TECHNICAL GUIDE

Algorithmic Municipal Bond Trading Using Python

Building systematic municipal-bond workflows with BondPUB

Published September 2026

Financial Technology Laboratories, Inc.

About this guide

This technical guide describes the data, services and application patterns required to automate municipal-bond trading workflows with Python and BondPUB.

The material is intended for trading-technology leaders, architects and developers evaluating systematic quote publication, RFQ response, inventory accumulation and divestment, order execution and downstream trade reporting.

Examples illustrate workflow logic and integration patterns. Firms should apply their own risk, compliance, security and supervisory controls and validate implementation details against current FTLabs interfaces.

IN THIS PUBLICATION

The municipal fixed income market remains one of the largest yet least electronically automated asset classes. While U.S. Treasuries are largely electronic and highly liquid, municipals are still traded over the phone or via request-for-quote (RFQ) protocols. The municipal market is characterized by fragmented liquidity, hundreds-of-thousands of distinct securities, and dealer-driven price discovery. As a result, municipal market participants face operational inefficiencies, inconsistent price transparency, and high quote-to-trade ratios without widespread intelligent automation.

Automating less liquid securities like municipals introduces complexities that are not present in equities or on-the-run government bonds. Factors such as varying credit quality, tax treatment, structural features, and localized demand make generic pricing models unreliable. Moreover, trade execution workflows must incorporate compliance filters, accurate calculations, inventory constraints, and trader-defined discretion. The aggregation of liquidity from alternative transaction systems (ATS) platforms, order management systems (OMS) and execution management systems (EMS) offers some enhanced liquidity for these instruments, but also implies that quote generation, RFQ filtering, and order routing in such a fragmented environment must be algorithmically managed with minimal latency and high reliability, especially during periods of market volatility.

Fixed income ATSs such as Tradeweb, ICE BondPoint, MarketAxess, and ICE TMC have enabled partial systematic trade automation through click-to-buy and RFQ workflows. These platforms facilitate systematic trading by allowing broker-dealers to expose inventory, respond to inquiries, and execute trades electronically. Automated engagement with these venues requires systems capable of managing simultaneous quote distribution and aggregation, RFQ response logic, and real-time pricing overlays within regulatory and inventory constraints.

As market structure has evolved, additional sources of electronic liquidity - including buy-side EMSs that support outbound RFQs - become increasingly relevant. Integrating with these EMS platforms expands the dealer’s electronic reach and complements existing ATS connectivity. A unified architecture that incorporates both ATS venues and EMS-based RFQ flows enables the algorithms described below to operate across a broader spectrum of counterparties, increasing execution opportunities while preserving pricing control and compliance integrity.

The BondPUB system from Financial Technology Laboratories, Inc. implements a unified, high-level common interface layer for quote and RFQ aggregation and distribution, order processing, and trade reporting. This interface hides the differences in the trading workflows and protocols of the ATS vendors and the reporting protocols of the clearing vendors.

2. Required Components

Each firm that wants to implement algorithmic municipal fixed income trading must consider the following components of a robust system - both data and system elements.

Data

The quality of the data used within the system is important - include the following classes of data:

- **Market Price Data -** Accurate pricing is important to the success of fixed income algorithmic trading - whether that is used to price outgoing quotes, respond to RFQs, or identify and prioritize deals. The quality of the price data largely determines the profitability of these algorithmic workflows.
- **Bond Description Data -** BondPUB supports the use of multiple descriptive data vendors for retrieval of varying levels of information as needed. Some of the sources of municipal bond terms & conditions data may be open source and others may require licensing. The data vendor(s) must provide the data elements needed to either identify, calculate, or screen securities.

