Cryptocurrency Matching Engine - API Specification

---

1. Executive Summary

1.1 Architecture Overview

The Cryptocurrency Matching Engine implements a dual-protocol API architecture combining RESTful HTTP endpoints for order management with WebSocket streaming for real-time market data dissemination. The system is built on Flask with Socket.IO for WebSocket communications, utilizing eventlet for high-concurrency asynchronous processing.

Core Components:

- **REST API Layer**: HTTP endpoints for order submission, cancellation, and system queries
- **WebSocket Layer**: Real-time streaming of market data, trade executions, and order events
- **Matching Engine**: Core order matching logic with price-time priority execution
- **Event System**: Asynchronous event broadcasting to prevent blocking order processing

1.2 Design Principles

Real-Time Data Streaming: All market data updates are pushed via WebSocket connections with sub-second latency targeting. L2 order book updates, trade executions, and order state changes are broadcast immediately upon occurrence.

Low Latency Processing: The system implements per-symbol locking, validation caching, and asynchronous event emission to achieve order processing latencies under 100ms for standard operations.

Comprehensive Error Handling: Every API endpoint returns structured error responses with specific error codes, enabling programmatic error handling. HTTP status codes follow REST conventions (200, 400, 404, 429, 500).

1.3 Authentication and Rate Limiting

Rate Limiting Strategy: Token bucket algorithm with 10,000 requests per 60-second window per IP address (configurable via `RateLimiter` class in app.py:142). Exceeded limits return HTTP 429 with retry-after headers.

CORS Policy: Wildcard origins ("*") enabled for WebSocket connections to support browser-based clients. Production deployments should restrict to specific domains.

Security Considerations: No authentication currently implemented - suitable for development/testing only. Production requires JWT or API key authentication.

---

2. Order Submission API

2.1 POST /order Endpoint

Implementation Location: `api/app.py:269-320`

The primary order submission endpoint accepts JSON payloads containing order specifications and returns execution results including any generated trades.

HTTP Method: POST
Content-Type: application/json
Rate Limited: Yes (10,000 requests/minute)

2.1.1 Request Format

```json
{
  "symbol": "BTC-USDT",
  "side": "buy",
  "order_type": "limit",
  "quantity": "0.001",
  "price": "30000.00",
  "client_order_id": "my-order-123"
}
```

Required Parameters:

- `symbol` (string): Trading pair identifier (e.g., "BTC-USDT", "ETH-USDT")
- `side` (string): Order direction - "buy" or "sell"
- `order_type` (string): Order execution type - "market", "limit", "ioc", "fok"
- `quantity` (string): Order size in base currency (Decimal precision)

Optional Parameters:

- `price` (string): Limit price (required for "limit", "ioc", "fok" orders)
- `client_order_id` (string): Custom order identifier for client tracking

2.1.2 Order Type Specifications

Market Orders: Execute immediately at best available prices. No price parameter required. Partial fills cancel remaining quantity if insufficient liquidity exists.

Limit Orders: Execute at specified price or better. Unfilled portions remain on order book until cancelled or filled. Price parameter mandatory.

Immediate-or-Cancel (IOC): Execute immediately at specified price or better. Cancel any unfilled portion. No resting on order book.

Fill-or-Kill (FOK): Execute entire order immediately at specified price or better, or cancel completely. Validates sufficient liquidity before execution.

2.1.3 Response Format

Successful Submission (HTTP 200):

```
json

{
  "order_id": "550e8400-e29b-41d4-a716-446655440000",
  "client_order_id": "my-order-123",
  "status": "filled",
  "filled_quantity": "0.001000",
  "remaining_quantity": "0.000000",
  "trade_count": 1,
  "processing_time_ms": 2.34
}
```

Response Fields:

- `order_id` (string): Unique system-generated order identifier (UUID)
- `status` (string): Order state - "filled", "partially_filled", "resting", "cancelled", "rejected"
- `filled_quantity` (string): Executed quantity in base currency
- `remaining_quantity` (string): Unfilled quantity remaining
- `trade_count` (integer): Number of trades generated by this order
- `processing_time_ms` (float): Order processing latency in milliseconds

2.1.4 Error Responses

Validation Error (HTTP 400):

```
json
```

```
{
  "error": "Invalid quantity: must be positive",
  "error_type": "OrderValidationError",
  "error_code": "VALIDATION_ERROR",
  "processing_time_ms": 0.12
}
```

Rate Limit Exceeded (HTTP 429):

```
json

{
  "error": "Rate limit exceeded",
  "error_code": "RATE_LIMIT",
  "retry_after": 60
}
```

