**Introduction**

The Backtesting Framework is designed to evaluate options trading strategies by simulating trades on historical data. It tracks the performance of these strategies through key metrics such as profit and loss (PnL) and portfolio greeks (delta, theta, gamma).

**Data Layer**

**DataHandler Class**

**Purpose**: Handles the fetching, preprocessing, and providing of historical data necessary for backtesting.

**Attributes**:

* data\_source: The source or path of the historical data.
* data\_params: Parameters for fetching data such as date range, symbols, and frequency.
* preprocessing\_params: Parameters for cleaning and aligning the data.

**Key Methods**:

* \_\_init\_\_(data\_source, data\_params, preprocessing\_params): Initializes the data handler with specified parameters.
* fetch\_and\_process\_data(params): Fetches and preprocesses historical data.
* fetch\_data(params): Retrieves raw data from the source.
* preprocess\_data(raw\_data): Cleans and aligns the raw data.

**Signal Layer**

**Signal Class**

**Purpose**: Generates trading signals based on predefined criteria or strategies. You can modify and keep adding new strategies to this class in the form of methods.

**Attributes**:

* signal\_params: Parameters for signal generation, such as indicators or thresholds.
* signals: A DataFrame to store generated signals.

**Key Methods**:

* \_\_init\_\_(signal\_params): Initializes the signal generator with specified parameters.
* generate\_signals(data): Generates signals based on the input data and predefined criteria.

**Strategy Layer**

**Position Class**

**Purpose**: Represents a trading position, including entry and exit details.

**Attributes**:

* entry\_timestamp: The timestamp when the position was entered.
* legs: A list of legs (individual trades) that make up the position.
* entry\_premium: The premium paid or received when entering the position.
* margin\_used: The margin required for the position.
* strategy\_type: The type of strategy (e.g., credit or debit spread).
* stop\_loss: The stop loss level for the position.
* target\_profit: The target profit level for the position.
* exit\_premium: The premium at the time of exit.
* exit\_timestamp: The timestamp when the position was exited.
* pnl: The profit or loss of the position.

**Key Methods**:

* \_\_init\_\_(entry\_timestamp, legs, stop\_loss, target\_profit, entry\_premium, margin\_used, strategy\_type): Initializes the position with specified parameters.
* add\_exit\_details(exit\_timestamp, pnl): Adds exit details to the position.
* get\_current\_leg\_prices(options\_data, timestamp): Retrieves the current prices of the legs in the position.

**DynamicLegStrategy Class**

**Purpose**: Defines the logic for dynamically generating the legs of a strategy.

**Attributes**:

* strategy\_params: Parameters for defining the strategy.

**Key Methods**:

* \_\_init\_\_(strategy\_params): Initializes the strategy with specified parameters.
* get\_legs(options\_data, timestamp, atm\_strike, instruments): Generates the legs for a given strategy.

**Execution Layer**

**SpreadBacktester Class**

**Purpose**: Manages the execution of trades and the backtesting process.

**Attributes**:

* options\_data: The options data used for backtesting.
* signals: The trading signals generated for the strategy.
* stop\_loss: The stop loss level for the strategy.
* target\_profit: The target profit level for the strategy.
* instruments\_with\_actions: Instruments and actions defined for the strategy.
* sl\_percentage\_based: Boolean indicating if stop loss is percentage based.
* tp\_percentage\_based: Boolean indicating if target profit is percentage based.
* strategy\_type: The type of strategy being backtested.
* positions: A list of active positions.
* trades: A list of executed trades.
* positions\_not\_counted: Counter for positions not meeting criteria.
* portfolio\_metrics: Dictionary for tracking portfolio greeks (delta, theta, gamma).

**Key Methods**:

* \_\_init\_\_(options\_data, signals, stop\_loss, target\_profit, instruments\_with\_actions, sl\_percentage\_based, tp\_percentage\_based, strategy\_type): Initializes the backtester with specified parameters.
* classifying\_signals(): Classifies signals into entry and exit signals.
* execute\_trades(timestamp): Executes trades based on signals and manages positions.
* enter\_spread(timestamp): Enters a spread position based on the signals.
* manage\_positions(timestamp): Manages active positions and checks exit conditions.
* check\_exit\_conditions(position, timestamp): Checks if exit conditions are met for a position.
* close\_position(position, timestamp, reason): Closes a position and records trade details.

**Overall Structure and Connections**

1. **DataHandler**: The starting point where historical data is fetched and preprocessed. This data is essential for generating signals and backtesting strategies.
2. **Signal**: Uses the historical data to generate trading signals based on predefined criteria. These signals indicate potential entry and exit points for trades.
3. **Position**: Represents individual trades and their details, including entry, exit, and profit/loss.
4. **DynamicLegStrategy**: Defines the logic for creating the legs of a strategy dynamically based on the current market conditions.
5. **SpreadBacktester**: Manages the overall backtesting process, including executing trades based on signals, managing positions, and tracking performance.

**Workflow**

1. **Initialization**: The DataHandler is initialized and historical data is fetched and preprocessed.
2. **Signal Generation**: The Signal class generates trading signals using the processed data.
3. **Strategy Execution**: The SpreadBacktester uses the signals to enter and manage positions. The DynamicLegStrategy helps define the specific trades to be executed.
4. **Trade Management**: The Position class tracks each trade, including entry and exit details.
5. **Performance Tracking**: The SpreadBacktester tracks overall performance, including portfolio greeks and trade outcomes.

This structure ensures that each component interacts seamlessly to enable comprehensive backtesting of trading strategies.

**Conclusion**

This documentation provides an overview of the Backtesting Framework, outlining the purpose and functionality of each part of the codebase. By understanding the interaction between the Position and SpreadBacktester classes and their methods, users can effectively use and extend the framework to backtest various trading strategies.