System Components

The system capabilities are centrally important as well. In addition to the highest quality input data, successful algorithmic trading needs modular, robust and flexible software systems. The foundational components provided with the BondPUB system include:

- **Integration Services -** BondPUB incorporates access to their Integration-Platform-as-a-Service (IPaaS), hosted on Amazon AWS, where the network integrations to the major ATSs, clearing vendors, and data vendors are hosted. Access to these services is normalized, aggregated, and delivered to the client-installed BondPUB server system over secure internet connections.
- **Calculation Services -** BondPUB incorporates the Fixed Income Security Analytics (FISA) bond calculation services for performing accurate municipal bond calculations.
- **Client-Side Trading Middleware -** BondPUB provides an application server hosted on the client premises or on the cloud to handle the real-time messaging traffic between the integration platform and the traders or other users of either user-interface or API services.
- **Trading User-Interface -** BondPUB provides a Windows user-interface (UI) application to monitor the firm's algorithmic trading, which also functions as an OMS and EMS. This allows traders the ability to manage orders and trades for their sales channels or external trading venues such as ATS platforms. A web-based HTML5 front-end is also available for extending the firm's product to their retail or institutional sales channels.
- **System Service Architecture -** For algorithmic trading, software development kit (SDK) for accessing the BondPUB API using Python, MS .NET, or Java may provide the flexibility needed for the firm to implement their ideas, incorporating the capabilities of the BondPUB UI as needed. The system API services are reviewed in greater detail in the next section.

3 System service architecture

BondPUB serves as a modular API gateway that abstracts differences across trading venues and clearing firms, providing consistent services for quoting, RFQs, orders, and trades.

BondPUB provides a secure and modular API gateway designed to provide municipal market participants with programmatic access to fixed income trading venues, RFQ workflows, and trade submission infrastructure. At its core, BondPUB acts as a middleware layer between trader-facing systems and external market venues, abstracting away the technical variations across trading platforms and systems, differences in protocols, and clearing firm interfaces.

BondPUB offers an API suite with endpoints for retrieving security descriptions, market prices, receiving and publishing quotes, receiving and publishing RFQs, receiving and confirming orders, receiving and completing trades notifications, and receiving and completing post-trade messages.

To simplify the presentation of the application architecture, discussion of algorithm implementation is limited to a few of the key service models: Price, MarketRFQ, Quote, and Order.

Within each of these services objects, the BondPUB API exposes method calls such as create_quote(), update_quote(), subscribe_to_quote_updates(), on_order_update(), get_orders() and so on. These service modules are orchestrated through asynchronous workflows, enabling simultaneous handling of incoming market data, quoting, RFQ processing, and order dialogue. For readability, the method call parameters are not included. For more information on the method parameters accepted refer to the detailed API documentation.

For example, an application using the BondPUB Python SDK to implement an AutoPublish algorithm:

- Ingest the firm's positions
- `Interface with the BondPUB API to create_quotes()`
- `Retrieve market prices with get_price()`
- `Stream those quotes to the designated trading destinations with quote_publication().`

As calls are made to on_order_update() to register incoming orders, quote levels are updated using update_quote() and re-distributed with quote_publication().

Acceptable orders are acknowledged and confirmed with update_order().

BondPUB's API abstraction ensures consistent application logic across multiple trading venues and clearing relationships. This enables the execution of trading logic in near real-time.

4. Automation Algorithms

Five basic algorithmic modules suitable for municipal bonds are described below, each designed to automate the logic of a specific workflow for the municipal fixed income market.

The example algorithms described below illustrate some of the workflows that can be automated, but use of the BondPUB API allows the implementation of many other possible algorithms.

These algorithms interface with the BondPUB API's Service layer - particularly the Price, MarketRFQ, Quote, and Order models - to query instrument prices, stream quotes, respond to RFQs, and receive and execute orders. Each algorithm is independently configurable and designed to operate concurrently across multiple trading venues.

- AutoPublish streams municipal offering (or bid) quotes to connected trading venues for transacting. It uses the Quote quote_publications() to create offering quotes and then adjust available size dynamically as orders are received. As fills are received through on_order_update(), update_order_status() is used to communicate acceptance and confirmation. This algorithm ensures continuous quoting while managing exposure limits per security and venue to eliminate overselling.
- AutoRespond uses MarketRFQs via on_marketRFQ_update()to receive incoming aggregated municipal bid-wanted (or offer-wanted) RFQs. Each RFQ is filtered based on bond characteristics, lists of security identifiers to exclude, and internal inventory checks. If an RFQ passes all the checks, the algorithm retrieves the market price and submits a bid quote against the bid-wanted via quote_publication(). If awarded as the winning high bid, order_execute() is triggered.
- AutoAccumulate monitors aggregated market quote feeds from connected trading venues and identifies bonds that match a buy-list or satisfy acquisition filters of bond characteristics (e.g., issuer,

maturity range, state, rating). Once a candidate bond is identified, the algorithm places an order directed at the quote using create_order(). Orders are updated via order_update() and on_order_update_order().

