automat colecția la prima inserare**[17].
- Inserăm un document (un dicționar Python) folosind insert_one. MongoDB va adăuga automat câmpul _id ca identificator unic (dacă nu este furnizat). PyMongo ne întoarce un obiect de rezultat (InsertOneResult) din care extragem inserted_id pentru a vedea ID-ul asignat[18].
- Folosim insert_many pentru a insera o listă de documente într-o singură operație (eficient pentru volume mari de date).
- Pentru interogare, folosim find cu un dicționar de filtrare – în exemplu, {"sensor_id": 101} – pentru a obține toate documentele unde sensor_id este 101. PyMongo returnează un cursor care iterează prin rezultatele găsite. Fiecare rezultat este un dicționar ce conține chei corespunzătoare câmpurilor documentului. Afișând direct doc, vom vedea un output de forma: {'_id': ObjectId('...'), 'sensor_id': 101, 'locatie': 'Laborator 1', 'valoare': 23.5, 'unitate': 'C'}.

Notă: Observăm că nu am definit nicio schemă – documentele inserate pentru sensor_id:101 pot avea câmpuri suplimentare față de altele sau pot lipsi câmpuri. Acest **schema-less design** oferă flexibilitate: dacă de mâine vrem să adăugăm câmpul "baterie" în măsurătorile senzorilor, putem începe să-l includem în noile documente fără vreo migrare, iar documentele vechi pur și simplu nu îl vor avea (caz care trebuie gestionat la citire). Dezavantajul este potențiala inconsistență dacă aplicația se așteaptă ca toate documentele să aibă anumite date – de aceea, deseori se aplică reguli de validare la nivel de colecție sau la nivel de aplicație, pentru a păstra o formă de disciplină. MongoDB permite setarea de validatori de schemă dacă se dorește.

Avantajele NoSQL/MongoDB: Scalabilitatea orizontală (prin *sharding* – distribuirea colecțiilor pe mai multe noduri), performanță ridicată la scrieri și citiri masive, flexibilitatea în modelarea datelor și un model de date foarte potrivit pentru obiecte complex imbricate (documentele pot conține array-uri și sub-documente). În plus, limbajul de interogare MongoDB este expresiv, permițând filtrări și proiecții avansate (inclusiv agregări de date direct pe server, similar cu GROUP BY din SQL). Pentru Python, **PyMongo face integrarea foarte ușoară**, iar combinația Python + MongoDB este des întâlnită în aplicații moderne ce necesită viteză de dezvoltare și adaptabilitate a modelului de date[19]. De exemplu, dacă aplicația voastră Python are

nevoie de o bază de date la fel de flexibilă precum limbajul însuși, MongoDB este o alegere potrivită[19].

Studii de caz și utilizări tipice:

Bazele de date NoSQL precum MongoDB sunt folosite în aplicații web de mare trafic (rețele sociale, sisteme de comentarii, sisteme de catalog de produse) unde structura datelor poate varia. De pildă, dacă am proiecta un sistem de profil al utilizatorilor pentru o rețea socială, am putea stoca în MongoDB documente cu informații de profil – unii utilizatori ar putea avea câmpuri extra (ex. link site personal) fără ca alții să le aibă, lucru dificil de modelat într-un tabel SQL rigid.

MongoDB este folosit și pentru stocarea datelor de tip jurnal (log-uri de evenimente), a datelor de telemetrie sau sesiuni de utilizator, unde scrierea rapidă și schema flexibilă contează. Multe companii combină tehnologiile: folosesc SQL acolo unde datele sunt strict structurate și tranzacțiile complexe sunt necesare, și NoSQL unde este nevoie de viteză și flexibilitate[14][20].

Un exemplu real este platforma de streaming Netflix, care folosește atât servicii pe baze relaționale pentru date critice (abonamente, plăți), cât și MongoDB/Cassandra pentru stocarea preferințelor de vizionare și a log-urilor, pentru a scala la nivel global.

