

# Investment Tracker: Guía Completa para Pensar como Senior Developer

## Índice

1. La Mentalidad Senior
  2. Visión General del Proyecto
  3. Arquitectura y Por Qué Importa
  4. Estructura de Carpetas Explicada
  5. Capas de la Aplicación
  6. Patrones de Diseño Utilizados
  7. Flujo de Datos Completo
  8. Cada Módulo en Detalle
  9. Decisiones Técnicas y Sus Razones
  10. Cómo Escalaría Esto en el Mundo Real
  11. Debugging y Mantenimiento
  12. Conceptos Clave para Entrevistas
- 

## 1. La Mentalidad Senior

### ¿Qué diferencia a un junior de un senior?

Junior	Senior
"¿Cómo hago que funcione?"	"¿Cómo hago que sea mantenible, testeable y escalable?"
Escribe código	Diseña sistemas
Resuelve el problema inmediato	Anticipa problemas futuros
Copia soluciones	Entiende trade-offs
"Ya funciona"	"¿Qué pasa si...?"

### Principios que guían el pensamiento senior:

#### 1. KISS (Keep It Simple, Stupid)

- La solución más simple que funcione es generalmente la mejor
- Complejidad = bugs + dificultad de mantenimiento

## 2. DRY (Don't Repeat Yourself)

- Si escribes el mismo código dos veces, probablemente debería ser una función
- Pero cuidado: DRY llevado al extremo también es malo (abstracción prematura)

## 3. YAGNI (You Ain't Gonna Need It)

- No construyas funcionalidades "por si acaso"
- Construye lo que necesitas HOY, refactoriza cuando sea necesario

## 4. Separation of Concerns

- Cada módulo/clase/función debe tener UNA responsabilidad
- Si describes lo que hace y usas "y", probablemente hace demasiado

## 5. Fail Fast

- Detecta errores lo antes posible
  - Valida inputs, usa tipos, escribe tests
- 

## 2. Visión General del Proyecto

### ¿Qué es Investment Tracker?

Es un sistema de gestión de carteras de inversión personal que permite:

- Registrar operaciones (compras, ventas, dividendos, traspasos)
- Calcular rentabilidades y plusvalías
- Generar informes fiscales
- Comparar rendimiento con benchmarks

### ¿Por qué este stack tecnológico?

Python 3.10+



SQLite + SQLAlchemy (persistencia)

Pandas + NumPy (cálculos)

↓  
Streamlit + Plotly (interfaz)

## Decisión: ¿Por qué Python?

- Excelente para análisis de datos (pandas, numpy)
- Rápido de desarrollar (prototipado)
- Gran ecosistema financiero (yfinance, etc.)
- Tu background ya es Python

## Decisión: ¿Por qué SQLite?

- Sin servidor (un solo archivo .db)
- Perfecto para uso personal
- Portable y fácil de hacer backup
- Suficiente para miles de transacciones

## Alternativa descartada: PostgreSQL

- Requiere servidor
- Overkill para uso personal
- Añade complejidad de deployment

## Decisión: ¿Por qué SQLAlchemy?

- ORM = trabaja con objetos Python, no SQL crudo
- Abstacta la base de datos (podrías cambiar a PostgreSQL sin reescribir código)
- Previene SQL injection automáticamente
- Migraciones más fáciles

## Decisión: ¿Por qué Streamlit?

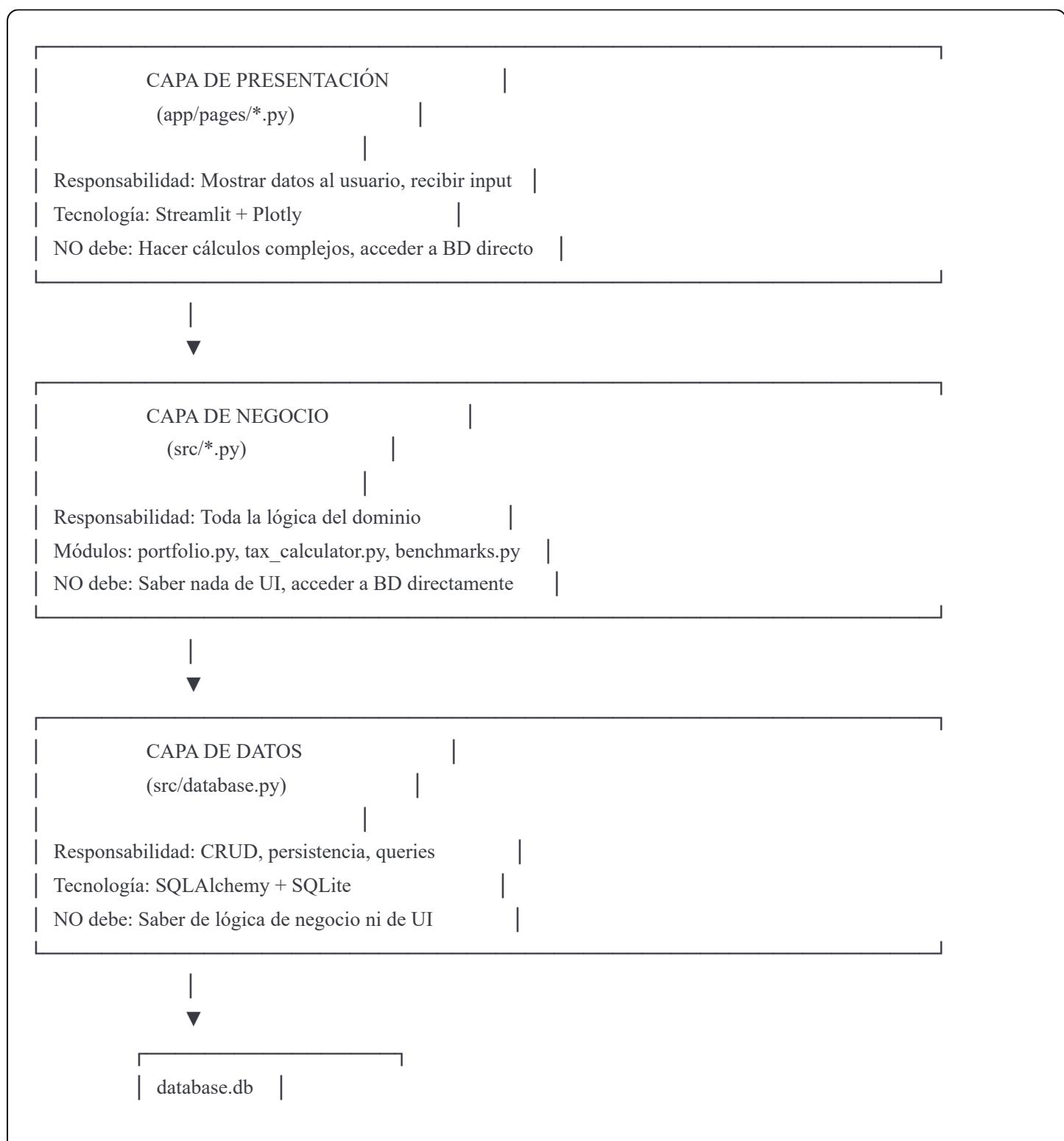
- Cero HTML/CSS/JavaScript
- UI automática desde Python
- Perfecto para dashboards de datos
- Recarga en vivo durante desarrollo

