Building a TradingView-Quality Futures Charting Webapp with IB Gateway

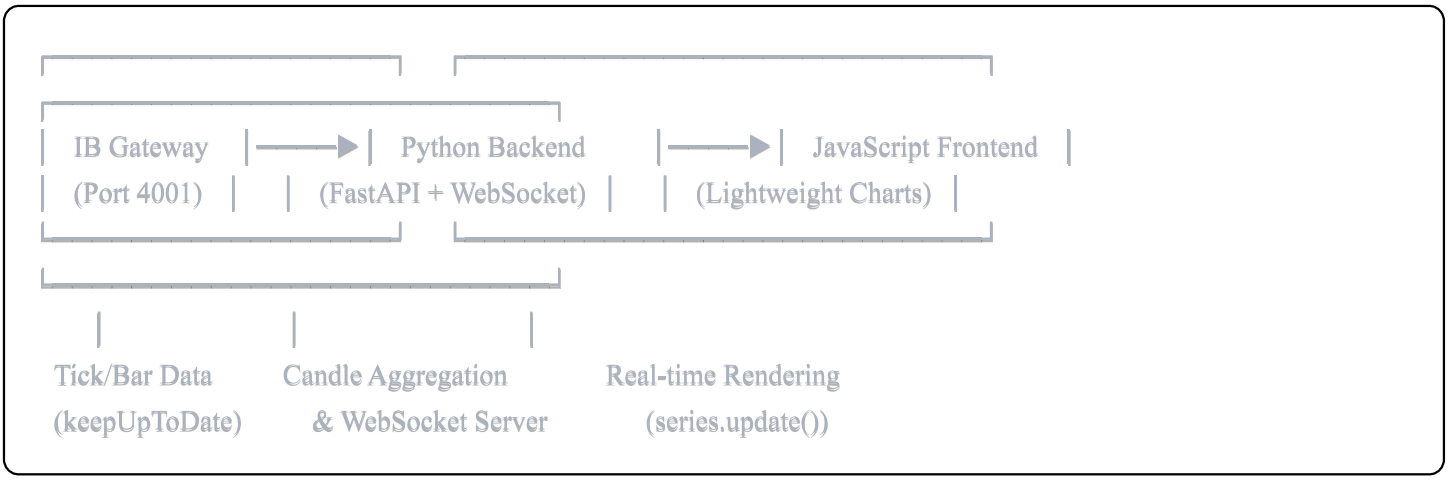
TradingView's own Lightweight Charts library paired with ib_insync provides a production-ready path to a professional futures charting webapp. This implementation guide delivers exact contract specifications for MNQ, MGC, and MES futures, working code for 1-year historical data with real-time streaming, and a plugin architecture for custom indicators—all running as a clean localhost application.

The architecture connects IB Gateway through a Python FastAPI backend via WebSocket to a JavaScript frontend using TradingView's open-source Lightweight Charts library. This combination matches TradingView's visual quality exactly (it's their own code) while handling **375,000+ candles** with sub-5ms real-time updates. Index.dev

---

Architecture overview and technology stack

The system uses three layers: IB Gateway provides market data, a Python service handles connection management and candlestick aggregation, and the browser renders charts.



Technology choices:

- **IB Connection**: ib_insync (or its successor ib_async) — synchronous/async Python wrapper
- **Backend**: FastAPI with native async WebSocket support FastAPI Orchestra
- **Frontend**: TradingView Lightweight Charts v5.x — **45KB**, Canvas-based, TradingView handles 60K+ candles at 60 FPS Index.dev
- **Indicators**: pandas-ta for calculations, plugin architecture for extensibility

---

IB Gateway connection and futures contract specifications

Contract definitions for micro futures

Each futures contract requires specific parameters. These are the exact specifications for MNQ, MES, and MGC:

```python
from ib_insync import IB, Future, ContFuture

# MNQ - Micro E-mini Nasdaq-100
mnq = Future(
    symbol='MNQ',
    exchange='CME',
    currency='USD',
    lastTradeDateOrContractMonth='202503'  # March 2025 expiry
)

# MES - Micro E-mini S&P 500
mes = Future(
    symbol='MES',
    exchange='CME',
    currency='USD',
    lastTradeDateOrContractMonth='202503'
)

# MGC - Micro Gold (note: COMEX exchange, not CME)
mgc = Future(
    symbol='MGC',
    exchange='COMEX',
    currency='USD',
    lastTradeDateOrContractMonth='202502'
)

# For historical data spanning multiple expirations, use continuous contracts:
mnq_continuous = ContFuture(symbol='MNQ', exchange='CME', currency='USD')
```