Concluzie la 8.2: Python, împreună cu bibliotecile potrivite, permite acces uniform la diferite tipuri de baze de date. Studenții ar trebui să aleagă tehnologia în funcție de cerințe: pentru date relaționale și consistență puternică – un SGBD SQL cu un ORM poate fi preferat; pentru volum foarte mare de date distribuite și flexibilitate – un NoSQL cu driver dedicat poate simplifica dezvoltarea. De multe ori, soluția optimă implică **combinarea** tehnologiilor (de ex., folosirea unei baze SQL pentru tranzacțiile financiare și a unei baze NoSQL pentru analiza click-urilor utilizatorilor)[20]. Python strălucește prin ușurința cu care se poate conecta la ambele lumi și poate orchestra datele între ele.

8.3 Dezvoltarea aplicațiilor distribuite cu Python

O **aplicație distribuită** este un program ale cărui componente rulează pe mai multe calculatoare/dispozitive simultan și comunică printr-o rețea pentru a îndeplini împreună o sarcină[21].

Spre deosebire de o aplicație monolit (standalone) care rulează integral pe un singur sistem, o aplicație distribuită poate avea, de exemplu, un **frontend (client)** pe un dispozitiv al utilizatorului și un **backend (server)** pe un server la distanță, sau poate consta din mai multe servicii separate care colaborează (microservicii).

Avantajele aplicațiilor distribuite includ scalabilitatea (pot rula pe mai multe mașini pentru a deservi mai mulți utilizatori) și toleranța la defecte (dacă un nod cade, altele pot prelua sarcina)[22].

Dezavantajele vin din complexitate sporită: trebuie gestionate comunicația, sincronizarea și eventual probleme de consistență a datelor între noduri.

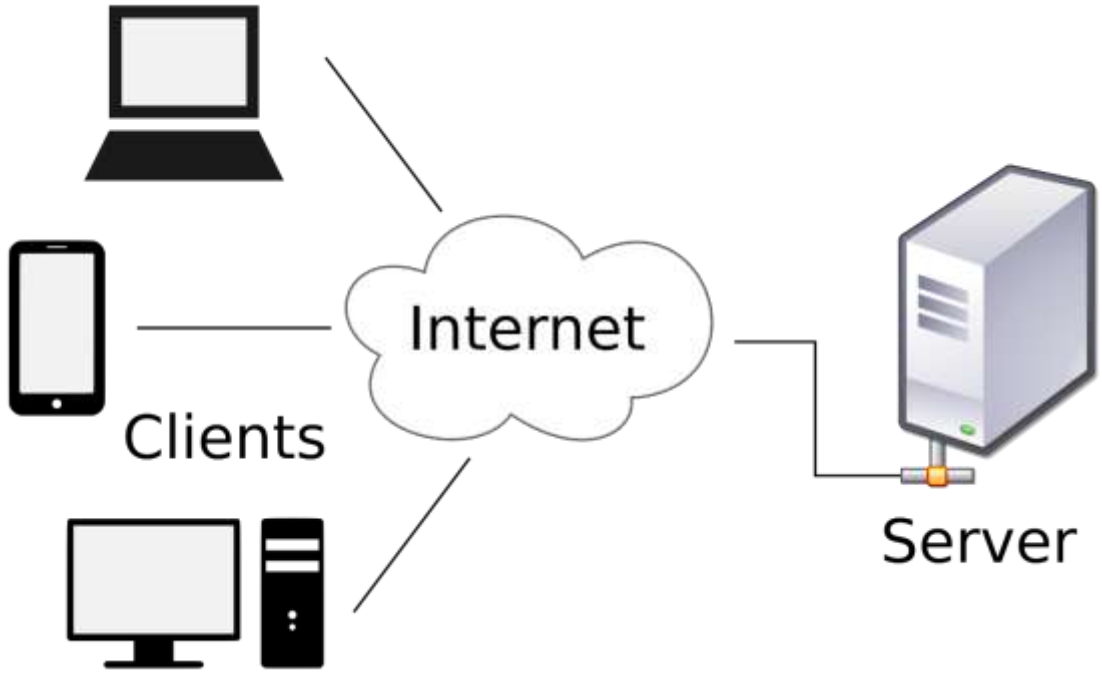

Figura 1: Modelul client–server, un exemplu simplu de aplicație distribuită, unde mai mulți clienți (programe utilizator) comunică prin rețea cu un server central pentru a accesa un serviciu sau resursă comună[23][24].