## Alternativas descartadas:

- Flask/Django: Requieren frontend separado
  - Dash: Más complejo, menos intuitivo
  - Jupyter: No es una app "real"
- 

## 3. Arquitectura y Por Qué Importa

### Arquitectura de 3 Capas



## ¿Por qué separar en capas?

### 1. Testabilidad

python

```
# MALO: Todo mezclado
def show_portfolio_page():
    conn = sqlite3.connect('db.db')
    cursor = conn.execute("SELECT * FROM transactions")
    total = 0
    for row in cursor:
        total += row[3] * row[4] # ¿Qué es row[3]? ¿row[4]?
    st.metric("Total", total)
```

# BUENO: Separado

```
def show_portfolio_page():
    portfolio = Portfolio() # Capa de negocio
    total = portfolio.get_total_value() # Método claro
    st.metric("Total", total) # Solo UI
```

Con separación, puedes testear `Portfolio.get_total_value()` sin necesitar Streamlit.

### 2. Reutilización

python

```
# El mismo módulo Portfolio se puede usar desde:
# - Streamlit (app/pages/)
# - Jupyter notebooks (notebooks/)
# - Scripts CLI (scripts/)
# - Tests (tests/)
# - Futura API REST
```

**3. Mantenibilidad** Si cambia la UI (de Streamlit a Flask), solo tocas `app/`. Si cambia la BD (de SQLite a PostgreSQL), solo tocas `database.py`.

### 4. Estructura de Carpetas Explicada

investment\_tracker/

```
src/          # CAPA DE NEGOCIO + DATOS
    # "El cerebro" de la aplicación

        __init__.py      # Hace de src/ un paquete Python

        database.py      # CAPA DE DATOS
            # - Modelos SQLAlchemy (tablas)
            # - Clase Database (CRUD)
            # - Única puerta a la BD

        portfolio.py     # CAPA DE NEGOCIO
            # - Cálculos de cartera
            # - Posiciones actuales
            # - Rentabilidades

        tax_calculator.py # CAPA DE NEGOCIO
            # - Cálculos fiscales
            # - FIFO/LIFO
            # - Plusvalías

        dividends.py      # CAPA DE NEGOCIO
            # - Gestión dividendos
            # - Yield, calendario

        benchmarks.py     # CAPA DE NEGOCIO
            # - Comparación con índices
            # - Métricas de riesgo

        market_data.py    # CAPA DE NEGOCIO + EXTERNA
            # - Descarga precios Yahoo
            # - Cache de precios

        data_loader.py    # UTILIDAD
            # - Importar/exportar CSV

        utils.py          # UTILIDADES COMUNES
            # - Funciones helper

app/          # CAPA DE PRESENTACIÓN
    # "La cara" de la aplicación

        main.py          # Punto de entrada Streamlit

        pages/           # Páginas de la app
            1_dashboard.py
            2_Añadir_Operación.py
            3_Análisis.py
```

```

|   |   └── 4_土豪_Fiscal.py
|   └── 5_分红_Dividendos.py
|   └── 6_跑分_Benchmarks.py
|
└── components/      # Componentes reutilizables
    ├── charts.py      # Funciones de gráficos
    ├── tables.py      # Funciones de tablas
    └── forms.py       # Formularios
|
└── data/            # ALMACENAMIENTO
    ├── database.db    # Base de datos SQLite
    ├── benchmark_data/ # CSVs de índices
    └── exports/        # Informes generados
|
└── tests/           # TESTS
    ├── test_portfolio.py
    ├── test_tax_calculator.py
    └── ...
|
└── notebooks/       # EXPLORACIÓN
    └── *.ipynb        # Jupyter notebooks
|
└── scripts/         # UTILIDADES STANDALONE
    └── *.py           # Scripts ejecutables
|
├── config.py        # Configuración global
├── requirements.txt  # Dependencias
└── README.md        # Documentación

```

## ¿Por qué esta estructura?

**Convención sobre configuración:** Esta estructura es un estándar de facto en Python. Cualquier desarrollador que vea esto sabrá inmediatamente dónde está cada cosa.