Contract	Exchange	Multiplier	Tick Size	Tick Value
MNQ	CME/GLOBEX	$2/point	0.25	$0.50 Mrci
MES	CME/GLOBEX	$5/point	0.25	$1.25 Tradervps
MGC	COMEX	10 oz	$0.10/oz	$1.00 QuantVPS

Connection setup

```python
from ib_insync import IB, util

# For Jupyter compatibility
util.startLoop()

ib = IB()

# Connection ports:
# IB Gateway Live: 4001 | IB Gateway Paper: 4002
# TWS Live: 7496 | TWS Paper: 7497

ib.connect(
    host='127.0.0.1',
    port=4002,       # Paper trading Gateway
    clientId=1,      # Must be unique per connection
    timeout=10
)

# Verify contract validity
ib.qualifyContracts(mnq)  # Populates conId and other fields
```

Required TWS/Gateway settings: Enable API connections in File → Global Configuration → API → Settings. AlgoTrading101 Check "Enable ActiveX and Socket Clients" AlgoTrading101 and set the appropriate port. Elite Trader +2

Fetching one year of historical 1-minute data

IB imposes rate limits: **maximum ~2,000 bars per request** for 1-minute data. Fetching one year requires chunking into daily requests with pacing delays.

Historical data request parameters

python

```
# Single day request (optimal chunk size)
bars = ib.reqHistoricalData(
    contract=mnq,
    endDateTime='',          # Empty = current time
    durationStr='1 D',       # 1 day of data
    barSizeSetting='1 min',  # 1-minute bars
    whatToShow='TRADES',     # OHLCV from actual trades
    useRTH=0,                # 0 = include extended hours (essential for futures)
    formatDate=1,            # yyyyMMdd HH:mm:ss format
    keepUpToDate=False       # False for historical-only
)

# Each bar contains:
# bar.date, bar.open, bar.high, bar.low, bar.close, bar.volume, bar.wap, bar.barCount
```

Full year download with rate limiting

python

```python
from datetime import datetime, timedelta
import time
import pandas as pd

def download_year_of_minute_data(ib, contract, end_date=None):
    """Download 1 year of 1-minute bars with proper rate limiting."""

    all_bars = []
    end = end_date or datetime.now()
    start = end - timedelta(days=365)
    current_end = end
    request_count = 0

    while current_end > start:
        try:
            bars = ib.reqHistoricalData(
                contract,
                endDateTime=current_end.strftime('%Y%m%d-%H:%M:%S'),
                durationStr='1 D',
                barSizeSetting='1 min',
                whatToShow='TRADES',
                useRTH=0,
                timeout=120
            )

            if bars:
                all_bars = bars + all_bars  # Prepend older data
                # Move to day before first bar received
                first_bar_time = datetime.strptime(bars[0].date, '%Y%m%d  %H:%M:%S')
                current_end = first_bar_time - timedelta(minutes=1)
            else:
                current_end -= timedelta(days=1)

            request_count += 1

            # Rate limiting: 6 requests per 10 seconds max
            if request_count % 5 == 0:
                time.sleep(2)

            # Longer pause every 50 requests (avoid pacing violations)
            if request_count % 50 == 0:
                print(f"Fetched {len(all_bars)} bars, pausing...")
                time.sleep(60)
```

```
        except Exception as e:
            print(f"Error: {e}, retrying in 10s...")
            time.sleep(10)

    # Convert to DataFrame
    df = util.df(all_bars)
    df['time'] = pd.to_datetime(df['date']).astype(int) // 10**9  # Unix timestamp
    return df[['time', 'open', 'high', 'low', 'close', 'volume']]

# Usage
ib.qualifyContracts(mnq)
historical_df = download_year_of_minute_data(ib, mnq)
historical_df.to_parquet('mnq_1min_1year.parquet')  # Cache locally
```

