API Documentation

Strategy Development API

Base Strategy Interface

All trading strategies must inherit from the base `Strategy` class:

cpp class Strategy { public: virtual void on_data(const Candle& candle, const Candle* high_tf = nullptr) = 0; virtual bool should_buy() const = 0; virtual bool should_sell() const = 0; virtual void enable_debug(const std::string& strategy_name, const std::string& symbol) {} virtual void init(TradeEngine* engine) { this->engine = engine; } virtual ~Strategy() = default;

protected: TradeEngine* engine = nullptr; };

Methods

`on_data(const Candle& candle, const Candle* high_tf = nullptr)` - **Purpose**: Process new market data and update strategy state - **Parameters**: - `candle`: Current timeframe market data (typically 1-minute) - `high_tf`: Higher timeframe data (optional, e.g., 15-minute) - **Called**: For every new market data point - **Implementation**: Update indicators, generate signals

`should_buy() const` - **Purpose**: Indicate if strategy wants to open a long position - **Returns**: `true` if buy signal is active, `false` otherwise - **Called**: After each `on_data()` call

`should_sell() const` - **Purpose**: Indicate if strategy wants to close current position - **Returns**: `true` if sell signal is active, `false` otherwise - **Called**: After each `on_data()` call

`enable_debug(const std::string& strategy_name, const std::string& symbol)` - **Purpose**: Enable debug logging for strategy development - **Parameters**: - `strategy_name`: Name of the strategy - `symbol`: Trading symbol (e.g., "BTCUSDT") - **Implementation**: Open debug log file and set debug flag

`init(TradeEngine* engine)` - **Purpose**: Initialize strategy with trading engine reference - **Parameters**: `engine`: Pointer to the trading engine - **Usage**: Access portfolio information and trading capabilities

Data Structures

Candle Structure

cpp struct Candle { std::string symbol; // Trading symbol std::string timestamp_str; // Human-readable timestamp std::chrono::system_clock::time_point timestamp; // System timestamp double open; // Opening price double high; // Highest price double low; // Lowest price double close; // Closing price double volume; // Trading volume };

Trade Record Structure

cpp struct TradeRecord { std::string symbol; // Trading symbol std::chrono::system_clock::time_point entry_time; // Trade entry time std::chrono::system_clock::time_point exit_time; // Trade exit time double signal_price; // Price when signal generated double entry_price; // Actual entry price (with slippage) double exit_price; // Actual exit price (with slippage) double profit; // Trade profit/loss bool win; // True if profitable trade };

Strategy Examples

1. Simple Moving Average Strategy

cpp class SMA_Strategy : public Strategy { public: SMA_Strategy(int short_period = 3, int long_period = 8); void on_data(const Candle& low_tf, const Candle* high_tf = nullptr) override; bool should_buy() const override; bool should_sell() const override; void enable_debug(const std::string& strategy_name, const std::string& symbol) override;

private: std::deque price_window; int short_period; int long_period; double short_sma = 0.0; double long_sma = 0.0; bool buy_signal = false; bool sell_signal = false; bool in_position = false; std::ofstream debug_log;

```
double compute_sma(const std::deque<double>& prices, int period) const;
```

};

Implementation Example: cpp void SMA_Strategy::on_data(const Candle& low_tf, const Candle* high_tf) { price_window.push_back(low_tf.close); if (price_window.size() > static_cast(long_period)) price_window.pop_front();

```
buy_signal = sell_signal = false;

if (price_window.size() >= static_cast<size_t>(short_period)) {
    short_sma = compute_sma(price_window, short_period);

    if (price_window.size() >= static_cast<size_t>(long_period)) {
        long_sma = compute_sma(price_window, long_period);
    }

    double current_price = low_tf.close;
    bool momentum_up = /* calculate momentum */;
    bool above_sma = current_price > short_sma;

    if (!in_position && momentum_up && above_sma) {
        buy_signal = true;
        in_position = true;
    }
    else if (in_position && (/* exit conditions */)) {
        sell_signal = true;
        in_position = false;
```

```
    }
}

}
```

