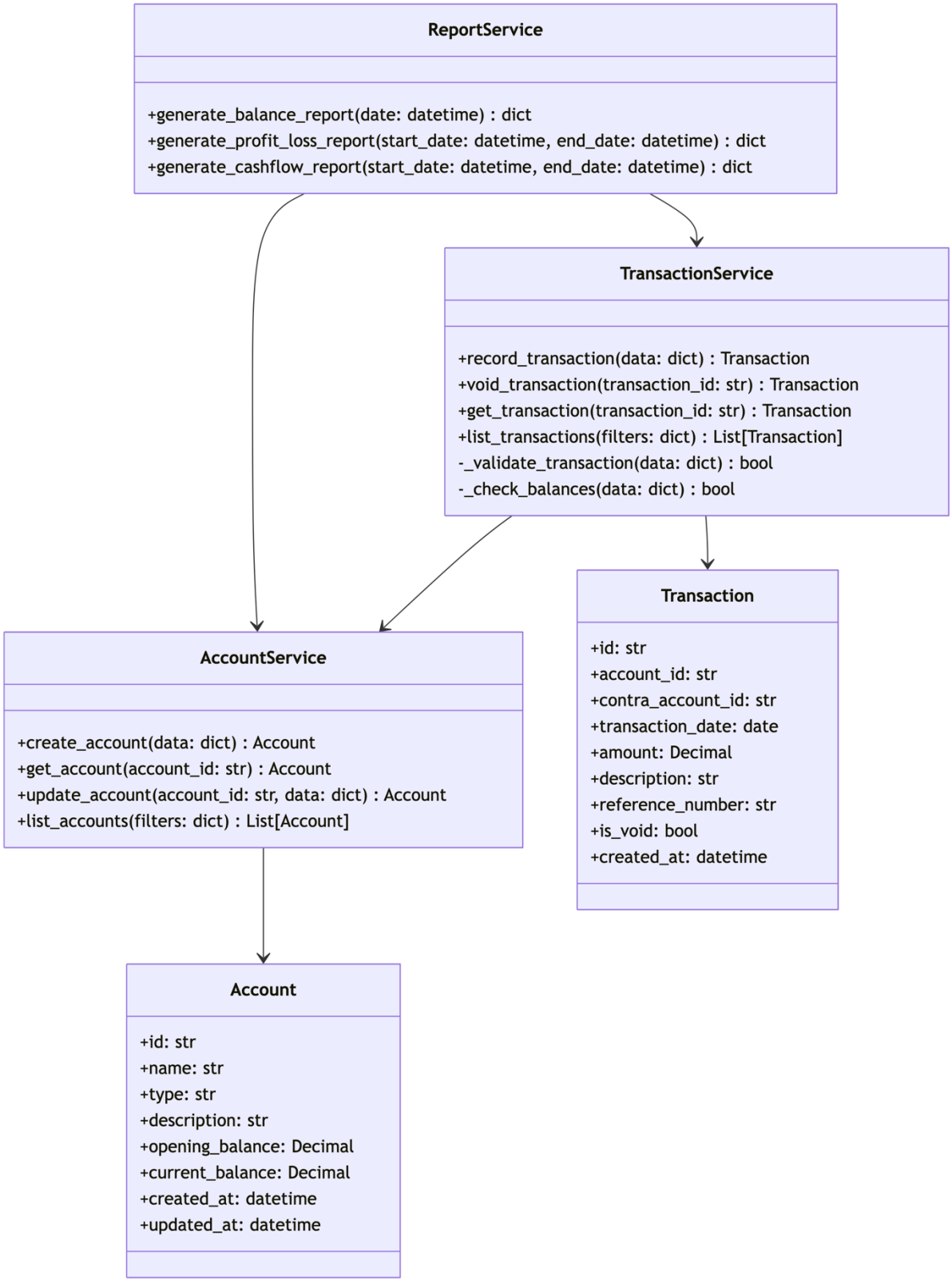
Low-Level Design (LLD) for Apartment Complex Account Management System (JavaScript/Python Stack)

1. Class Diagrams (Python Backend)

Core Class Structure

```
classDiagram
    class AccountService {
        +create_account(data: dict) Account
        +get_account(account_id: str) Account
        +update_account(account_id: str, data: dict) Account
        +list_accounts(filters: dict) List[Account]
    }

    class TransactionService {
        +record_transaction(data: dict) Transaction
        +void_transaction(transaction_id: str) Transaction
        +get_transaction(transaction_id: str) Transaction
        +list_transactions(filters: dict) List[Transaction]
        -_validate_transaction(data: dict) bool
        -_check_balances(data: dict) bool
    }

    class ReportService {
        +generate_balance_report(date: datetime) dict
        +generate_profit_loss_report(start_date: datetime, end_date: datetime)
dict
        +generate_cashflow_report(start_date: datetime, end_date: datetime) di
ct
    }

    class Account {
        +id: str
        +name: str
        +type: str
        +description: str
        +opening_balance: Decimal
        +current_balance: Decimal
        +created_at: datetime
        +updated_at: datetime
    }

    class Transaction {
        +id: str
        +account_id: str
        +contra_account_id: str
        +transaction_date: date
        +amount: Decimal
        +description: str
        +reference_number: str
        +is_void: bool
        +created_at: datetime
    }
```

```
AccountService --> Account
TransactionService --> Transaction
TransactionService --> AccountService
ReportService --> AccountService
ReportService --> TransactionService
```