Real-time streaming with live candlestick formation

The key to TradingView-style live candle updates is keepUpToDate=True, which streams bar updates as they form. Interactivebrokers

The keepUpToDate pattern

python

```
from ib_insync import IB, Future
import asyncio

class RealtimeBars:
    def __init__(self, ib, contract, on_bar_callback):
        self.ib = ib
        self.contract = contract
        self.on_bar = on_bar_callback
        self.bars = None

    def start(self):
        """Start streaming real-time bars."""
        self.bars = self.ib.reqHistoricalData(
            self.contract,
            endDateTime='',
            durationStr='1 D',       # Load 1 day of history initially
            barSizeSetting='1 min',
            whatToShow='TRADES',
            useRTH=0,
            keepUpToDate=True        # ← This enables real-time streaming
        )

        # Subscribe to bar updates
        self.bars.updateEvent += self._on_update

    def _on_update(self, bars, hasNewBar):
        """Called on every bar update (multiple times per minute)."""
        current_bar = bars[-1]
        bar_data = {
            'time': int(current_bar.date.timestamp()) if hasattr(current_bar.date, 'timestamp')
                    else int(datetime.strptime(current_bar.date, '%Y%m%d  %H:%M:%S').timestamp()),
            'open': current_bar.open,
            'high': current_bar.high,
            'low': current_bar.low,
            'close': current_bar.close,
            'volume': int(current_bar.volume)
        }
        self.on_bar(bar_data, is_new_bar=hasNewBar)

    def stop(self):
        if self.bars:
            self.ib.cancelHistoricalData(self.bars)
```

This fires `_on_update` every time the current bar's OHLCV changes—typically several times per second during active trading. The `hasNewBar` flag indicates when a new minute begins. `ib_insync`

FastAPI WebSocket backend

The backend bridges IB Gateway to browser clients via WebSocket.

```python
```

```python
# backend/app.py
from fastapi import FastAPI, WebSocket, WebSocketDisconnect
from fastapi.staticfiles import StaticFiles
from fastapi.responses import FileResponse
import asyncio
import json
from typing import Set
from contextlib import asynccontextmanager
from ib_insync import IB, Future

# Connection management
class ConnectionManager:
    def __init__(self):
        self.active_connections: Set[WebSocket] = set()

    async def connect(self, websocket: WebSocket):
        await websocket.accept()
        self.active_connections.add(websocket)

    def disconnect(self, websocket: WebSocket):
        self.active_connections.discard(websocket)

    async def broadcast(self, message: dict):
        disconnected = set()
        for connection in self.active_connections:
            try:
                await connection.send_json(message)
            except:
                disconnected.add(connection)
        self.active_connections -= disconnected

manager = ConnectionManager()
ib = IB()
realtime_bars = {}

@asynccontextmanager
async def lifespan(app: FastAPI):
    # Startup
    await ib.connectAsync('127.0.0.1', 4002, clientId=1)
    yield
    # Shutdown
    ib.disconnect()
```

```
app = FastAPI(lifespan=lifespan)
app.mount("/static", StaticFiles(directory="frontend/static"), name="static")

@app.get("/")
async def root():
    return FileResponse("frontend/templates/index.html")

@app.websocket("/ws/{symbol}")
async def websocket_endpoint(websocket: WebSocket, symbol: str):
    await manager.connect(websocket)

    # Create contract
    contract = Future(symbol=symbol, exchange='CME', currency='USD',
                      lastTradeDateOrContractMonth='202503')
    ib.qualifyContracts(contract)

    # Callback for bar updates
    async def on_bar_update(bar_data, is_new_bar):
        await websocket.send_json({
            'type': 'bar_update',
            'data': bar_data,
            'is_new_bar': is_new_bar
        })

    # Start real-time bars
    bars_handler = RealtimeBars(ib, contract,
                                lambda data, is_new: asyncio.create_task(on_bar_update(data, is_new)))
    bars_handler.start()

    # Send initial historical data
    historical = [
        {
            'time': int(b.date.timestamp()) if hasattr(b.date, 'timestamp') else 0,
            'open': b.open, 'high': b.high, 'low': b.low,
            'close': b.close, 'volume': int(b.volume)
        }
        for b in bars_handler.bars[:-1]  # All except forming bar
    ]
    await websocket.send_json({'type': 'historical', 'data': historical})

    try:
        while True:
            await websocket.receive_text()  # Keep alive
    except WebSocketDisconnect:
```

```
    bars_handler.stop()
    manager.disconnect(websocket)
```

Frontend with TradingView Lightweight Charts

The frontend uses Lightweight Charts directly for maximum visual fidelity and control.