- AutoDivest reduces or eliminates positions in specific bonds at the best price. It publishes offer quotes using quote_publication() and, if the position remains unsold over a defined interval, the algorithm gradually reduces the price and sends updates through quote_publication(...). This price-decay model increases execution probability while controlling timing and loss thresholds.
- DealFinder scans incoming market quotes and compares the prices against a source of market prices. It ranks opportunities for purchase based on spread or other user-defined properties. For qualified opportunities, it can route orders through create_order() and order_execute()depending on the committed budget.

5 Security description integration

Efficient and accurate security identification and description is a foundational step in any fixed income trading workflow. Before publishing an offering quote, accepting an order, or clearing a trade, the security in question must be properly identified and described, so that it can be screened for suitability and accurately calculated. This process ensures that for a particular CUSIP, the issuer name, maturity date, coupon rate, call features, and other key attributes necessary for accurate calculation of price/yield are available for use.

Accurate and complete descriptive data is also needed so that the filtering and evaluation functions that make up the heart of any algorithm can classify and evaluate the characteristics of the bond to make the important decisions about whether to transact.

The BondPUB API addresses this critical requirement through its Issue model which provides a streamlined, bi-directional communication mechanism for handling these security lookups. The service supports two primary functions:

lookup_issue_by_cusip(...) - Initiate a request from the dealer's system to the BondPUB platform, typically using a CUSIP to retrieve the bond description from the persistence layer. If the bond data is not yet available, the BondPUB system requests and retrieves it from a configured data source for municipal descriptive data.

get_issue(...) - Return the set of standardized descriptive fields necessary for calculation and filtering prior to quote publication, order validation, or trade capture.

This interaction allows for seamless integration with the firm's internal security master or third-party vendors like ICE, Bloomberg, or others. BondPUB acts as an intermediary, normalizing responses from multiple data providers.

In the context of algorithmic trading, pre-trade workflows often include AI-driven filtering and classification logic using elements of the bond description. For instance, a Python-based quoting engine may receive a list of municipal bond RFQs and perform a batch get_issue() operation to populate the persistence layer with full security descriptions to filter and bid only those bonds issued

in a particular state, with maturity date in a particular date range, with ratings in a particular range of ratings, or other criteria.

6 Pricing feed integration

The BondPUB API framework is designed to accept security prices from multiple sources for different security types, including third-party vendors and dealer-developed internal pricing engines, and serve them out through a consistent interface. Regardless of whether prices are delivered via streaming WebSockets, scheduled batch file download, or REST APIs, BondPUB transforms all incoming data into a normalized schema usable through the BondPUB API service models.

To support this dynamic environment, BondPUB's API Price model provides the following functions:

subscribe_to_price_updates(...) - Initiate a real-time subscription to a pricing source and the service begins delivering streaming updates via get_price().

on_price_update(...) - Upon change, receive the most recent price or rate in a normalized format that includes metadata such as bid/ask yield, bid/ask price, and timestamp.

get_price(...) - Request a specific price without initiating a subscription.

This approach also allows BondPUB to support validation using a secondary price feed. A primary pricing feed may be supplemented by a secondary feed to act as a 'guardrail' for validation prior to quote publication. For example, if the primary feed values a bond at 101.25 but the guardrail feed shows 98.75 which exceeds a defined variance limit, the application may suppress the quote and flag the security for review. This type of validation improves pricing robustness and reduces the chance of misquoting.

Additional pricing controls are available through BondPUB API, including:

- the use of multiple simultaneous RFQ Filters to match bonds with specific characteristics
- bid & offer side Markup Rulesets per trading account that adjusts outgoing quotes by a markup per destination based on quantity and time to maturity
- a Filter Aggression Factor that adjusts the prices from a pricing source up or down by a factor for all outgoing bids/offers published against RFQs matching a particular RFQ Filter.
- RFQ Filter Action allows the firm to control how RFQ responses are priced and submitted:
    - 'Auto-Add' RFQs matching a Filter to the Bid/Offer List for manual processing,
    - 'Auto-Price' for manual trader approval supports automatic pricing of RFQs but requires manual review and submission of the bid/offer, or
    - 'Auto-Respond' which automatically prices and submits the bid/offer against an RFQ matching a Filter. After submission, there remains a manual review period leading up to the Due Time of each RFQ.