Design Patterns Applied

1. **Repository Pattern** for data access:

python
```python
class AccountRepository:
    def __init__(self, db_session):
        self.db = db_session

    def create(self, account_data):
        account = Account(**account_data)
        self.db.add(account)
        self.db.commit()
        return account
```

2. **Dependency Injection** for services:

python
```python
class TransactionService:
    def __init__(self, account_repo, transaction_repo):
        self.account_repo = account_repo
        self.transaction_repo = transaction_repo

    def record_transaction(self, transaction_data):
        # Implementation using injected repos
```

2. Database Schema (PostgreSQL)

Detailed Table Definitions

2. Database Schema (PostgreSQL)

Detailed Table Definitions

sql
```sql
CREATE TABLE accounts (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name VARCHAR(255) NOT NULL,
    type VARCHAR(50) NOT NULL CHECK (type IN ('INCOME', 'EXPENSE', 'ASSET'
, 'LIABILITY')),
    description TEXT,
    opening_balance DECIMAL(19,4) NOT NULL,
    current_balance DECIMAL(19,4) NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT CURRENT_TIMESTAMP
,
    updated_at TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

```sql
CREATE TABLE transactions (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    account_id UUID NOT NULL REFERENCES accounts(id),
    contra_account_id UUID NOT NULL REFERENCES accounts(id),
    transaction_date DATE NOT NULL,
    amount DECIMAL(19,4) NOT NULL CHECK (amount > 0),
    description VARCHAR(500),
    reference_number VARCHAR(100),
    is_void BOOLEAN NOT NULL DEFAULT FALSE,
    created_at TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT CURRENT_TIMESTAMP
,
    CHECK (account_id != contra_account_id)
);

CREATE TABLE balance_history (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    account_id UUID NOT NULL REFERENCES accounts(id),
    balance_date DATE NOT NULL,
    balance DECIMAL(19,4) NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT CURRENT_TIMESTAMP
,
    UNIQUE(account_id, balance_date)
);
```

Indexes

sql

```sql
CREATE INDEX idx_transactions_account_id ON transactions(account_id);
CREATE INDEX idx_transactions_contra_account_id ON transactions(contra_acc
ount_id);
CREATE INDEX idx_transactions_date ON transactions(transaction_date);
CREATE INDEX idx_transactions_void_status ON transactions(is_void) WHERE i
s_void = TRUE;
CREATE INDEX idx_balance_history_account_date ON balance_history(account_i
d, balance_date);
```

3. API Contracts (FastAPI Backend)

Account Management

Create Account

- **Endpoint**: `POST /api/accounts`
- **Headers**:

    - `Authorization: Bearer <JWT>`
    - `Content-Type: application/json`
- **Request Body**:

json
```
{
  "name": "Rent Income",
  "type": "INCOME",
  "description": "Monthly rental income",
  "opening_balance": 0.00
}
```

- **Success Response (201 Created)**:

json
```
{
  "id": "550e8400-e29b-41d4-a716-446655440000",
  "name": "Rent Income",
  "type": "INCOME",
  "current_balance": 0.00,
  "created_at": "2023-10-15T14:30:00Z"
}
```

Transaction Management

Record Transaction

- **Endpoint**: `POST /api/transactions`
- **Headers**:
    - `Authorization: Bearer <JWT>`
    - `Content-Type: application/json`
- **Request Body**:

json
```
{
  "account_id": "550e8400-e29b-41d4-a716-446655440000",
  "contra_account_id": "550e8400-e29b-41d4-a716-446655440001",
  "transaction_date": "2023-10-15",
  "amount": 1200.00,
  "description": "October rent payment",
  "reference_number": "RENT-1023"
}
```

- **Success Response (201 Created)**:

json
```
{
  "id": "660e8400-e29b-41d4-a716-446655440000",
```