Complete HTML/JavaScript implementation

```html
```

```
<!-- frontend/templates/index.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Futures Chart</title>
    <style>
        * { margin: 0; padding: 0; box-sizing: border-box; }
        body {
            background-color: #131722;
            font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto, sans-serif;
        }
        #chart { width: 100vw; height: 100vh; }
        .status {
            position: absolute; top: 10px; left: 10px;
            color: #758696; font-size: 12px; z-index: 100;
        }
        .status.connected { color: #26a69a; }
        .status.disconnected { color: #ef5350; }
    </style>
</head>
<body>
    <div id="status" class="status">Connecting...</div>
    <div id="chart"></div>

    <script type="module">
        import { createChart, ColorType } from 'https://unpkg.com/lightweight-charts@5.0.0/dist/lightweight-charts.standalone.

        // ═══════════════════════════════════════════════════════════════
        // TRADINGVIEW EXACT COLOR SCHEME
        // ═══════════════════════════════════════════════════════════════
        const COLORS = {
            UP: '#26a69a',              // TradingView green
            DOWN: '#ef5350',            // TradingView red
            VOLUME_UP: 'rgba(38, 166, 154, 0.5)',
            VOLUME_DOWN: 'rgba(239, 83, 80, 0.5)',
            BACKGROUND: '#131722',
            TEXT: '#d1d4dc',
            GRID: '#1e222d',
            CROSSHAIR: '#758696'
        };

        // ═══════════════════════════════════════════════════════════════
        // CHART INITIALIZATION
```

```
// ═══════════════════════════════════════════════════════════════
const container = document.getElementById('chart');

const chart = createChart(container, {
  width: container.clientWidth,
  height: container.clientHeight,
  layout: {
    background: { type: ColorType.Solid, color: COLORS.BACKGROUND },
    textColor: COLORS.TEXT,
  },
  grid: {
    vertLines: { color: COLORS.GRID },
    horzLines: { color: COLORS.GRID },
  },
  crosshair: {
    vertLine: { color: COLORS.CROSSHAIR, labelBackgroundColor: '#2a2e39' },
    horzLine: { color: COLORS.CROSSHAIR, labelBackgroundColor: '#2a2e39' },
  },
  rightPriceScale: {
    borderColor: '#2a2e39',
    scaleMargins: { top: 0.1, bottom: 0.2 },
  },
  timeScale: {
    borderColor: '#2a2e39',
    timeVisible: true,
    secondsVisible: false,
  },
});

// Candlestick series
const candleSeries = chart.addCandlestickSeries({
  upColor: COLORS.UP,
  downColor: COLORS.DOWN,
  borderVisible: false,
  wickUpColor: COLORS.UP,
  wickDownColor: COLORS.DOWN,
});

// Volume series (overlay at bottom)
const volumeSeries = chart.addHistogramSeries({
  priceFormat: { type: 'volume' },
  priceScaleId: '',
});
volumeSeries.priceScale().applyOptions({
```

```
    scaleMargins: { top: 0.85, bottom: 0 },
});

// Responsive resize
window.addEventListener('resize', () => {
    chart.applyOptions({
        width: container.clientWidth,
        height: container.clientHeight
    });
});

// ═══════════════════════════════════════════════════════════════
// WEBSOCKET CONNECTION
// ═══════════════════════════════════════════════════════════════
const symbol = 'MNQ';  // Change to MES or MGC as needed
let ws;
let reconnectAttempts = 0;

function connect() {
    ws = new WebSocket(`ws://localhost:8000/ws/${symbol}`);

    ws.onopen = () => {
        document.getElementById('status').textContent = `${symbol} Connected`;
        document.getElementById('status').className = 'status connected';
        reconnectAttempts = 0;
    };

    ws.onmessage = (event) => {
        const msg = JSON.parse(event.data);

        if (msg.type === 'historical') {
            // Load historical data
            candleSeries.setData(msg.data);

            // Set volume with colors
            const volumeData = msg.data.map(bar => ({
                time: bar.time,
                value: bar.volume,
                color: bar.close >= bar.open ? COLORS.VOLUME_UP : COLORS.VOLUME_DOWN
            }));
            volumeSeries.setData(volumeData);

            chart.timeScale().fitContent();
        }
```

```
            if (msg.type === 'bar_update') {
                const bar = msg.data;

                // Update candlestick (handles both new bars and updates)
                candleSeries.update(bar);

                // Update volume
                volumeSeries.update({
                    time: bar.time,
                    value: bar.volume,
                    color: bar.close >= bar.open ? COLORS.VOLUME_UP : COLORS.VOLUME_DOWN
                });
            }
        };

        ws.onclose = () => {
            document.getElementById('status').textContent = 'Disconnected - Reconnecting...';
            document.getElementById('status').className = 'status disconnected';

            // Exponential backoff reconnection
            const delay = Math.min(1000 * Math.pow(2, reconnectAttempts), 30000);
            reconnectAttempts++;
            setTimeout(connect, delay);
        };

        ws.onerror = (error) => console.error('WebSocket error:', error);
    }

    connect();
  </script>
</body>
</html>
```