7 Quote integration

Effective algorithmic trading in the fixed income market requires comprehensive quote management as a prelude to efficient order execution. To support this, the BondPUB API provides the Quote

model - a standardized interface for managing inbound and outbound quoting workflows across multiple ATSs. This service includes methods such as get_quote(), on_quote_update(), update_quote(), and quote_publication(). Together, they enable an event-driven framework for implementing algorithmic strategies such as AutoPublish, AutoDivest, and DealFinder introduced above.

In algorithmic workflows such as AutoAccumulate or AutoDivest, these models can be used to discreetly source liquidity for bonds targeted for accumulation or liquidation. For example, a variation of AutoAccumulate may identify bonds that meet credit and yield curve targets but lack actionable depth in displayed quotes. In such cases, the algorithm may use quote_publication() to initiate interest and elicit response.

Algorithmic strategies such as DealFinder and AutoPublish depend heavily on this function to track the best bid/ask across venues, identify arbitrage opportunities, or to identify pricing mismatches between the prices on the market quotes received and pricing retrieved through the Price model. For instance, a variation of the DealFinder algorithm can compare incoming market quotes to the price retrieved from a favored pricing service and prioritize the most undervalued for purchase and automatic relisting with an accurate market price via AutoPublish.

- **quote_publication(...)** is the primary mechanism for submitting ask or bid prices to one or more venues. Each call specifies at least CUSIP, side, size, price and targeted platforms. Quotes published through this function are automatically formatted and routed by BondPUB, enabling consistent deployment of pricing logic across the fragmented ecosystem of ATS and buyside platforms. Quote refresh intervals, expiration handling, and venue-specific nuances are abstracted by the API, allowing the algorithm to focus purely on pricing logic and execution intent.
- **receive_publication_status(...)** provides real-time feedback on the status of quotes previously submitted to trading venues via the quote_publication() function. This function does not initiate any change to the state of a quote or RFQ; rather, it is designed solely to receive acknowledgements, confirmations, or rejections from each connected venue regarding the published activity. By listening to receive_publication_status(), the algorithm ensures it has up-to-date knowledge of what the market sees, which is essential for inventory reconciliation, compliance audit trails, and reliable strategy execution.

The combination of these capabilities allows an algorithm using the system to operate with a high degree of autonomy while maintaining compliance, pricing discipline, and inventory alignment. By embedding these functions into event-driven Python strategies, market participants can scale their quoting operations, respond fluidly to market demand, and uncover hidden trading opportunities within a single, standardized interface.

8 Request for quote integration

In the same way, comprehensive RFQ management complements the Quote workflows and brings added dimension to the possible algorithms. The BondPUB API supports this through the MarketRFQ model which standardizes the interfaces for managing inbound request-for-quote workflows across multiple ATSs. This service includes core functions such as get_marketRFQ(), on_marketRFQ_update() and subscribe_to_marketRFQ_updates(). Together, they enable a unified, event-driven framework for

implementing algorithmic strategies such as AutoRespond and AutoAccumulate. For real-time handling of incoming RFQs, the following functions are used:

- **subscribe_to_marketRFQ_updates() -** Initiate a real-time subscription of the aggregated RFQ sources available to the client.
- **on_marketRFQ_update(...) -** Enable real-time ingestion of inbound RFQs from market participants across connected venues. Each message includes CUSIP, side (buy or sell), requested quantity, venue, timestamp, etc.
- **get_marketRFQ() -** Request a specific RFQ for screening and response. This function acts as the entry point for the AutoRespond algorithm, triggering evaluation of the security and retrieval of pricing based on the security terms & conditions, include/exclude lists, lot size, and market conditions.