**Separación física = separación lógica:**

- `src/` = lógica que no depende de UI
- `app/` = todo lo relacionado con Streamlit
- `data/` = archivos que no son código
- `tests/` = pruebas separadas del código de producción

## 5. Capas de la Aplicación

**Capa de Datos:** `src/database.py`

**Responsabilidad única:** Interactuar con SQLite.

```
python
```

```

# ESTRUCTURA BÁSICA

# 1. MODELOS (tablas de la BD)

class Transaction(Base):
    __tablename__ = 'transactions'
    id = Column(Integer, primary_key=True)
    date = Column(Date, nullable=False)
    type = Column(String(20), nullable=False) # buy, sell, dividend...
    ticker = Column(String(50), nullable=False)
    quantity = Column(Float, nullable=False)
    price = Column(Float, nullable=False)
    # ... más campos

class Dividend(Base):
    __tablename__ = 'dividends'
    # ... campos

class BenchmarkData(Base):
    __tablename__ = 'benchmark_data'
    # ... campos

class AssetPrice(Base):
    __tablename__ = 'asset_prices'
    # ... campos

# 2. CLASE DATABASE (operaciones CRUD)

class Database:
    def __init__(self, db_path=None):
        # Configurar conexión
        self.engine = create_engine(f'sqlite:///{{db_path}}')
        self.session = Session(self.engine)
        # Crear tablas si no existen
        Base.metadata.create_all(self.engine)

    # CREATE
    def add_transaction(self, data: dict) -> int:
        transaction = Transaction(**data)
        self.session.add(transaction)
        self.session.commit()
        return transaction.id

    # READ
    def get_transactions(self, **filters) -> List[Transaction]:
        query = self.session.query(Transaction)
        # Aplicar filtros...

```

```
return query.all()

# UPDATE
def update_transaction(self, id: int, data: dict) -> bool:
    # ...

# DELETE
def delete_transaction(self, id: int) -> bool:
    # ...

def close(self):
    self.session.close()
```

## Patrón utilizado: Repository Pattern

- La clase `Database` es el único punto de acceso a los datos
- El resto de la app no sabe que existe SQLite
- Podrías cambiar a PostgreSQL y solo tocarías este archivo

## Capa de Negocio: `(src/portfolio.py)`, etc.

**Responsabilidad:** Implementar la lógica del dominio (finanzas).

python

```

class Portfolio:
    def __init__(self, db_path=None):
        # Usa la capa de datos, no accede a SQLite directamente
        self.db = Database(db_path)

    def get_current_positions(self, current_prices=None):
        """
        LÓGICA DE NEGOCIO:
        - Obtener transacciones
        - Calcular posiciones usando FIFO
        - Calcular plusvalías latentes
        - Retornar DataFrame estructurado
        """

        transactions = self.db.get_transactions() # Usa capa de datos

        # FIFO: First In, First Out
        positions = {}
        for t in transactions:
            if t.type == 'buy':
                # Añadir al pool de lotes
            elif t.type == 'sell':
                # Reducir lotes más antiguos primero

        # Calcular valores de mercado
        for ticker, pos in positions.items():
            if current_prices and ticker in current_prices:
                market_value = pos['quantity'] * current_prices[ticker]
            # ...

        return pd.DataFrame(...)

    def get_total_value(self):
        positions = self.get_current_positions()
        return positions['market_value'].sum()

```

## ¿Por qué esta separación?

- `Portfolio` no sabe nada de SQL
- `Portfolio` no sabe nada de Streamlit
- Solo sabe de lógica financiera
- Fácil de testear con datos mock

## Capa de Presentación: `app/pages/*.py`

**Responsabilidad:** Mostrar datos y recibir input del usuario.

```
python
```

```
# app/pages/1_Dashboard.py

import streamlit as st
from src.portfolio import Portfolio # Usa capa de negocio
from src.database import Database # Solo para precios

# CONFIGURACIÓN DE PÁGINA
st.set_page_config(page_title="Dashboard", layout="wide")
st.title("Dashboard")

# OBTENER DATOS (delegando a capa de negocio)
portfolio = Portfolio()
db = Database()
current_prices = db.get_all_latest_prices()
positions = portfolio.get_current_positions(current_prices=current_prices)

# MOSTRAR DATOS (solo UI)
col1, col2, col3 = st.columns(3)

with col1:
    st.metric("Valor Total", f"{positions['market_value'].sum():,.2f}€")

with col2:
    st.metric("Coste", f"{positions['cost_basis'].sum():,.2f}€")

# GRÁFICOS
fig = plot_allocation_donut(positions) # Componente reutilizable
st.plotly_chart(fig)

# TABLAS
st.dataframe(positions)
```

**¿Qué NO debe hacer la capa de presentación?**

```
python
```

```
# ❌ MALO: Lógica de negocio en la UI
```

```
total = 0
```

```
for _, row in transactions.iterrows():
    if row['type'] == 'buy':
        total += row['quantity'] * row['price']
    elif row['type'] == 'sell':
        total -= row['quantity'] * row['price']
st.metric("Total", total)
```

```
# ✅ BUENO: Delegar a capa de negocio
```

```
total = portfolio.get_total_value()
st.metric("Total", total)
```

## 6. Patrones de Diseño Utilizados

### 1. Repository Pattern (database.py)

**Problema:** El código de acceso a datos está disperso por toda la app. **Solución:** Centralizar todo el acceso a datos en una clase.

```
python
```

```

# SIN patrón: Acceso directo por todas partes
# portfolio.py
conn = sqlite3.connect('db.db')
cursor.execute("SELECT * FROM transactions WHERE type='buy'")

# tax_calculator.py
conn = sqlite3.connect('db.db')
cursor.execute("SELECT * FROM transactions WHERE type='sell'")

# CON patrón: Acceso centralizado
# database.py
class Database:
    def get_transactions(self, type=None):
        query = self.session.query(Transaction)
        if type:
            query = query.filter(Transaction.type == type)
        return query.all()

# portfolio.py
db = Database()
buys = db.get_transactions(type='buy')

# tax_calculator.py
db = Database()
sells = db.get_transactions(type='sell')

```

## Beneficios:

- Un solo lugar para cambiar queries
- Fácil de mockear en tests
- Consistencia en el acceso a datos

## 2. Factory Methods (múltiples módulos)

**Problema:** Crear objetos complejos con muchas configuraciones. **Solución:** Métodos que crean objetos preconfigurados.

python

```
# database.py

def transactions_to_dataframe(self, transactions):
    """Factory method: convierte objetos a DataFrame"""
    return pd.DataFrame([t.to_dict() for t in transactions])

# Uso
df = db.transactions_to_dataframe(db.get_transactions())
```