Arhitecturi distribuite comune:

Cel mai simplu tip este **arhitectura client-server** (ilustrată în figura de mai sus). Aici, un **server** oferă servicii (de exemplu, acces la o bază de date, autentificare, stocare de fișiere) și unul sau mai mulți **clienți** solicită servicii. Comunicarea se face de obicei prin protocoale standard (HTTP, TCP/IP, etc.) într-un model request-response: clientul trimite cereri, serverul procesează și întoarce răspunsuri[24]. Extensia acestui model este arhitectura multi-tier (de exemplu **trei niveluri**: prezentare – logică de aplicație – stocare de date[25]), unde un server web poate acționa ca intermediar între clientul front-end și un server de baze de date back-end. În ultimele decenii, a apărut și modelul pe microservicii: aplicația este împărțită în servicii mai mici, fiecare rulând independent (adesea containerizat în Docker) și comunicând prin API-uri sau mesaje. Aceste microservicii pot avea fiecare propria baza de date, adesea NoSQL, pentru a fi foarte scalabile și decuplate.

Python în rol de server și client: Python este foarte des folosit în dezvoltarea backend-urilor de aplicații web și distribuirea componentelor server. Există framework-uri web populare precum **Flask**, **Django** sau **FastAPI** care permit crearea rapidă de **servicii web (API REST)**.

Un serviciu web este o aplicație server care expune funcționalități (de exemplu, operații pe baza de date) prin endpoint-uri HTTP pe care clienții (de exemplu aplicații front-end, mobile sau alte servicii) le pot apela.

- *Exemplu:* Să considerăm o aplicație mobilă de inventar care are nevoie să sincronizeze datele cu un server central. Pe server, putem scrie în Python un API REST care permite aplicației mobile să trimită datele noi și să le ceară pe cele actualizate. Serverul Python

(folosind Flask, de pildă) va primi cererea, va accesa baza de date (posibil folosind SQLAlchemy sau PyMongo, după caz) și va întoarce un răspuns JSON. Avem astfel un **sistem distribuit**: multiple aplicații mobile (clienți) comunică cu un serviciu Python central (server) care la rândul lui interacționează cu o bază de date (sau mai multe). Dacă dorim scalare, putem rula mai multe instanțe ale serverului Python în containere Docker, și un load balancer va distribui cererile între ele.