```
  "account_id": "550e8400-e29b-41d4-a716-446655440000",
  "contra_account_id": "550e8400-e29b-41d4-a716-446655440001",
  "amount": 1200.00,
  "new_balances": {
    "account": 1200.00,
    "contra_account": -1200.00
  }
}
```

4. Algorithm & Logic Specifications

Transaction Processing Algorithm (Python)

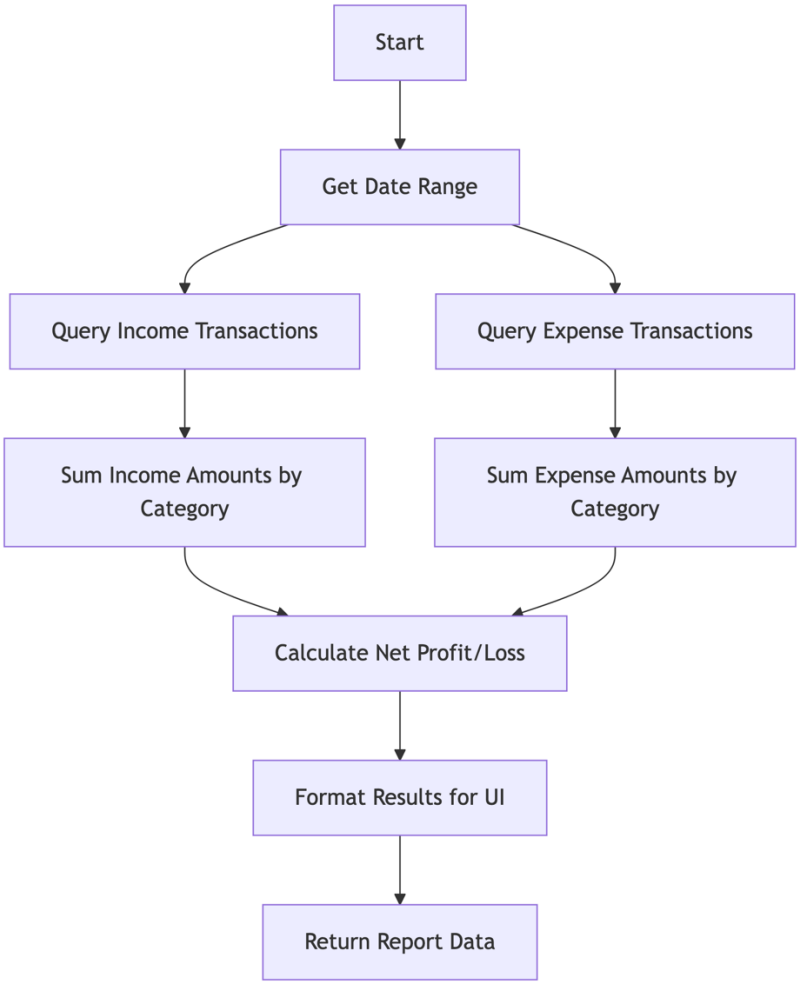
python

```python
async def process_transaction(transaction_data: dict):
    # Validate input
    if transaction_data['amount'] <= 0:
        raise HTTPException(status_code=400, detail="Amount must be positi
ve")

    if transaction_data['account_id'] == transaction_data['contra_account_
id']:
        raise HTTPException(status_code=400, detail="Account and contra ac
count must be different")

    # Get accounts with row-level locking
    async with db.begin():
        account = await get_account_with_lock(transaction_data['account_id
'])
        contra_account = await get_account_with_lock(transaction_data['con
tra_account_id'])

        # Check sufficient funds
        if account.type in ['ASSET', 'LIABILITY']:
            if account.current_balance + transaction_data['amount'] < 0:
                raise HTTPException(status_code=402, detail="Insufficient
funds")

        # Create transaction record
        transaction = await create_transaction_record(transaction_data)

        # Update balances
        account.current_balance += transaction_data['amount']
        contra_account.current_balance -= transaction_data['amount']

        await db.commit()
```

```
        return {
            "transaction": transaction,
            "new_balances": {
                "account": account.current_balance,
                "contra_account": contra_account.current_balance
            }
        }
```

Profit/Loss Calculation Flowchart

```
flowchart TD
    A[Start] --> B[Get Date Range]
    B --> C[Query Income Transactions]
    B --> D[Query Expense Transactions]
    C --> E[Sum Income Amounts by Category]
    D --> F[Sum Expense Amounts by Category]
    E --> G[Calculate Net Profit/Loss]
    F --> G
    G --> H[Format Results for UI]
    H --> I[Return Report Data]
```



5. Error Handling & Logging

Custom Exception Hierarchy (Python)

python
```python
class FinancialSystemError(Exception):
    """Base exception class"""
    status_code = 500

class ValidationError(FinancialSystemError):
    status_code = 400

class InsufficientFundsError(FinancialSystemError):
    status_code = 402

class NotFoundError(FinancialSystemError):
    status_code = 404

class AuthorizationError(FinancialSystemError):
    status_code = 403
```

Structured Logging Configuration

python
```python
# logging_config.py
import logging
from pythonjsonlogger import jsonlogger

def setup_logging():
    logger = logging.getLogger()
    logger.setLevel(logging.INFO)

    handler = logging.StreamHandler()
    formatter = jsonlogger.JsonFormatter(
        '%(asctime)s %(levelname)s %(name)s %(message)s'
    )
    handler.setFormatter(formatter)
    logger.addHandler(handler)
```

6. Security Controls

Input Validation (Python with Pydantic)

python
```python
from pydantic import BaseModel, Field, validator

class TransactionCreate(BaseModel):
    account_id: str = Field(..., min_length=36, max_length=36)
    contra_account_id: str = Field(..., min_length=36, max_length=36)
```