Internal Error (HTTP 500):

```
json

{
  "error": "Internal server error",
  "error_code": "INTERNAL_ERROR",
  "processing_time_ms": 1.45
}
```

2.1.5 Validation Logic Implementation

The order validation process (implemented in `matcher.py:_validate_order_params_cached`) performs:

1. **Parameter Normalization**: String trimming and case standardization
2. **Type Validation**: Decimal conversion with precision checking
3. **Business Rule Validation**: Symbol-specific limits, minimum quantities, notional values
4. **Order Type Consistency**: Price requirements for limit orders, market order restrictions
5. **Result Caching**: Common parameter combinations cached to reduce CPU overhead

2.2 Order Cancellation API

Endpoint: POST /cancel
Implementation: `api/app.py:349-394`

Request Format:

```
json
```

```
{
  "symbol": "BTC-USDT",
  "order_id": "550e8400-e29b-41d4-a716-446655440000"
}
```

Response Format:

```
json

{
  "success": true,
  "order_id": "550e8400-e29b-41d4-a716-446655440000",
  "processing_time_ms": 1.23
}
```

---

3. Market Data Dissemination API

3.1 L2 Order Book Updates

WebSocket Event: "l2_update"
Namespace: "/market"
Implementation: Connection management in `app.py:ConnectionManager` class

The system broadcasts real-time order book snapshots to subscribed clients whenever book state changes occur due to order additions, cancellations, or executions.

3.1.1 Data Format Specification

```
json

{
  "timestamp": 1640995200.123456,
  "symbol": "BTC-USDT",
  "asks": [
    ["30100.00", "0.500000"],
    ["30105.00", "1.200000"],
    ["30110.00", "0.750000"]
  ],
  "bids": [
    ["30095.00", "0.800000"],
    ["30090.00", "2.100000"],
    ["30085.00", "1.500000"]
  ]
}
```

Field Specifications:

- `timestamp` (float): Unix timestamp with microsecond precision
- `symbol` (string): Trading pair identifier
- `asks` (array): Sell orders sorted by price (ascending), each [price, aggregate_quantity]
- `bids` (array): Buy orders sorted by price (descending), each [price, aggregate_quantity]

3.1.2 Subscription Management

Subscribe to Symbol Updates:

```javascript
socket.emit("subscribe", {
  "type": "l2_updates",
  "symbol": "BTC-USDT"
});
```

Subscription Confirmation:

```json
{
  "type": "l2_updates",
  "symbol": "BTC-USDT",
  "timestamp": 1640995200.123456
}
```

3.1.3 Performance Characteristics

Update Frequency: Immediate broadcast on every order book modification
Latency Target: <50ms from matching engine event to client delivery
Connection Scaling: Supports 1000+ concurrent WebSocket connections per symbol
Message Queuing: Bounded queues prevent memory exhaustion during high-volume periods

3.2 Best Bid/Offer (BBO) API

REST Endpoint: GET /bbo/{symbol}
WebSocket Integration: Included in L2 updates
Implementation: `app.py:448-461`

REST Response Format:

```json
```

```
{
  "symbol": "BTC-USDT",
  "best_bid": "30095.00",
  "best_ask": "30100.00",
  "spread": "5.00",
  "mid_price": "30097.50",
  "timestamp": 1640995200.123456
}
```

3.3 Connection Management Architecture

Implementation: `app.py:ConnectionManager` class (lines 69-149)

The connection manager maintains WebSocket session state and subscription mappings using thread-safe data structures:

```python
self.connections: Dict[str, Dict] = {}  # session_id -> connection_info
self.symbol_subscribers: Dict[str, set] = defaultdict(set)
self.trade_subscribers: set = set()
```

Connection Lifecycle:

1. **Connect**: Client establishes WebSocket connection to `/market` namespace
2. **Subscribe**: Client requests specific data feeds (L2, trades)
3. **Stream**: Server broadcasts relevant updates to subscribed clients
4. **Disconnect**: Automatic cleanup of subscriptions and session state

---

4. Trade Execution Data API

4.1 Real-Time Trade Stream

WebSocket Event: "trade"
Trigger: Automatic emission from matching engine on order execution
Implementation: Event handlers in `app.py:213-228`

