🧮 Unimplemented Functions

🧩 Unimplemented Functions – Proposition 2 Fractional Asset Trading System

📄 Document Overview

This page outlines the unimplemented functions required to complete the fractional asset trading system for Proposition 2. The system is designed to create real-world assets, divide them into fractions, track fractional ownership, record value changes over time, and enable traceable trading.

✅ Current Implementation Status

The system already includes:

- ✅ Database models: *Users, Assets, Fractions, Offers, Transactions, AssetValueHistory*
- ✅ CRUD operations for: *Assets, Fractions, Offers, Transactions*
- ✅ Manual trading execution (accepting offers)
- ✅ Portfolio management (user holdings, transaction history)
- ✅ Manual asset value adjustments (admin-only)
- ✅ Basic fraction creation and management

🔧 Critical Missing Functions

1. Automatic Asset Value Updates via API

Priority: HIGH

Current Issue: Asset values are only updated manually.

Goal: Automate asset valuation updates via external APIs.

Required Features:

- *API-driven updates*
- *Scheduled value fetching*
- *Override tracking*

Code Implementation Idea:

```
# In AssetValueService
@staticmethod
def update_asset_value_from_api(asset_id: int, new_value: float, source_api: str, api_key: str = None) -> AssetValueHistory:
    """
    Update asset value from external API source.
    This can override manual administrator adjustments.
    """
    # Validate API source and key
    # Create value history record with source='api_update'
    # Update asset.total_value
    # Optionally notify administrators of override
```

Files to Add/Update:

- `app/services/asset_value_service.py`
- `app/controllers/asset_controller.py`
- `app/routes/assets.py`
- `app/integrations/` (e.g. `real_estate_api.py`, `stock_api.py`)
- `app/tasks/value_update_tasks.py` (for scheduling)
- `config.py` (API keys & scheduling config)

---

2. Fraction Splitting and Merging

Priority: HIGH

Goal: Allow users to split or merge fractions.

Features:

- Fraction can be split into smaller fractions
- Multiple fractions can be merged into one

Example – Splitting Function:

```
# In FractionService
@staticmethod
def split_fraction(fraction_id: int, split_ratios: List[float], new_owners: List[int] = None) -> List[Fraction]:
    """
    Split a fraction into multiple smaller fractions.

    Args:
        fraction_id: ID of fraction to split
        split_ratios: List of ratios for each new fraction (must sum to 1.0)
        new_owners: Optional list of new owners (if None, keeps same owner)

    Returns:
        List of newly created fractions
    """
    # Validate fraction exists and is active
    # Validate split ratios sum to 1.0
    # Create new fractions with proportional units
    # Mark original fraction as inactive
    # Create transaction records for the split
    # Update value_perunit for new fractions
```

Example – Merging Function:

```
# In FractionService
@staticmethod
def merge_fractions(fraction_ids: List[int], target_owner_id: int = None) -> Fraction:
    """
    Merge multiple fractions of the same asset into one.

    Args:
        fraction_ids: List of fraction IDs to merge
        target_owner_id: Owner of the merged fraction (if None, uses first fraction's owner)

    Returns:
        New merged fraction
    """
    # Validate all fractions belong to same asset
    # Validate all fractions are active
    # Calculate total units and weighted average value_perunit
    # Create new merged fraction
    # Mark original fractions as inactive
    # Create transaction records for the merge
```

Files to Update:

- `app/services/fraction_service.py`
- `app/controllers/fraction_controller.py`
- `app/routes/fractions.py`

---

3. Advanced Trading Features

Priority: MEDIUM

3.1 Automatic Offer Matching

- Matches compatible buy/sell orders automatically

```
# In TradingService
@staticmethod
def find_matching_offers(asset_id: int) -> List[Dict[str, Any]]:
    """
    Find and automatically execute matching buy/sell offers.
    """
    # Find overlapping price ranges
    # Execute trades automatically
    # Notify users of executed trades
```

3.2 Order Book Management

- Adds full order book tracking and sorting by price

```
# Order book for better price discovery
```

```
class OrderBookService:
    def get_order_book(self, asset_id: int) -> Dict[str, Any]:
        """
        Get complete order book for an asset.
        """
        # Return buy/sell orders sorted by price
        # Include depth information
```

3.3 Price Discovery Algorithm

- Calculates live asset price using trades and order spread

```
# Advanced pricing based on market activity
def calculate_market_price(asset_id: int) -> float:
    """
    Calculate market price based on recent trades and current offers.
    """
    # Analyze recent transaction prices
    # Consider current offer spreads
    # Apply weighted averaging
```

Files to Create/Update:

- `app/services/trading_service.py`
- `app/services/order_book_service.py`
- `app/services/pricing_service.py`
- `app/algorithms/price_discovery.py`

---

4. Enhanced Fraction Management

Priority: MEDIUM

4.1 Fraction Transfer Service

Allows ownership transfers via gifting, inheritance, sale, etc.