### 3. Strategy Pattern (tax\_calculator.py)

**Problema:** Diferentes algoritmos para el mismo propósito (FIFO vs LIFO). **Solución:** Encapsular cada algoritmo y hacerlos intercambiables.

```
python

class TaxCalculator:
    def __init__(self, method='FIFO'): # Estrategia inyectada
        self.method = method

    def get_available_lots(self, ticker):
        lots = self._get_all_lots(ticker)

        if self.method == 'FIFO':
            # Ordenar del más antiguo al más reciente
            return sorted(lots, key=lambda x: x['date'])
        elif self.method == 'LIFO':
            # Ordenar del más reciente al más antiguo
            return sorted(lots, key=lambda x: x['date'], reverse=True)

# Uso
tax_fifo = TaxCalculator(method='FIFO')
tax_lifo = TaxCalculator(method='LIFO')

# Mismo método, diferente comportamiento
lots_fifo = tax_fifo.get_available_lots('AAPL')
lots_lifo = tax_lifo.get_available_lots('AAPL')
```

### 4. Facade Pattern (benchmarks.py)

**Problema:** Subsistema complejo con muchas clases y funciones. **Solución:** Interfaz simplificada que oculta la complejidad.

```
python
```

```
class BenchmarkComparator:
```

```
    """Facade que simplifica todas las operaciones de benchmark"""
```

```
def __init__(self):
```

```
    self.db = Database()      # Subsistema 1
```

```
    self.portfolio = Portfolio() # Subsistema 2
```

```
    # Configuración interna...
```

```
# Interfaz simple
```

```
def download_benchmark(self, name, start, end):
```

```
    """Oculta: yfinance, parsing, guardado en BD"""
```

```
    pass
```

```
def compare_to_benchmark(self, benchmark_name, start, end):
```

```
    """Oculta: obtener series, alinear fechas, normalizar"""
```

```
    pass
```

```
def calculate_risk_metrics(self, ...):
```

```
    """Oculta: volatilidad, beta, sharpe, sortino, VaR..."""
```

```
    pass
```

```
# Uso simple desde UI
```

```
bc = BenchmarkComparator()
```

```
bc.download_benchmark('SP500', '2024-01-01', '2025-01-01')
```

```
comparison = bc.compare_to_benchmark('SP500', '2024-01-01', '2025-01-01')
```

```
# ¡No necesito saber cómo funciona internamente!
```

## 5. Dependency Injection (en todas partes)

**Problema:** Clases acopladas difíciles de testear. **Solución:** Inyectar dependencias en lugar de crearlas internamente.

```
python
```

```

# ❌ ACOPLADO
class Portfolio:
    def __init__(self):
        self.db = Database() # Siempre usa Database real
        # Imposible testear sin BD real

# ✅ DESACOPLADO (inyección)
class Portfolio:
    def __init__(self, db_path=None):
        self.db = Database(db_path) if db_path else Database()
        # Puedo pasar un path de test

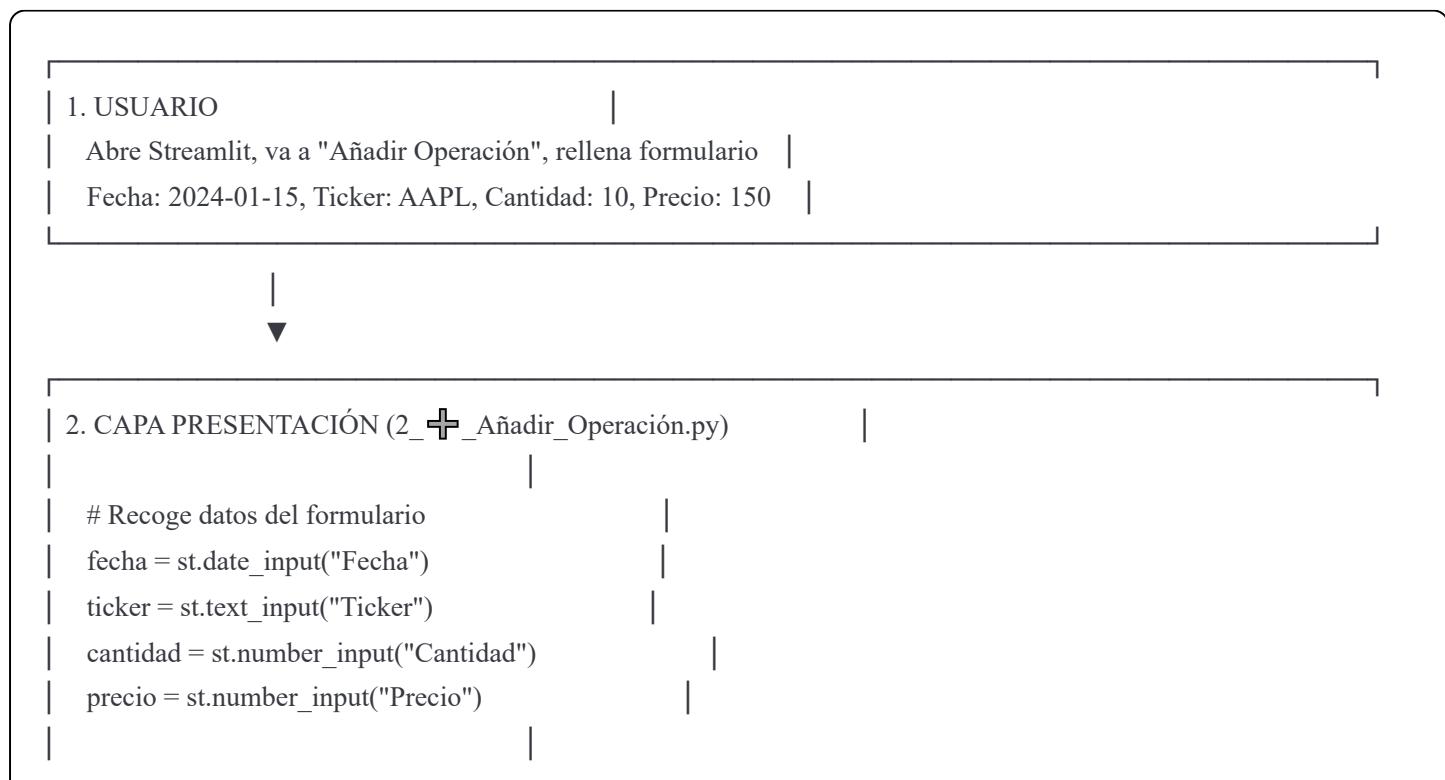
# Aún mejor (inyección completa)
class Portfolio:
    def __init__(self, database=None):
        self.db = database if database else Database()
        # Puedo pasar un mock completo

# En test
mock_db = MockDatabase() # Datos fake
portfolio = Portfolio(database=mock_db)
# Testeo sin BD real