Custom indicator system with plugin architecture

The indicator system calculates values server-side (for performance with large datasets) and sends them to the frontend for rendering.

Indicator base class and implementations

python

```python
# backend/indicators/base.py
from abc import ABC, abstractmethod
import pandas as pd
from typing import Dict, Any, List

class Indicator(ABC):
    """Base class for all indicators."""

    def __init__(self, params: Dict[str, Any]):
        self.params = params
        self.name = self.__class__.__name__

    @abstractmethod
    def calculate(self, df: pd.DataFrame) -> pd.DataFrame:
        """Calculate indicator values. Returns DataFrame with 'time' column."""
        pass

    @property
    @abstractmethod
    def plot_config(self) -> Dict[str, Any]:
        """Return plotting configuration."""
        pass

# backend/indicators/moving_averages.py
class SMA(Indicator):
    """Simple Moving Average"""

    def calculate(self, df: pd.DataFrame) -> pd.DataFrame:
        period = self.params.get('period', 20)
        return pd.DataFrame({
            'time': df['time'],
            'value': df['close'].rolling(window=period).mean()
        }).dropna()

    @property
    def plot_config(self):
        return {
            'type': 'line',
            'color': self.params.get('color', '#2962FF'),
            'lineWidth': 2,
            'pane': 'main'
        }
```

```python
class EMA(Indicator):
    """Exponential Moving Average"""

    def calculate(self, df: pd.DataFrame) -> pd.DataFrame:
        period = self.params.get('period', 20)
        return pd.DataFrame({
            'time': df['time'],
            'value': df['close'].ewm(span=period, adjust=False).mean()
        }).dropna()

    @property
    def plot_config(self):
        return {
            'type': 'line',
            'color': self.params.get('color', '#FF6D00'),
            'lineWidth': 2,
            'pane': 'main'
        }

# backend/indicators/oscillators.py
class RSI(Indicator):
    """Relative Strength Index"""

    def calculate(self, df: pd.DataFrame) -> pd.DataFrame:
        period = self.params.get('period', 14)
        delta = df['close'].diff()
        gain = delta.where(delta > 0, 0).rolling(window=period).mean()
        loss = (-delta.where(delta < 0, 0)).rolling(window=period).mean()
        rs = gain / loss
        rsi = 100 - (100 / (1 + rs))

        return pd.DataFrame({
            'time': df['time'],
            'value': rsi
        }).dropna()

    @property
    def plot_config(self):
        return {
            'type': 'line',
            'color': self.params.get('color', '#7B1FA2'),
            'lineWidth': 2,
            'pane': 'separate',  # Render in separate pane below price
            'priceScaleId': 'rsi',
```

```
            'levels': [30, 70]   # Overbought/oversold lines
        }

class MACD(Indicator):
    """Moving Average Convergence Divergence"""

    def calculate(self, df: pd.DataFrame) -> pd.DataFrame:
        fast = self.params.get('fast', 12)
        slow = self.params.get('slow', 26)
        signal = self.params.get('signal', 9)

        ema_fast = df['close'].ewm(span=fast, adjust=False).mean()
        ema_slow = df['close'].ewm(span=slow, adjust=False).mean()
        macd_line = ema_fast - ema_slow
        signal_line = macd_line.ewm(span=signal, adjust=False).mean()
        histogram = macd_line - signal_line

        return pd.DataFrame({
            'time': df['time'],
            'macd': macd_line,
            'signal': signal_line,
            'histogram': histogram
        }).dropna()

    @property
    def plot_config(self):
        return {
            'type': 'macd',
            'macdColor': '#2962FF',
            'signalColor': '#FF6D00',
            'histogramUpColor': 'rgba(38, 166, 154, 0.5)',
            'histogramDownColor': 'rgba(239, 83, 80, 0.5)',
            'pane': 'separate'
        }
```