```
    transaction_date: date
    amount: Decimal = Field(..., gt=0)
    description: str = Field(None, max_length=500)
    reference_number: str = Field(None, max_length=100)

    @validator('contra_account_id')
    def accounts_must_differ(cls, v, values):
        if 'account_id' in values and v == values['account_id']:
            raise ValueError("Account and contra account must be different
")
        return v
```

JWT Authentication (FastAPI)

python

```python
from fastapi.security import OAuth2PasswordBearer
from jose import JWTError, jwt

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")

async def get_current_user(token: str = Depends(oauth2_scheme)):
    credentials_exception = HTTPException(
        status_code=401,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username: str = payload.get("sub")
        if username is None:
            raise credentials_exception
    except JWTError:
        raise credentials_exception

    user = get_user(username)
    if user is None:
        raise credentials_exception
    return user
```

7. Performance Optimizations

Caching with Redis (Python)

python

```python
from redis import asyncio as aioredis
from fastapi_cache import FastAPICache
```

```python
from fastapi_cache.backends.redis import RedisBackend
from fastapi_cache.decorator import cache

async def startup():
    redis = aioredis.from_url("redis://localhost")
    FastAPICache.init(RedisBackend(redis), prefix="fastapi-cache")

@router.get("/accounts/{account_id}")
@cache(expire=300)
async def get_account(account_id: str):
    return await AccountService.get_account(account_id)
```

Database Optimization (SQLAlchemy)

python

```python
# Optimized query for transaction history
async def get_transaction_history(account_id: str, start_date: date, end_d
ate: date):
    return await db.execute(
        select(Transaction)
        .where(Transaction.account_id == account_id)
        .where(Transaction.transaction_date.between(start_date, end_date))
        .where(Transaction.is_void == False)
        .order_by(Transaction.transaction_date.desc())
        .limit(100)
    )
```

8. Integration Details

Bank API Integration (Python)

python

```python
import httpx

async def fetch_bank_statements(account_number: str, from_date: date, to_d
ate: date):
    async with httpx.AsyncClient() as client:
        response = await client.get(
            f"{BANK_API_URL}/statements",
            params={
                "account": account_number,
                "from": from_date.isoformat(),
                "to": to_date.isoformat()
            },
            headers={"Authorization": f"Bearer {BANK_API_KEY}"},
            timeout=30.0
```

```
        )
        response.raise_for_status()
        return response.json()
```

9. Testing Guidance

Unit Test Examples (Python with pytest)

python

```python
@pytest.mark.asyncio
async def test_transaction_with_insufficient_funds():
    asset_account = Account(
        name="Test Asset",
        type="ASSET",
        opening_balance=100.00,
        current_balance=100.00
    )

    expense_account = Account(
        name="Test Expense",
        type="EXPENSE",
        opening_balance=0.00,
        current_balance=0.00
    )

    transaction_data = {
        "account_id": str(asset_account.id),
        "contra_account_id": str(expense_account.id),
        "transaction_date": date.today(),
        "amount": Decimal("150.00"),
        "description": "Test transaction"
    }

    with pytest.raises(HTTPException) as exc_info:
        await TransactionService().record_transaction(transaction_data)

    assert exc_info.value.status_code == 402
    assert "Insufficient funds" in exc_info.value.detail
```

10. Deployment Considerations

Docker Configuration

dockerfile

```
# backend/Dockerfile
```

```
FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

dockerfile

```
# frontend/Dockerfile
FROM node:16-alpine

WORKDIR /app

COPY package*.json ./
RUN npm install

COPY . .

RUN npm run build

CMD ["npm", "start"]
```

Kubernetes Deployment

yaml

```
# k8s/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
spec:
  replicas: 3
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
      - name: backend
        image: backend:latest
```

```
        ports:
        - containerPort: 8000
        envFrom:
        - configMapRef:
            name: backend-config
        resources:
          requests:
            cpu: "100m"
            memory: "256Mi"
          limits:
            cpu: "500m"
            memory: "512Mi"
```

This LLD provides a comprehensive technical blueprint for implementing the system using JavaScript for the frontend and Python for the backend, with detailed specifications for each component of the architecture.