```

## 7. Flujo de Datos Completo

### Ejemplo: Usuario registra una compra



```
| if st.button("Guardar"):
|     # Prepara datos
|     data = {
|         'date': fecha,
|         'type': 'buy',
|         'ticker': ticker,
|         'quantity': cantidad,
|         'price': precio,
|         'total': cantidad * precio
|     }
|     # Llama a capa de datos
|     db.add_transaction(data)
```

### 3. CAPA DATOS (database.py)

```
| def add_transaction(self, data):
|     # Validar datos
|     if data['quantity'] <= 0:
|         raise ValueError("Cantidad debe ser > 0")
|     # Crear objeto ORM
|     transaction = Transaction(
|         date=data['date'],
|         type=data['type'],
|         ticker=data['ticker'].upper(),
|         quantity=data['quantity'],
|         price=data['price'],
|         total=data['total']
|     )
|     # Persistir en BD
|     self.session.add(transaction)
|     self.session.commit()
|
|     return transaction.id
```

### 4. BASE DE DATOS (database.db)

```
| INSERT INTO transactions
| (date, type, ticker, quantity, price, total)
| VALUES ('2024-01-15', 'buy', 'AAPL', 10, 150, 1500)
```

## 5. FEEDBACK AL USUARIO

```
st.success("✅ Compra registrada correctamente")  
st.balloons()
```

## Ejemplo: Usuario ve el Dashboard

### 1. USUARIO

```
Abre Streamlit, navega a Dashboard
```

### 2. CAPA PRESENTACIÓN (1\_📊\_Dashboard.py)

```
# Inicializar capas inferiores  
portfolio = Portfolio()  
db = Database()  
  
# Obtener precios actuales  
current_prices = db.get_all_latest_prices()
```

### 3. CAPA DATOS

```
def get_all_latest_prices():  
    # Query para último precio de cada ticker  
    subq = self.session.query(  
        AssetPrice.ticker,  
        func.max(AssetPrice.date)  
    ).group_by(AssetPrice.ticker).subquery()  
    ...  
    return {ticker: price for ...}  
  
Retorna: {'AAPL': 175.50, 'GOOGL': 140.25, ...}
```

## 2. CAPA PRESENTACIÓN (continúa)

```
# Obtener posiciones con precios actuales  
positions = portfolio.get_current_positions()  
current_prices=current_prices  
)
```

## 4. CAPA NEGOCIO (portfolio.py)

```
def get_current_positions(self, current_prices=None):  
    # Obtener transacciones de capa de datos  
    transactions = self.db.get_transactions()  
  
    # LÓGICA DE NEGOCIO: Calcular posiciones FIFO  
    positions = {}  
    for t in transactions:  
        if t.type == 'buy':  
            # Añadir al pool...  
        elif t.type == 'sell':  
            # Reducir lotes FIFO...  
  
    # Calcular valores de mercado  
    for ticker, pos in positions.items():  
        if current_prices and ticker in current_prices:  
            pos['market_value'] = (  
                pos['quantity'] * current_prices[ticker]  
            )  
            pos['unrealized_gain'] = (  
                pos['market_value'] - pos['cost_basis']  
            )  
  
    return pd.DataFrame(positions)
```

Retorna: DataFrame con ticker, quantity, cost, value, gain

## 2. CAPA PRESENTACIÓN (final)

```
| # Mostrar métricas |  
| total_value = positions['market_value'].sum() |  
| total_cost = positions['cost_basis'].sum() |  
| gain = positions['unrealized_gain'].sum() |  
| |  
| col1, col2, col3 = st.columns(3) |  
| col1.metric("Valor Total", f'{total_value:.2f}€') |  
| col2.metric("Coste", f'{total_cost:.2f}€') |  
| col3.metric("Ganancia", f'{gain:.2f}€') |  
| |  
| # Mostrar gráfico |  
| fig = plot_allocation_donut(positions) |  
| st.plotly_chart(fig) |
```

## 8. Cada Módulo en Detalle

### src/database.py - La Fundación

**Propósito:** Única puerta de acceso a los datos persistentes.

**Componentes:**

```
python
```

```

# MODELOS (representan tablas)

class Transaction(Base):
    """Una operación: compra, venta, dividendo, traspaso"""
    id: int (PK)
    date: Date
    type: str # 'buy', 'sell', 'dividend', 'transfer_in', 'transfer_out'
    ticker: str
    name: str
    quantity: float
    price: float
    commission: float
    total: float
    currency: str
    asset_type: str # 'accion', 'fondo', 'etf'
    cost_basis_eur: float # Para traspasos
    realized_gain_eur: float # Para ventas
    notes: str

class Dividend(Base):
    """Un dividendo recibido"""
    id, ticker, date, gross_amount, net_amount, withholding_tax

class BenchmarkData(Base):
    """Datos históricos de un índice"""
    id, benchmark_name, date, value

class AssetPrice(Base):
    """Precio histórico de un activo de la cartera"""
    id, ticker, date, close_price, adj_close_price

# CLASE DATABASE (operaciones)

class Database:
    # Conexión
    def __init__(self, db_path)
    def close(self)

    # Transacciones CRUD
    def add_transaction(data) -> id
    def get_transactions(filters) -> List[Transaction]
    def update_transaction(id, data) -> bool
    def delete_transaction(id) -> bool
    def transactions_to_dataframe(trans) -> DataFrame

    # Dividendos CRUD
    def add_dividend(data) -> id
    def get_dividends(filters) -> List[Dividend]

