Explanation of Design Patterns in StockInsight web application

1. Singleton Pattern

Part of code Implementation:

```python
class DatabaseConnection:  1 usage
    _instance = None
    _lock = threading.Lock()

    def __new__(cls):
        if not cls._instance:
            with cls._lock:
                if not cls._instance:
                    cls._instance = super(DatabaseConnection, cls).__new__(cls)
                    cls._instance._connection = None
        return cls._instance

    def connect(self):
        if not self._connection:
            base_dir = os.path.dirname(os.path.abspath(__file__))
            db_path = os.path.join(base_dir, '../../stock_data.db')
            try:
                self._connection = sqlite3.connect(db_path, check_same_thread=False)
                self._connection.row_factory = sqlite3.Row
                return self._connection
            except sqlite3.Error as e:
                print(f"Database connection failed: {e}")
                raise
        return self._connection

    def close(self):
        if self._connection:
            self._connection.close()
            self._connection = None
```

- The Singleton pattern is used to ensure that there is only one instance of the DatabaseConnection class throughout the application lifecycle.
- This is crucial for our application to efficiently manage database resources, avoid multiple connections, and ensure data consistency during concurrent operations.

- It helps prevent resource contention in a multithreaded environment, which is important for the smooth functioning of a stock market application with frequent database access.
- This is crucial for our application to efficiently manage database resources, avoid multiple connections, and ensure data consistency during concurrent operations.
- It helps prevent resource contention in a multithreaded environment, which is important for the smooth functioning of a stock market application with frequent database access.

2. Strategy Pattern

Part of code Implementation:

```python
from abc import ABC, abstractmethod

class PredictionStrategy(ABC):
    @abstractmethod
    def predict(self, issuer_name, look_back, days_ahead):
        pass

class LSTMPredictionStrategy:
    def predict(self, issuer_name, period):
        look_back = 60
        model_map = {
            1: "model_1_day.h5",
            7: "model_1_week.h5",
            30: "model_1_month.h5"
        }
        model_name = model_map.get(period)

        if not model_name:
            raise ValueError("Invalid period. Use 1, 7, or 30.")

        # Model loading and prediction logic ...

class PredictionContext:
    def __init__(self, strategy):
        self.strategy = strategy

    def set_strategy(self, strategy):
        self.strategy = strategy

    def predict(self, issuer_name, look_back, days_ahead):
        return self.strategy.predict(issuer_name, look_back, days_ahead)
```

- The Strategy pattern allows the application to dynamically switch between different prediction algorithms based on user needs or input.
- The context class (PredictionContext) enables switching strategies at runtime, making it possible to respond to user preferences or changing requirements dynamically. For example, the application can switch between short-term and long-term forecasting algorithms without significant refactoring.
- Each prediction algorithm is encapsulated in its own class, adhering to the Single Responsibility Principle. This reduces the complexity of the prediction module and ensures that each class focuses on one specific algorithm. This makes the code clean, easy to read, and maintain.
- Since algorithms are implemented independently, testing and debugging individual strategies is straightforward. Any modification or optimization in one algorithm does not affect others.
- This makes our application flexible and extensible, enabling us to easily integrate new prediction methods without modifying existing logic. Stock markets are dynamic and often require different approaches for analysis and prediction. With implementing Strategy Pattern in our application we can easily switch or add new prediction algorithms without altering the core application logic.

3. Model-View-Controller (MVC) Pattern

- **Model:** Responsible for data handling, including database interactions and business logic (e.g., db_singleton.py).
- **View:** Manages the user interface, templates, and static resources (e.g., templates and static folders).
- **Controller:** Handles application logic, routing, and coordination between the Model and View (e.g., controller folder).
- The MVC pattern organizes the application into three interconnected layers, ensuring clean separation of concerns which is very important for our application
- It is essential for our application to handle complex data interactions in the Model, user interfaces in the View, and routing/business logic in the Controller, allowing independent development and testing of each component.

- It enables efficient handling of user requests and ensures that the application remains responsive and modular, which is vital for maintaining a robust stock market analysis platform.
- We implemented it in our application because our application grows, we can easily add new views for additional user interfaces (e.g., mobile apps) or expand the Model to include new data sources, like real-time stock prices. Also if we decide to update the UI using modern front-end frameworks like React or Angular, we can do so without impacting the Controller or Model layers.