For instance, as part of the implementation of the AutoRespond algorithm, the application may include an evaluation function that declines to respond to RFQ inquiries smaller than a certain size, securities from issuers on an exclude list, or issues with certain uses of proceeds (e.g. lease revenue). If the security passes all the evaluation criteria, the system assigns a price and responds using the quote_publication() function. This closed-loop automation allows firms to efficiently engage with inbound demand while preserving trader oversight for edge cases or high-risk exposures.

In an algorithmic workflow such as AutoAccumulate, the Quote and MarketRFQ models may be used separately or in combination to discreetly source liquidity for bonds targeted for accumulation.

9 Order receipt and execution

Once a quote or RFQ interaction leads to a match or acceptance, the workflow shifts to the completion of the order.

The BondPUB API provides the Order model to standardize and automate this phase of the trading lifecycle.

The Order model includes the following key functions that support full-cycle order execution and post-trade messaging, enabling both passive (quote-based) and active (order-routed) workflows to integrate seamlessly with middle- and back-office systems.

- **subscribe_to_order_updates() -** Register to receive real-time aggregated order updates from connected venues.
- **on_order_update(...)**- Receive details of the order, such as CUSIP, side, settlement date, size, price, counterparty, and execution venue. This serves as the primary event that signals an execution is underway.
- **get_order() -** Retrieve the order object for further processing.
- **update_order_status() -** Send status updates to the venue or counterparty, such as cancel or fill notification.
- **execute_order() -** manage asynchronous status changes and events for orders. Fixed income order workflows often involve partial fills, cancellations, amendments, or expiration notices, depending on the rules of the underlying venue.

10 Trade and clearing integration

The Trade model abstracts away the clearing vendor-specific differences in trade messaging, allowing algorithmic applications to operate over a consistent, unified interface. It enables real-time interaction with in-house or vendor clearing systems.

Together with the Order model, the Trade model supports a responsive and compliant order lifecycle management without requiring venue-specific logic in the strategy layer. These functions are available to use in all strategies to complete orders identified and effected through pricing, quoting, and RFQ responses.

- **subscribe_to_trade_updates() -** Register to receive real-time trade updates from a clearing vendor or destination.
- **on_trade_update(...)**- Receive details of the trade such as status change, response message, or transaction identifier.
- **save_and_report_trade(...) -** Submit completed trades to downstream clearing and regulatory systems. This includes trade information such as the executed price, size, counterparty, time of execution, regulatory and compliance information, and clearing details. BondPUB handles the formatting and submission of this data to supported clearing vendors, ensuring the trade is captured in the appropriate books and records system. This function is used in all strategies to ensure that trades resulting from quote or RFQ activity are properly reconciled and reported. Whether the trade originated on a dealer-to-dealer platform or from a client RFQ interaction, save_and_report_trade() enables straight-through processing (STP) to the firm's back-office and compliance environment.
- **get_trades() -** Receive one or more messages in response to the trade record submitted through save_and_report_trade(). These may track the progress of the trade through the clearing vendor's system by indicating receipt of the trade, notification of account or security errors, and processing or rejection of the trade.

By abstracting trade reporting, the BondPUB API allows strategies to remain focused on execution logic without being entangled in the operational heterogeneity of fixed income markets. This ensures that all executions - whether passive, aggressive, or negotiated - are efficiently processed across one or more clearing vendor relationships.

11 Putting AutoPublish together

The AutoPublish algorithm is intended to automatically publish quotes from an inventory or offering list, manage quoting, handle inbound buy and/or sell orders, and then report the trades to clearing.

In pseudocode, the AutoPublish algorithm might look like this:

- Subscribe to real-time updates
  - `subscribe_to_quote_updates()`
  - `subscribe_to_order_updates()`
  - `subscribe_to_trade_updates()`
- Filter for positions to publish using the Quote model
  - `quote_filter()`
  - `get_quotes()`
- Loop through each position
  - Retrieve the current bond description using the Issue model:
  - `lookup_issue_by_cusip(…)`