```

```

# Benchmarks CRUD

def add_benchmark_data(name, date, value)
def get_benchmark_data(name, start, end) -> List[BenchmarkData]
def get_available_benchmarks() -> List[str]

# Precios de activos

def add_asset_price(ticker, date, price)
def get_asset_prices(ticker, start, end) -> List[AssetPrice]
def get_latest_price(ticker) -> float
def get_all_latest_prices() -> Dict[str, float]

```

## **src/portfolio.py** - El Cerebro Financiero

**Propósito:** Toda la lógica de cálculo de cartera.

python

```

class Portfolio:

    # Posiciones

    def get_current_positions(current_prices=None) -> DataFrame
        """
        Calcula posiciones actuales usando FIFO.
        Retorna: ticker, quantity, avg_price, cost_basis,
        market_value, unrealized_gain, unrealized_gain_pct
        """

```

```

    def get_position(ticker) -> Dict
        """
        Detalle de una posición específica"""

```

```

    # Valor de cartera
    def get_total_value() -> float
    def get_total_cost() -> float
    def get_unrealized_gains() -> float

```

```

    # Análisis
    def get_performance_by_asset() -> DataFrame
        """
        Rentabilidad individual de cada activo"""

```

```

    def get_allocation() -> DataFrame
        """
        Distribución porcentual de la cartera"""

```

```

    # Histórico
    def get_invested_capital_timeline(start, end) -> DataFrame
    def get_return_over_time(start, end) -> DataFrame

```

## **src/tax\_calculator.py - El Gestor Fiscal**

**Propósito:** Todo lo relacionado con impuestos y fiscalidad española.

python

```

class TaxCalculator:
    def __init__(self, method='FIFO'): # o 'LIFO'

        # Gestión de lotes
    def get_available_lots(ticker) -> List[Dict]
        """
        Lotes de compra disponibles para vender.
        Considera: compras, ventas previas, traspasos.
        """

    def get_all_available_lots() -> DataFrame
        """Todos los lotes de todos los tickers"""

        # Cálculo de plusvalías
    def calculate_sale_gain(ticker, quantity, sale_price) -> Dict
        """
        Calcula plusvalía de una venta usando FIFO/LIFO.
        Retorna: gain, lots_sold, cost_basis, sale_proceeds
        """

    # Informes fiscales
    def get_fiscal_year_summary(year) -> Dict
        """Resumen: ganancias, pérdidas, neto"""

    def get_fiscal_year_detail(year) -> DataFrame
        """Detalle de cada venta del año"""

    def calculate_tax(taxable_base) -> Dict
        """Calcula impuesto según tramos IRPF español"""

    # Utilidades
    def simulate_sale(ticker, quantity, price) -> Dict
        """Simular impacto fiscal antes de vender"""

    def check_wash_sale(ticker, sale_date) -> Dict
        """Verificar regla de los 2 meses"""

    def get_wash_sales_in_year(year) -> DataFrame
        """Todas las ventas afectadas por regla 2 meses"""

        # Exportación
    def export_fiscal_report(year, filepath)
        """Exportar informe fiscal a Excel"""

```

## **src/dividends.py** - Gestión de Dividendos

**Propósito:** Tracking y análisis de dividendos.

python

```
class DividendManager:  
    # CRUD  
    def add_dividend(ticker, gross, net, date, notes)  
    def get_dividends(ticker=None, year=None) -> List  
  
    # Análisis  
    def get_total_dividends(year=None) -> float  
    def get_dividends_by_ticker() -> DataFrame  
    def get_dividend_yield(ticker) -> float  
  
    # Calendario  
    def get_dividends_timeline(start, end) -> DataFrame  
    def get_upcoming_dividends() -> DataFrame # Si hay ex-dates  
  
    # Integración  
    def get_total_return_with_dividends(ticker) -> Dict  
    """capital_gain + dividend_yield = total_return"""
```

## **src/benchmarks.py** - Comparación con Índices

**Propósito:** Comparar cartera con índices de mercado.

python

```

class BenchmarkComparator:
    # Datos de benchmarks
    def download_benchmark(name, start, end) -> int
    """Descarga datos de Yahoo Finance"""

    def get_available_benchmarks() -> List[Dict]
    def get_benchmark_series(name, start, end) -> Series

    # Datos de cartera
    def get_portfolio_series(start, end) -> Series
    """Valor de la cartera por día"""

    # Comparación
    def compare_to_benchmark(name, start, end) -> DataFrame
    """
    Compara cartera vs benchmark.
    Retorna: date, portfolio_normalized, benchmark_normalized, outperformance
    """

    def calculate_returns(name, start, end) -> Dict
    """Rendimientos y outperformance"""

    # Métricas de riesgo
    def calculate_risk_metrics(name, start, end, risk_free_rate) -> Dict
    """
    Volatilidad, Beta, Alpha, Sharpe Ratio, Sortino Ratio,
    Max Drawdown, Calmar Ratio, Tracking Error, Information Ratio,
    Value at Risk (VaR)
    """

    # Normalización
    def normalize_to_base_100(series) -> Series