4.1.1 Trade Data Format

```json
```

```
{
  "timestamp": 1640995200.123456,
  "symbol": "BTC-USDT",
  "trade_id": "BTC-USDT-123-abc12345",
  "price": "30000.00",
  "quantity": "0.001000",
  "aggressor_side": "buy",
  "maker_order_id": "550e8400-e29b-41d4-a716-446655440000",
  "taker_order_id": "6ba7b810-9dad-11d1-80b4-00c04fd430c8"
}
```


Field Specifications:

- `trade_id` (string): Unique trade identifier with format "{symbol}-{sequence}-{uuid}"
- `aggressor_side` (string): Side of the order that initiated the trade ("buy"/"sell")
- `maker_order_id` (string): UUID of the resting order that provided liquidity
- `taker_order_id` (string): UUID of the incoming order that removed liquidity

4.1.2 Event Handler Architecture

Trade Generation Flow:

1. **Matching Engine**: Executes trade between orders in `book.py:_execute_trade`
2. **Event Emission**: Engine calls registered trade handlers via `matcher.py:EventBus`
3. **WebSocket Broadcast**: `handle_trade` function broadcasts to subscribed clients
4. **Client Processing**: Frontend receives trade and updates UI components

Asynchronous Processing: Trade events are queued in background threads to prevent blocking order processing. Event queue bounded at 10,000 messages to prevent memory issues.

4.1.3 Audit Trail Features

Trade Sequencing: Each symbol maintains monotonic sequence numbers ensuring ordered trade history
Unique Identification: Trade IDs combine symbol, sequence, and UUID components for global uniqueness
Timestamp Precision: Microsecond-precision Unix timestamps for accurate chronological ordering
Order Correlation: Maker/taker order IDs enable complete audit trail reconstruction

4.2 Trade Subscription Management

Subscribe to Trade Feed:

```
javascript
```

```
socket.emit("subscribe", {
  "type": "trades"
});
```

Trade Feed Events: Clients receive all trades across all symbols. Symbol-specific filtering performed client-side for bandwidth efficiency.

---

5. WebSocket Protocol Specification

5.1 Connection Management

Endpoint: `ws://localhost:5000/socket.io/`
Namespace: `/market`
Transport: WebSocket preferred, polling fallback
Implementation: Socket.IO 4.5.4 with eventlet async mode

5.1.1 Connection Flow

1. **Initial Connection**:

```
javascript

  const socket = io("/market", {
    path: "/socket.io/",
    transports: ['websocket', 'polling'],
    timeout: 5000
  });
```

2. **Welcome Message**:

```
json

  {
    "status": "connected",
    "session_id": "abc123def456",
    "timestamp": 1640995200.123456,
    "symbols": ["BTC-USDT", "ETH-USDT", "BNB-USDT", "SOL-USDT"]
  }
```

5.1.2 Heartbeat Protocol

Client Ping:

```
javascript
```

```
socket.emit("ping");
```

Server Pong:

```
json

{
  "timestamp": 1640995200.123456
}
```

Interval: 30-second client-initiated pings for connection health monitoring

5.2 Subscription Model

Available Subscription Types:

- `l2_updates`: Real-time order book snapshots for specific symbols
- `trades`: Global trade execution stream across all symbols
- `order_events`: Order state change notifications (filled, cancelled, etc.)

Subscription Request Format:

```
javascript

socket.emit("subscribe", {
  "type": "l2_updates",
  "symbol": "BTC-USDT"  // Optional for symbol-specific feeds
});
```

5.3 Error Handling

Connection Errors: Automatic reconnection with exponential backoff (5 attempts max)
Subscription Errors: Invalid subscription requests return error events with diagnostic messages
Rate Limiting: WebSocket connections subject to same IP-based rate limits as REST endpoints

---

6. REST API Reference

6.1 System Health and Information

GET /health

Purpose: System status and operational metrics
Rate Limited: No

Response Format:

```json
{
  "status": "healthy",
  "timestamp": 1640995200.123456,
  "uptime": 86400.5,
  "version": "2.0.0",
  "connections": 42
}
```

GET /symbols

Purpose: Available trading pairs and market information
Implementation: `app.py:245-268`

Response Format:

```json
{
  "symbols": ["BTC-USDT", "ETH-USDT", "BNB-USDT", "SOL-USDT"],
  "symbol_info": {
    "BTC-USDT": {
      "bbo": {
        "best_bid": "30095.00",
        "best_ask": "30100.00"
      },
      "active_orders": 15,
      "total_trades": 1248
    }
  },
  "count": 4
}
```

6.2 Market Data Endpoints

GET /book/{symbol}

Purpose: L2 order book snapshot
Parameters: `?levels=10` (optional, max 100)