Transfer but not by selling approach.

```
# In FractionService
@staticmethod
def transfer_fraction(fraction_id: int, from_owner_id: int, to_owner_id: int,
                      transfer_type: str = 'gift', price: float = None) -> Transaction:
    """
    Transfer fraction ownership between users.

    Args:
        fraction_id: Fraction to transfer
        from_owner_id: Current owner
        to_owner_id: New owner
        transfer_type: 'gift', 'sale', 'inheritance', etc.
        price: Optional price for sale transfers
    """
    # Validate ownership
    # Create new fraction for recipient
    # Mark original fraction as inactive
    # Create transaction record
    # Handle payment if applicable
```

4.2 Fraction Valuation Service

Calculate live fraction value based on current asset value

```
# Real-time fraction valuation
def calculate_fraction_value(fraction_id: int) -> Dict[str, Any]:
    """
    Calculate current value of a fraction.
    """
    # Get latest asset value
    # Calculate fraction's proportional value
    # Include historical performance
    # Return detailed valuation report
```

Files to Create/Update:

- `app/services/fraction_valuation_service.py`
- `app/controllers/fraction_controller.py`

---

5. Advanced Portfolio Analytics

Priority: MEDIUM

5.1 Portfolio Analytics Service

```
# In PortfolioService
@staticmethod
def get_portfolio_analytics(user_id: int, period: str = '1Y') -> Dict[str, Any]:
    """
    Get comprehensive portfolio analytics.

    Returns:
        - Total portfolio value
        - Performance metrics
        - Risk analysis
        - Asset allocation
        - Historical performance
    """
```

5.2 Performance Tracking

```
# Track portfolio performance over time
def calculate_performance_metrics(user_id: int, start_date: datetime, end_date: datetime) -> Dict[str, Any]:
    """
    Calculate various performance metrics.
    """
    # Calculate returns, volatility, Sharpe ratio
    # Compare against benchmarks
    # Generate performance report
```

Files to Create:

- `app/services/performance_service.py`
- Update `models.py` with `PerformanceHistory`

---

6. Notification System

Priority: LOW

- Notify users of value changes, completed trades, etc.

```python
# Notification system for important events
class NotificationService:
    def send_trade_notification(self, user_id: int, trade_details: Dict[str, Any]):
        """
        Send notification about completed trade.
        """

    def send_value_change_notification(self, user_id: int, asset_id: int, old_value: float, new_value: float):
        """
        Send notification about asset value changes.
        """
```

Files to Create:

- `app/services/notification_service.py`
- `app/controllers/notification_controller.py`
- Update `models.py` with `Notification`

---

7. Audit and Compliance

Priority: LOW

- Add audit trails for compliance and rollback

```python
# Comprehensive audit logging
class AuditService:
    def log_asset_value_change(self, asset_id: int, old_value: float, new_value: float,
                              changed_by: int, reason: str, source: str):
        """
        Log asset value changes for compliance.
        """

    def log_fraction_transfer(self, fraction_id: int, from_user: int, to_user: int,
                             transfer_type: str, details: Dict[str, Any]):
        """
        Log fraction transfers for audit trail.
        """
```

Files to Create:

- `app/services/audit_service.py`
- Update `models.py` with `AuditLog`

🚦 Implementation Priority

Phase 1 (Critical – Immediate)

- ✅ Automatic Asset Value Updates via API
- ✅ Fraction Splitting and Merging

Phase 2 (Important – Short-term)

- 🔄 Advanced Trading Features
- 🔄 Enhanced Fraction Management

Phase 3 (Enhancement – Medium-term)

- 🔄 Portfolio Analytics
- 🔄 Notification System

Phase 4 (Compliance – Long-term)

- 🔄 Audit and Compliance

🧱 Technical Considerations

Database Updates

- Add `source` to `AssetValueHistory`
- Add `transfer_type` to `Transactions`
- Indexing for performance

API and Security

- RESTful endpoints
- API key management for integrations
- Input validation and logging

Performance

- Use async/background tasks where necessary
- Caching for order book and valuations

- Optimise queries on large datasets

🛑 Admin Override Policy

Manual admin overrides of API updates should:

- Be clearly logged with source "manual_override"
- Notify admins and stakeholders
- Preserve rollback capacity
- Be used in:
    - Emergency corrections
    - Market anomalies
    - Maintenance periods
    - Regulatory enforcement