2. EMA-RSI Strategy

cpp class EMARSI_Strategy : public Strategy { public: void on_data(const Candle& low_tf, const Candle* high_tf = nullptr) override; bool should_buy() const override; bool should_sell() const override; void enable_debug(const std::string& strategy_name, const std::string& symbol) override;

private: std::deque candles; double short_ema = 0.0, long_ema = 0.0; double rsi = 50.0, atr = 0.0; double entry_price = 0.0; bool in_position = false; int cooldown = 0; bool buy_signal = false, sell_signal = false;

```
// Strategy parameters
static constexpr int short_period = 20;
static constexpr int long_period = 100;
static constexpr int rsi_period = 14;
static constexpr double rsi_oversold = 30.0;
static constexpr double rsi_overbought = 70.0;

double ema(double prev_ema, double price, int period);
double compute_rsi() const;
double compute_atr() const;
```

};

Trading Engine API

TradeEngine Class

cpp class TradeEngine { public: TradeEngine(double capital, Mode mode, const std::string& log_file, int latency_sec = 60, double slippage_bps = 5.0, double stop_loss_pct = 0.02, double take_profit_pct = 0.04, double max_drawdown_pct = 0.5);

```
void update(const Candle& candle);
void queue_buy(double price, const std::string& ts,
               std::chrono::system_clock::time_point tp,
               const std::string& symbol);
void queue_sell(double price, const std::string& ts,
                std::chrono::system_clock::time_point tp,
                const std::string& symbol);

const std::vector<TradeRecord>& get_trades() const;
double get_portfolio_value() const;
double get_realized_pnl() const;
```

};

Methods

`queue_buy(double price, const std::string& ts, time_point tp, const std::string& symbol)` - **Purpose**: Queue a buy order for execution - **Parameters**: - `price`: Signal price when buy decision was made - `ts`: Timestamp string for logging - `tp`: System timestamp - `symbol`: Trading symbol - **Behavior**: Order will be executed after configured latency with slippage

`queue_sell(double price, const std::string& ts, time_point tp, const std::string& symbol)` - **Purpose**: Queue a sell order for execution - **Parameters**: Same as `queue_buy` - **Behavior**: Closes current position after latency with slippage

`update(const Candle& candle)` - **Purpose**: Update engine state with new market data - **Parameters**: `candle`: Current market data - **Behavior**: Processes pending orders, updates unrealized P&L

Risk Management API

RiskManager Class

cpp class RiskManager { public: RiskManager(const RiskLimits& limits);

```
bool can_open_position(const std::string& symbol, double size, double price);
bool can_increase_position(const std::string& symbol, double
additional_size);

void update_positions(const std::unordered_map<std::string, double>&
positions);
void update_pnl(double realized_pnl, double unrealized_pnl);

std::vector<std::string> get_risk_alerts() const;
bool is_risk_breach() const;

const RiskMetrics& get_metrics() const;
const RiskLimits& get_limits() const;
```

};

RiskLimits Structure

cpp struct RiskLimits { double max_position_size = 0.1; // 10% of portfolio double max_daily_loss = 0.02; // 2% daily loss limit double max_drawdown = 0.05; // 5% max drawdown double var_limit = 0.03; // 3% VaR limit double correlation_limit = 0.7; // Max correlation between positions int max_positions = 5; // Max concurrent positions };

Portfolio Management API

PortfolioManager Class

cpp class PortfolioManager { public: PortfolioManager(double initial_capital, std::shared_ptr risk_mgr);

```
bool open_position(const std::string& symbol, double quantity, double price,
                   const std::string& timestamp);
bool close_position(const std::string& symbol, double quantity, double price,
                    const std::string& timestamp);

const Position* get_position(const std::string& symbol) const;
std::vector<Position> get_all_positions() const;
double get_total_portfolio_value(const std::unordered_map<std::string,
double>& current_prices) const;

PortfolioMetrics get_metrics() const;
bool can_trade(const std::string& symbol, double quantity, double price);

double get_cash() const;
```

};

Position Structure

cpp struct Position { std::string symbol; double quantity = 0.0; double avg_price = 0.0; double unrealized_pnl = 0.0; double realized_pnl = 0.0; std::chrono::system_clock::time_point entry_time; std::vector trades; };