```

## src/market\_data.py - Precios de Mercado

**Propósito:** Descargar y gestionar precios reales de los activos.

python

```

class MarketDataManager:
    # Descarga
    def get_portfolio_tickers() -> List[Dict]
        """Tickers únicos de la cartera"""

    def download_ticker_prices(ticker, start, end) -> DataFrame
        """Descarga precios desde Yahoo Finance"""

    def download_portfolio_prices(start, end) -> Dict
        """Descarga precios de todos los activos"""

    # Obtención
    def get_ticker_prices(ticker, start, end) -> DataFrame
        """De cache, BD, o descarga"""

    # Valor de cartera
    def get_portfolio_market_value_series(start, end) -> DataFrame
        """
        Valor de mercado REAL por día.
        Columnas: date, market_value, cost_basis, unrealized_pnl,
        realized_pnl, total_value
        """

    def get_investing_style_data(start, end) -> DataFrame
        """
        Para gráfico estilo Investing.com.
        Columnas: open_positions_value, closed_positions_pnl,
        total_portfolio_value, invested_capital
        """

    def get_open_positions_only_series(start, end) -> DataFrame
        """Solo posiciones actualmente abiertas"""

    # Utilidades
    def get_download_status() -> DataFrame
        """Estado de precios descargados por ticker"""

    def clear_price_cache()

```

## 9. Decisiones Técnicas y Sus Razones

**¿Por qué FIFO por defecto?**

**Contexto:** Cuando vendes acciones, ¿cuáles vendes primero?

- **FIFO (First In, First Out):** Las más antiguas
- **LIFO (Last In, First Out):** Las más recientes

**Decisión:** FIFO por defecto.

**Razón:**

1. Es el método que asume Hacienda en España si no especificas otro
2. Generalmente más favorable fiscalmente (lotes antiguos suelen tener menor coste)
3. Más intuitivo para la mayoría de inversores

**¿Por qué SQLite y no CSV?**

**Decisión:** SQLite en lugar de archivos CSV.

**Razones:**

1. **ACID:** Transacciones atómicas (no se corrompen datos)
2. **Queries:** Filtrar/ordenar eficientemente
3. **Relaciones:** Vincular transacciones con dividendos
4. **Concurrencia:** Múltiples lecturas simultáneas
5. **Escalabilidad:** Funciona bien hasta millones de registros

**CSV sería problemático para:**

```
python
# ¿Cómo harías esto eficientemente en CSV?
"Dame las ventas de AAPL en 2024 ordenadas por fecha"
```

**¿Por qué Streamlit y no Flask?**

**Decisión:** Streamlit para la UI.

**Razones:**

1. **Zero frontend:** No necesitas HTML/CSS/JS
2. **Data-first:** Diseñado para dashboards de datos
3. **Rápido:** Prototipo funcional en horas
4. **Interactivo:** Widgets nativos
5. **Hot reload:** Cambios en vivo

## Flask sería mejor si:

- Necesitas API REST
- Quieres control total del frontend
- Múltiples usuarios concurrentes
- Autenticación compleja

## ¿Por qué ORM (SQLAlchemy) y no SQL crudo?

Decisión: Usar SQLAlchemy ORM.

```
python

#SQL CRUDO
cursor.execute("""
    SELECT * FROM transactions
    WHERE type = ? AND date >= ? AND date <= ?
    ORDER BY date ASC
""", ('buy', '2024-01-01', '2024-12-31'))
rows = cursor.fetchall()
# rows[0][3] = ??? (¿qué columna es?)

#ORM
transactions = session.query(Transaction).filter(
    Transaction.type == 'buy',
    Transaction.date >= '2024-01-01',
    Transaction.date <= '2024-12-31'
).order_by(Transaction.date.asc()).all()
# transaction.ticker = 'AAPL' (claro y tipado)
```

## Razones:

1. **Legibilidad:** Código más claro
2. **Seguridad:** Previene SQL injection
3. **Portabilidad:** Cambiar de SQLite a PostgreSQL es trivial
4. **Objetos:** Trabajas con Python, no strings SQL
5. **Validación:** Tipos y constraints automáticos

## ¿Por qué separar `market_data.py` de `benchmarks.py`?

Decisión: Dos módulos distintos para datos de mercado.

**benchmarks.py**: Datos de ÍNDICES (S&P 500, IBEX 35) **market\_data.py**: Datos de TUS ACTIVOS (AAPL, GOOGL)

## Razones:

1. **Responsabilidad única:** Cada módulo hace una cosa
  2. **Independencia:** Puedes usar market\_data sin benchmarks
  3. **Diferentes fuentes:** Índices vs acciones individuales
  4. **Diferentes usos:** Comparar vs valorar
- 

## 10. Cómo Escalaría Esto en el Mundo Real

### Nivel actual: App personal

1 usuario → SQLite → Streamlit local

### Nivel 2: App para varios usuarios (SaaS)

Cambios necesarios:

1. SQLite → PostgreSQL (concurrencia)
2. Autenticación (usuarios)
3. Multi-tenancy (datos separados por usuario)
4. Deployment (servidor)

python

```
# Cambio mínimo en database.py
# DE:
engine = create_engine('sqlite:///data/database.db')
# A:
engine = create_engine('postgresql://user:pass@host/db')

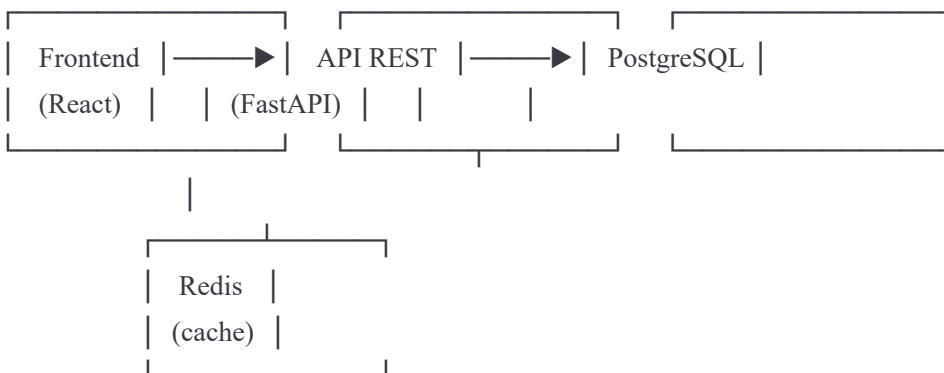
# El resto del código NO CAMBIA (gracias a SQLAlchemy)
```

### Nivel 3: App empresarial

Cambios adicionales:

1. Microservicios (separar backend)
2. API REST (Flask/FastAPI)
3. Frontend separado (React/Vue)

4. Cache (Redis)
5. Cola de tareas (Celery)
6. Monitorización (Prometheus)



## ¿Por qué nuestro código escalaría bien?

### Separación de capas:

- La UI (Streamlit) se reemplazaría por API + Frontend
- La lógica de negocio (portfolio.py, etc.) NO CAMBIA
- La capa de datos solo necesita cambiar connection string

### Sin lógica en la UI:

```
python

# NUESTRO CÓDIGO (bueno)
# Toda la lógica está en src/
# La UI solo llama métodos

# CÓDIGO MALO
# Lógica mezclada en la UI
# Habría que reescribir todo
```

## 11. Debugging y Mantenimiento

### Estrategias de debugging

#### 1. Logging (no implementado aún, pero debería)

```
python
```

```
import logging

logger = logging.getLogger(__name__)

def get_current_positions(self):
    logger.info("Calculando posiciones actuales")
    transactions = self.db.get_transactions()
    logger.debug(f"Obtenidas {len(transactions)} transacciones")

    # ...

    logger.info(f"Calculadas {len(positions)} posiciones")
    return positions
```

## 2. Usar el REPL de Python

```
python

# En terminal
python
>>> from src.portfolio import Portfolio
>>> p = Portfolio()
>>> positions = p.get_current_positions()
>>> positions.head() # Inspeccionar
```

## 3. Streamlit debugging

```
python

# En la UI
st.write("DEBUG:", variable) # Muestra cualquier cosa
st.dataframe(df) # Muestra DataFrames
st.json(dict_data) # Muestra dicts formateados
```

## 4. Tests unitarios

```
python
```

```
# tests/test_portfolio.py
def test_fifo_calculation():
    # Preparar datos de prueba
    mock_db = MockDatabase()
    mock_db.add_transaction({'type': 'buy', 'quantity': 100, 'price': 10})
    mock_db.add_transaction({'type': 'buy', 'quantity': 50, 'price': 15})
    mock_db.add_transaction({'type': 'sell', 'quantity': 80, 'price': 20})

    portfolio = Portfolio(database=mock_db)
    positions = portfolio.get_current_positions()

    # Verificar
    assert positions.loc[0, 'quantity'] == 70 # 100 + 50 - 80
    assert positions.loc[0, 'avg_price'] == 12.14 # Promedio ponderado
```

## Mantenimiento preventivo

### 1. Consistencia de código

```
bash

# Formateo automático
pip install black
black src/ # Formatea todo el código
```

### 2. Análisis estático

```
bash

# Detectar errores sin ejecutar
pip install flake8
flake8 src/ # Muestra problemas
```

### 3. Type hints (no implementado completamente)

```
python
```

```
# CON type hints (mejor)
def get_current_positions(
    self,
    current_prices: Optional[Dict[str, float]] = None
) -> pd.DataFrame:
    ...
    # Verificar tipos
    pip install mypy
    mypy src/ # Detecta errores de tipos
```

## 12. Conceptos Clave para Entrevistas

Preguntas que podrían hacerte:

### 1. "¿Por qué separaste en capas?"

Para separar responsabilidades, facilitar testing, y permitir cambios independientes. Por ejemplo, puedo cambiar de Streamlit a Flask sin tocar la lógica de negocio.

### 2. "¿Qué patrones de diseño usaste?"

Repository Pattern en database.py para centralizar acceso a datos. Strategy Pattern en tax\_calculator.py para FIFO/LIFO. Facade Pattern en benchmarks.py para simplificar operaciones complejas.

### 3. "¿Cómo manejas la persistencia de datos?"

SQLAlchemy ORM sobre SQLite. Los modelos definen el schema, la clase Database encapsula todas las operaciones CRUD. El resto de la app no sabe que existe SQL.

### 4. "¿Cómo escalarías esto?"

Cambiaría SQLite por PostgreSQL (solo cambiar connection string gracias a SQLAlchemy). Añadiría autenticación. Separaría Streamlit en API REST + frontend. La lógica de negocio no cambiaría.

### 5. "¿Cómo testearías esto?"

Tests unitarios para cada módulo de src/ con datos mock. Tests de integración para flujos completos. La separación de capas facilita mockear dependencias.

### 6. "¿Qué mejorarías?"

Añadir logging estructurado. Completar type hints. Más tests. Cache de consultas frecuentes. Validación más robusta de inputs.

Vocabulario técnico que deberías dominar:

- **ORM (Object-Relational Mapping):** Mapear objetos Python a tablas SQL
  - **CRUD:** Create, Read, Update, Delete
  - **Repository Pattern:** Centralizar acceso a datos
  - **Dependency Injection:** Pasar dependencias en lugar de crearlas
  - **Separation of Concerns:** Cada módulo hace una cosa
  - **FIFO/LIFO:** Métodos de valoración de inventario
  - **ACID:** Atomicity, Consistency, Isolation, Durability (propiedades de BD)
  - **Facade Pattern:** Interfaz simple para subsistema complejo
  - **Strategy Pattern:** Algoritmos intercambiables
- 

## Resumen Final

### Lo que hicimos bien:

1.  Separación clara de capas
2.  Módulos con responsabilidad única
3.  Abstracción de la base de datos
4.  Código reutilizable
5.  Arquitectura escalable

### Lo que podríamos mejorar:

1.  Más tests unitarios
2.  Type hints completos
3.  Logging estructurado
4.  Documentación de API (docstrings)
5.  Manejo de errores más robusto

### Tu progreso como developer:

[Junior]—————[Tú estás aquí]—————[Senior]

Entiendes Python    Entiendes arquitectura    Diseñas sistemas

Escribes código      Separas responsabilidades      Mentorizas otros

Resuelves problemas      Piensas en mantenibilidad      Tomas decisiones técnicas

Este proyecto demuestra que puedes:

- Diseñar una arquitectura de 3 capas
- Aplicar patrones de diseño
- Pensar en escalabilidad
- Escribir código mantenable

**Esó es exactamente lo que buscan las empresas.**