    - `get_issue(…)`
    - Retrieve current price for each security using the Price model:
    - `get_prices()`
    - Publish quotes to desired trading destinations
    - `quote_publication(…)`
- For each order received, process using the Order model:
    - `on_order_update()`
    - Reduce quantity and publish updated quotes
    - `update_quote(…) / quote_publication(…)`
    - Evaluate the order price and size
    - Internal logic
    - Accept or Decline on the order
    - `execute_order(…) or update_order_status(…)`
    - For each fill, communicate to clearing using Trade:
    - `Report the trade to clearing using save_and_report_trade(…)`
    - `Receive clearing status updates using on_trade_update(…)`
- For each price update received
    - `Re-publish quotes using quote_publication()`

12 Putting AutoRespond together

The AutoRespond algorithm is intended to automatically quote incoming bids- or offers-wanted, handle the resulting inbound buy and/or sell orders, and then report the trades to clearing.

In pseudocode, the AutoRespond algorithm might look like this:

- Subscribe to real-time updates
    - `subscribe_to_marketRFQ_updates()`
    - `subscribe_to_order_updates()`
    - `subscribe_to_trade_updates()`
- Loop and process incoming RFQs
    - `on_marketRFQ_update()`
    - Retrieve the current bond description using the Issue model:
    - `lookup_issue_by_cusip(…)`
    - `get_issue(…)`
    - Filter the RFQs by configured criteria using MarketRFQFilter.
    - Exit if not matching the filter
    - Retrieve current price for each security using the Price model:
    - `get_prices()`
    - Publish quotes to desired trading destination
    - `quote_publication(…)`
- For each bid- or offer-wanted won, process using the Order model:
    - `on_order_update()`
    - Validate and accept the order
    - `update_order_status()`
    - `execute_order()`
- For each completed order, communicate to clearing using Trade:
    - `Report the trade to clearing using save_and_report_trade(…)`
    - `Receive clearing status updates using on_trade_update(…)`

13 Putting AutoAccumulate together

The AutoAccumulate algorithm automatically monitors aggregated market quote feeds, identifies bonds that satisfy acquisition filters (i.e. CUSIP, Issuer name, etc.), places an order against the market quote, completes the transactions, and then reports the trades to clearing.

In pseudocode, the AutoAccumulate algorithm might look like this:

- Subscribe to real-time updates
  - `subscribe_to_marketQuote_updates()`
  - `subscribe_to_order_updates()`
  - `subscribe_to_trade_updates()`
- Loop and process incoming RFQs
  - `on_marketQuote_update()`
  - Retrieve the current bond description using the Issue model:
  - `lookup_issue_by_cusip(…)`
  - `get_issue(…)`
  - Filter the MarketQuotes by configured criteria using MarketQuoteFilter.
  - Exit if not matching the filter
  - Retrieve current price for each security using the Price model:
  - `get_prices()`
  - Publish quotes to desired trading destination
  - `quote_publication(…)`
- For each bid-wanted won, process using the Order model:
  - `on_order_update()`
  - Validate and accept the order
  - `update_order_status()`
  - `execute_order()`
- For each completed order, communicate to clearing using Trade:
  - `Report the trade to clearing using save_and_report_trade(…)`
  - `Receive clearing status updates using on_trade_update(…)`

14 Putting AutoDivest together

AutoDivest reduces or eliminates positions in specific bonds at the best price. It publishes offer quotes using quote_publication() and, if the position remains unsold over a defined interval, the algorithm gradually reduces the price and sends updates through quote_publication(...). This price-decay model increases execution probability while controlling timing and loss thresholds.

In pseudocode, the AutoDivest algorithm might look like this:

- Subscribe to real-time updates
  - `subscribe_to_quote_updates()`
  - `subscribe_to_order_updates()`
  - `subscribe_to_trade_updates()`
- Filter for positions to divest using the Quote model
  - `quote_filter()`
  - `get_quotes()`
- Loop through each position until sold
  - Retrieve the current bond description using the Issue model:
  - `lookup_issue_by_cusip(…)`
  - `get_issue(…)`
  - Retrieve current price for each security using the Price model:
  - `get_prices()`
  - Evaluate the time since last sale and adjust the aggression factor on the price.
  - Publish quotes to desired trading destinations
  - `quote_publication(…)`
- For each order received, process using the Order model:
  - `on_order_update()`
  - Reduce quantity and publish updated quotes
  - `update_quote(…) / quote_publication(…)`
  - Evaluate the order price and size
  - Internal logic
  - Accept or Decline on the order
  - `execute_order(…) or update_order_status(…)`
  - For each fill, communicate to clearing using Trade:
  - `Report the trade to clearing using save_and_report_trade(…)`

- `Receive clearing status updates using on_trade_update(…)`

15 Putting DealFinder together

The DealFinder algorithm scans incoming market quotes and compares the prices against a source of market prices. It ranks opportunities for purchase based on spread or other user-defined properties. For qualified opportunities, it can route orders through create_order() and order_execute()depending on the committed budget.