Performance Analytics API

PerformanceAnalyzer Class

cpp class PerformanceAnalyzer { public: static PerformanceReport analyze_strategy(const std::vector& trades, const std::vector& benchmark_returns, double initial_capital, const std::string& strategy_name = “”);

```
static void generate_html_report(
    const PerformanceReport& report,
    const std::string& filename,
    const std::string& strategy_name
);

static void compare_strategies(
    const std::vector<std::pair<std::string, PerformanceReport>>& strategies,
    const std::string& output_file
);
```

};

PerformanceReport Structure

cpp struct PerformanceReport { // Return Metrics double total_return = 0.0; double annualized_return = 0.0; double sharpe_ratio = 0.0; double sortino_ratio = 0.0; double calmar_ratio = 0.0;

```
// Risk Metrics
double max_drawdown = 0.0;
double volatility = 0.0;
double var_95 = 0.0;
double var_99 = 0.0;
double beta = 0.0;

// Trade Metrics
int total_trades = 0;
double win_rate = 0.0;
double avg_win = 0.0;
double avg_loss = 0.0;
double profit_factor = 0.0;
double avg_trade_duration_hours = 0.0;
```

};

Data Loading API

DataLoader Functions

cpp // Load single CSV file std::vector load_csv_data(const std::string& filename);

// Load multiple assets std::unordered_map<std::string, std::vector> load_multiple_assets(const std::vector<std::pair<std::string, std::string>>& asset_files);

TimeframeAggregator Class

cpp class TimeframeAggregator { public: void set_timeframes(const std::vector& tfs); void add(const Candle& candle); const Candle* get_latest(const std::string& tf) const;

private: int tf_to_seconds(const std::string& tf) const; };

Supported Timeframes: - `"1m"`: 1 minute (60 seconds) - `"5m"`: 5 minutes (300 seconds) - `"15m"`: 15 minutes (900 seconds) - `"1h"`: 1 hour (3600 seconds)

Utility Functions

Metrics Calculation

cpp class Metrics { public: static MetricsResult compute(const std::vector& trades, double initial_capital, const std::string& equity_csv_file = ““); };

struct MetricsResult { double cumulative_return = 0; double max_drawdown = 0; double sharpe_ratio = 0; double sortino_ratio = 0; int total_trades = 0; double win_rate = 0; double avg_profit = 0; };

Trade Logging

cpp void export_trades_to_csv(const std::string& filename, const std::vector& trades);

Configuration and Constants

Default Parameters

cpp // Trading Engine Defaults const double DEFAULT_SLIPPAGE_BPS = 2.0; // 2 basis points const int DEFAULT_LATENCY_SEC = 5; // 5 seconds const double DEFAULT_STOP_LOSS = 0.02; // 2% const double DEFAULT_TAKE_PROFIT = 0.04; // 4%

// Risk Management Defaults const double DEFAULT_MAX_POSITION = 0.1; // 10% const double DEFAULT_MAX_DAILY_LOSS = 0.02; // 2% const double DEFAULT_MAX_DRAWDOWN = 0.05; // 5% const int DEFAULT_MAX_POSITIONS = 5;

// Strategy Defaults const int DEFAULT_COOLDOWN_CANDLES = 10; const double DEFAULT_VOLUME_THRESHOLD = 1.2;

File Paths and Naming Conventions

cpp // Log Files “logs/trades_{STRATEGY}_{SYMBOL}.csv” // Trade records “logs/{STRATEGY}*{SYMBOL}equity.csv” // Equity curve ”logs/debug{STRATEGY}*{SYMBOL}.csv” // Debug information

// Report Files “reports/{STRATEGY}_{SYMBOL}_report.html” // HTML performance report “reports/optimization_results.txt” // Optimization results “reports/system_metrics.csv” // System performance metrics

// Plot Files “plots/{STRATEGY}_equity_curve.png” // Equity curve chart “plots/{STRATEGY}_drawdown.png” // Drawdown chart “plots/{STRATEGY}_win_loss_distribution.png” // Win/loss distribution ```

This API documentation provides the essential interfaces for developing custom strategies and extending the TradePulse system. All classes follow RAII principles and provide exception-safe operations.