Pentru a ilustra simplitatea cu care putem construi un *micro-serviciu* cu Python, iată un mini-exemplu folosind **Flask** (un micro-framework web foarte popular):

```
from flask import Flask, request, jsonify

app = Flask(__name__)

inventar = []  # resursă globală simplă (în practică ar fi o bază de date)

@app.route('/adauga', methods=['POST'])
def adauga_produs():
    data = request.get_json()  # datele trimise de client în format JSON
    inventar.append(data)
    return {"status": "Produs adăugat cu succes"}, 201

@app.route('/produse', methods=['GET'])
def listare_produse():
    # returnează lista întreagă de produse în format JSON
    return jsonify(inventar)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

În acest fragment, am definit două rute: POST /adauga pentru a adăuga un produs (clientul trimite detaliile produsului în format JSON în corpul cererii), și GET /produse pentru a obține lista tuturor produselor din inventar.

Am folosit o listă Python simplă ca stocare provizorie; într-o aplicație reală, în acele puncte am interoga o bază de date (de ex., să folosim SQLAlchemy să adauge un produs în tabelul aferent, sau PyMongo să insereze un document).

Totuși, și acest exemplu simplu demonstrează conceptul: un client (poate fi o aplicație front-end JavaScript sau o altă aplicație Python folosind biblioteca requests) poate face o cerere HTTP către serverul nostru Flask. De exemplu:

- O cerere POST la http://server:5000/adauga cu corp JSON {"id": 1, "nume": "Pix", "cantitate": 50} va provoca rularea funcției adauga_produs de pe server, care adaugă dict-ul în listă și răspunde cu status 201 (Created).
- Ulterior, o cerere GET la http://server:5000/produse va primi ca răspuns un JSON cu lista de produse (inclusiv obiectul adăugat anterior). Acesta poate fi consumat de client și afișat utilizatorului.

Un asemenea serviciu Python poate fi rulat într-un container Docker, și dacă traficul crește, putem avea mai multe containere identice, orchestrate de Kubernetes, pentru a servi în paralel cererile (fiecare instanță conectându-se la aceeași bază de date din spate). Astfel, am realizat o **scalare orizontală** a părții de aplicație. Python este compatibil cu instrumente moderne de cloud – de la

rularea în AWS Lambda (funcții serverless în cloud) până la microservicii complete pe AWS Elastic Beanstalk, Google Cloud Run, Azure App Service etc.

Alte librării și protocoale distribuite în Python: Pe lângă HTTP/REST (pentru care folosim adesea Flask/Django/FastAPI), Python suportă și alte abordări de comunicare distribuită:
- **RPC (Remote Procedure Call):** biblioteci precum xmlrpc.client / xmlrpc.server (în biblioteca standard) permit apeluri de funcții la distanță. De asemenea, biblioteci terțe ca **gRPC** (Google RPC) au binding-uri Python, permițând definiția unor servicii cu interfețe stricte (.proto) și comunicare eficientă binară.
- **Mesaje și cozi (message queues):** pentru arhitecturi orientate pe evenimente și microservicii decuplate, Python oferă clienți pentru sisteme de mesagerie precum RabbitMQ (ex. librăria pika), Kafka (kafka-python), Redis (folosit ca broker de mesaje prin redis-py). Astfel de sisteme permit comunicarea asincronă: un serviciu publică mesaje într-o coadă, altul le consumă, deci cooperarea este distribuită dar fără un apel direct. De exemplu, un serviciu de comandă produse poate publica un mesaj "comandă plasată" pe o coadă, iar alt serviciu separat (un microserviciu de stoc) primește mesajul și scade cantitatea din inventar; ambele servicii pot fi scrise în Python. Acest mod crește robustețea (sistemul continuă și dacă unul din servicii are temporar probleme, mesajele rămân în coadă).
- **Distributed computing / Big Data:** în zona de procesare distribuită a datelor, Python este integrat cu platforme precum **Apache Hadoop** și **Spark**. PySpark, de exemplu, permite scrierea de job-uri Spark (framework de procesare paralelă pe cluster) în Python. Biblioteci ca **Dask** și **Ray** permit crearea de aplicații Python care paralelizează sarcinile pe mai multe noduri/workeri, util în calcule științifice sau ML distribuit. Ca exemplu, Ray oferă un decoratori și mecanisme prin care transformă funcții Python obișnuite în task-uri distribuite pe un cluster, ascunzând complexitatea (vezi tutorialul “Writing your First Distributed Python Application with Ray” pentru o demonstrație)[26][27].

Securitate și concurență:

În aplicațiile distribuite, trebuie ținut cont de aspecte ca securizarea comunicațiilor (ex: folosirea HTTPS în loc de HTTP pentru a cripta datele între client și server) și gestionarea concurenței. Python oferă module pentru **threading**, **multiprocessing** și mecanisme asincrone (asyncio) care pot fi folosite pentru a crește gradul de paralelism al unui server. De exemplu, un server Flask tipic procesează cererile concurente fie prin multi-threading, fie cu ajutorul unui server extern (gunicorn) care lansează mai multe procese worker. Pentru operații I/O (precum așteptarea răspunsurilor de la baza de date sau de la alte servicii), Python 3 permite abordarea **async/await** (framework-uri ca FastAPI sunt construite în jurul acestui model, putând gestiona zeci de mii de conexiuni cu un consum redus de resurse, deoarece nu blochează firul de execuție când așteaptă I/O).

Studiu de caz integrator:

Să ne imaginăm o aplicație modernă de ridesharing (gen Uber). O astfel de platformă este inevitabil distribuită: există o aplicație mobilă pentru șoferi, una pentru pasageri, servere backend multiple (autentificare, hartă & direcții, procesare plăți, notificări șoferi etc.), baze de date diferite (relațional pentru tranzacții financiare, NoSQL pentru locații și tracking în timp real). Python ar putea fi folosit pentru a construi microserviciul care calculează potrivirile între cererea pasagerilor și șoferii disponibili: serviciul primește evenimente (cerere nouă de cursă, șofer X devine

disponibil), le procesează (calculează distanțe, caută în baza de date pozițiile șoferilor – posibil stocate într-un Redis sau Mongo pentru viteză), apoi trimite notificări.

Un astfel de microserviciu scris în Python poate folosi biblioteci de calcul (ex. geopy pentru distanțe), poate interoga un NoSQL (Redis) pentru localizări în timp real și poate comunica rezultatele către alte părți ale sistemului (prin mesaje sau apeluri API). Întregul ansamblu funcționează distribuit: multe instanțe ale serviciului de potrivire rulând în paralel, multe instanțe de gateway API care primesc cererile rest ale aplicațiilor mobile, o coadă de mesaje pe care se publică evenimente, etc. Python asigură un timp de dezvoltare rapid – esențial într-un mediu startup – și are performanță suficientă combinată cu componentele scalabile (de exemplu, implementările de baze de date și cache în C++).

Concluzii la 8.3: Python s-a impus ca un element central în dezvoltarea aplicațiilor distribuite moderne datorită **productivității ridicate** și a multitudinii de biblioteci care simplifică construirea de rețele, servicii web și integrarea cu baze de date distribuite. Fie că e vorba de un **microserviciu REST** care servește mii de cereri pe secundă, fie de un **script orchestral** care leagă componente disparate (de ex. extrage date dintr-un API, le pune într-o coadă, apoi într-o bază NoSQL pentru analytics), Python oferă instrumentele necesare. Firește, pentru aplicații distribuite la scară foarte mare, trebuie avute în vedere și optimizările infrastructurii (caching, load balancing, monitorizare). Majoritatea problemelor pot fi abordate incremental: de la un prototip Python care rulează pe un server, până la un cluster complet containerizat – același cod Python poate evolua pe măsură ce arhitectura devine mai complexă.

Bibliografie

[1] Tutorialul de Python — Python 3.15.0a0 documentație
https://docs.python.org/ro/dev/tutorial/index.html
[2] [3] [28] [29] sqlite3 — DB-API 2.0 interface for SQLite databases — Python 3.13.7 documentation
https://docs.python.org/3/library/sqlite3.html
[4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [32] [33] SQLAlchemy Tutorial: Practical Examples for Python Coders
https://www.pingcap.com/article/sqlalchemy-tutorial-practical-examples-for-python-coders/
[14] [15] [16] [17] [18] [19] [20] [34] Python and MongoDB: Connecting to NoSQL Databases – Real Python
https://realpython.com/introduction-to-mongodb-and-python/
[21] [22] [26] [27] What Is a Distributed Application? Computing System Examples
https://www.couchbase.com/blog/distributed-applications/
[23] [24] Client–server model - Wikipedia
https://en.wikipedia.org/wiki/Client%E2%80%93server_model
[25] Introduction of 3-Tier Architecture in DBMS - GeeksforGeeks
https://www.geeksforgeeks.org/dbms/introduction-of-3-tier-architecture-in-dbms-set-2/
[30] [31] SQLAlchemy ORM Tutorial for Python Developers
https://auth0.com/blog/sqlalchemy-orm-tutorial-for-python-developers/