In pseudocode, the DealFinder algorithm might look like this:

- Subscribe to real-time updates
  - `subscribe_to_marketQuote_updates()`
  - `subscribe_to_order_updates()`
  - `subscribe_to_trade_updates()`
- Loop and process incoming RFQs
  - `on_marketQuote_update()`
  - Retrieve the current bond description using the Issue model:
  - `lookup_issue_by_cusip(…)`
  - `get_issue(…)`
  - Filter the MarketQuotes by configured criteria using MarketQuoteFilter.
  - Exit if not matching the filter
  - Retrieve current price for each security using the Price model:
  - `get_prices()`
  - Compare and evaluate the bid or offer price of the MarketQuote to the market price from the pricing feed.
  - If the price exceeds transaction thresholds, initiate the order
  - `update_order_status()`
  - `execute_order()`
- For each order accepted, process using the Order model:
  - `on_order_update()`
  - Validate and accept the order
  - `update_order_status()`
  - `execute_order()`
- For each completed order, communicate to clearing using Trade:
  - `Report the trade to clearing using save_and_report_trade(…)`
  - `Receive clearing status updates using on_trade_update(…)`

Each successful transaction resulting from DealFinder could be automatically re-listed with an accurate market price using AutoPublish or AutoDivest.

16 Conclusion and future enhancements

The BondPUB API provides a powerful abstraction layer for automating fixed income trading, enabling firm desks to scale their participation across multiple trading platforms while maintaining control over inventory, pricing, and compliance. Using a unified set of services models - such as those primarily mentioned: Price, Quote, MarketRFQ, and Order market participants may abstract internal and vendor services, streamline quoting, order and RFQ response, and post-trade workflows with greatly reduced integration effort. BondPUB provides other models to round out the capabilities available, such as: Issue, Trade, QuoteFilter, OrderFilter, Trade Filter, Account, OrderStatusHistory, TradeStatusHistory, and much more.

The Python implementation outlined in this paper describes how quote streaming (AutoPublish), RFQ evaluation (AutoRespond), market scanning (DealFinder), accumulation (AutoAccumulate), and

liquidation (AutoDivest) can be expressed as modular strategies operating on real-time data and pricing feeds. Each strategy benefits from the standardization, logging, and risk controls embedded in the BondPUB API design. Each strategy can also take advantage of the BondPUB UI for handling much of the configuration, review, and monitoring functions.

As the market continues to evolve, several future enhancements may extend the functionality and reach of this architecture:

- **Machine Learning Integration -** Algorithms may be enhanced using ML-based inventory scoring, trade outcome prediction, and RFQ prioritization. These models could drive real-time decisions about pricing aggressiveness or liquidity targeting across venues.
- **Latency Optimization -** Planned performance enhancement using ZeroMQ, event batching, and WebSocket multiplexing can further reduce response times for RFQs and streaming quote updates. This will be especially valuable in competitive, high-frequency ATS environments.
- **EMS/OMS Platform Expansion -** Integrating BondPUB with additional buy-side execution platforms and OMSs broadens the flow of RFQs and quotes and creates more opportunities for algorithmic participation in municipal markets.

Together, these enhancements point toward a robust and intelligent trading architecture - one that maintains regulatory discipline while adapting to new market structure realities.

Continue the conversation

FTLabs helps fixed-income firms connect market data, trading venues, pricing, orders, trades and analytics through configurable workflows. Code and API examples in this guide are illustrative and should be validated against current FTLabs documentation before implementation.

Talk with FTLabs

ftlabs.com | (321) 248-4248

10 N. Broadway, Suite 10, Edmond, OK 73034 |