Response Format:

```json
```

```
{
  "timestamp": 1640995200.123456,
  "symbol": "BTC-USDT",
  "bids": [["30095.00", "0.800000"]],
  "asks": [["30100.00", "0.500000"]]
}
```


GET /statistics

Purpose: Engine performance metrics
Parameters: `?symbol=BTC-USDT` (optional)

Response Format:


```
json

{
  "uptime_seconds": 86400.5,
  "orders_processed": 15420,
  "trades_executed": 6234,
  "symbols_count": 4,
  "orders_per_second": 178.47,
  "validation_cache_hits": 12450,
  "error_counts": {
    "OrderValidationError": 23,
    "InsufficientLiquidityError": 5
  }
}
```


6.3 Order Management

GET /order/{order_id}

Purpose: Order status query
Parameters: `?symbol=BTC-USDT` (required)

Response Format:

```
json
```

```
{
  "order_id": "550e8400-e29b-41d4-a716-446655440000",
  "symbol": "BTC-USDT",
  "status": "partially_filled",
  "filled_quantity": "0.000500",
  "remaining_quantity": "0.000500"
}
```


6.4 Error Response Format

All endpoints return consistent error structures:


```
json

{
  "error": "Human-readable error description",
  "error_code": "MACHINE_READABLE_CODE",
  "detail": "Additional technical details",
  "processing_time_ms": 1.23
}
```


Standard Error Codes:

- `VALIDATION_ERROR`: Invalid request parameters
- `NOT_FOUND`: Resource does not exist
- `RATE_LIMIT`: Request rate exceeded
- `INTERNAL_ERROR`: Server-side processing failure

---

7. Client Integration Examples

7.1 JavaScript WebSocket Client

The reference implementation in `main.js` demonstrates complete integration:


```
javascript
```

```javascript
// Connection establishment
const socket = io("/market", {
  path: "/socket.io/",
  transports: ['websocket', 'polling'],
  timeout: 5000
});

// Market data subscription
socket.emit("subscribe", {
  type: "l2_updates",
  symbol: "BTC-USDT"
});

// Trade feed subscription
socket.emit("subscribe", {
  type: "trades"
});

// Event handlers
socket.on("l2_update", (data) => {
  updateOrderBook(data.bids, data.asks);
});

socket.on("trade", (trade) => {
  displayTrade(trade);
  playTradeSound();
});
```

7.2 Order Submission Example

javascript

```javascript
async function submitOrder(orderData) {
  const response = await fetch('/order', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({
      symbol: "BTC-USDT",
      side: "buy",
      order_type: "limit",
      quantity: "0.001",
      price: "30000.00"
    })
  });

  const result = await response.json();
  if (response.ok) {
    console.log(`Order submitted: ${result.order_id}`);
  } else {
    console.error(`Order failed: ${result.error}`);
  }
}
```

7.3 Rate Limiting Considerations

Clients should implement exponential backoff for HTTP 429 responses:

javascript

```javascript
async function submitOrderWithRetry(orderData, maxRetries = 3) {
  for (let i = 0; i < maxRetries; i++) {
    const response = await fetch('/order', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(orderData)
    });

    if (response.status !== 429) {
      return response.json();
    }

    // Exponential backoff
    await new Promise(resolve =>
      setTimeout(resolve, Math.pow(2, i) * 1000)
    );
  }
  throw new Error('Max retries exceeded');
}
```

8. Technical Implementation Notes

8.1 Performance Optimizations

Per-Symbol Locking: Individual locks per trading pair prevent global contention (implemented in optimized `matcher.py`)
Validation Caching: Common parameter combinations cached to reduce parsing overhead
Asynchronous Event Emission: Trade and order events processed in background threads via `EventBus`
Response Minimization: Order responses contain only essential fields to reduce serialization time

8.2 Scalability Considerations

Connection Pooling: WebSocket connections managed per-symbol to distribute load
Message Queuing: Bounded queues prevent memory exhaustion during traffic spikes
Rate Limiting: Configurable per-IP limits protect against abuse
Horizontal Scaling: Stateless design enables load balancer distribution

8.3 Reliability Features

Graceful Degradation: API endpoints remain functional during WebSocket issues
Error Recovery: Automatic reconnection logic with exponential backoff
Audit Logging: Comprehensive logging of all order and trade events
Health Monitoring: `/health` endpoint provides operational visibility

---

Document End

This specification covers the complete API interface for the high-performance cryptocurrency matching engine, providing developers with all necessary information for successful integration and operation.