Indicator registry and manager

python

```python
# backend/indicators/manager.py
from typing import Dict, Type, List
import pandas as pd

INDICATOR_REGISTRY: Dict[str, Type[Indicator]] = {
    'sma': SMA,
    'ema': EMA,
    'rsi': RSI,
    'macd': MACD,
}

class IndicatorManager:
    """Manages indicator calculations and updates."""

    def __init__(self):
        self.active_indicators: List[Indicator] = []

    def add_indicator(self, indicator_type: str, params: dict) -> Indicator:
        indicator_class = INDICATOR_REGISTRY.get(indicator_type.lower())
        if not indicator_class:
            raise ValueError(f"Unknown indicator: {indicator_type}")

        indicator = indicator_class(params)
        self.active_indicators.append(indicator)
        return indicator

    def calculate_all(self, df: pd.DataFrame) -> Dict[str, dict]:
        """Calculate all active indicators and return serializable results."""
        results = {}
        for indicator in self.active_indicators:
            name = f"{indicator.name}_{indicator.params.get('period', '')}"
            calc_df = indicator.calculate(df)
            results[name] = {
                'data': calc_df.to_dict('records'),
                'config': indicator.plot_config
            }
        return results

    def update_indicator(self, indicator: Indicator, new_bar: dict, df: pd.DataFrame) -> dict:
        """Incrementally update a single indicator with new bar data."""
        # For efficiency, recalculate only the last few values
        calc_df = indicator.calculate(df.tail(100))
        if len(calc_df) > 0:
```

```
        return calc_df.iloc[-1].to_dict()
    return None
```

Frontend indicator rendering

javascript

```javascript
// Add to the frontend JavaScript

class IndicatorRenderer {
    constructor(chart) {
        this.chart = chart;
        this.series = {};  // indicator_name -> series
    }

    addIndicator(name, config, data) {
        if (config.pane === 'main') {
            // Overlay on main chart
            const series = this.chart.addLineSeries({
                color: config.color,
                lineWidth: config.lineWidth || 2,
                priceLineVisible: false,
                lastValueVisible: false,
            });
            series.setData(data.map(d => ({ time: d.time, value: d.value })));
            this.series[name] = series;
        }

        if (config.pane === 'separate' && config.type === 'line') {
            // Separate pane (e.g., RSI)
            const series = this.chart.addLineSeries({
                color: config.color,
                lineWidth: config.lineWidth || 2,
                pane: 1,  // Separate pane
            });
            series.setData(data.map(d => ({ time: d.time, value: d.value })));
            this.series[name] = series;

            // Add reference lines (e.g., 30/70 for RSI)
            if (config.levels) {
                config.levels.forEach(level => {
                    // Could add horizontal line markers here
                });
            }
        }
    }

    updateIndicator(name, newPoint) {
        if (this.series[name]) {
            this.series[name].update(newPoint);
```

```
        }
    }
}

// Usage in WebSocket handler:
// const indicatorRenderer = new IndicatorRenderer(chart);
// When receiving indicator data:
// indicatorRenderer.addIndicator('SMA_20', config, data);
```

Complete project structure and startup script

```
futures-chart/
├── run.py                  # Single-command startup
├── requirements.txt
├── config.yaml
├── backend/
│   ├── __init__.py
│   ├── app.py              # FastAPI application
│   ├── ib_service.py       # IB Gateway connection handling
│   └── indicators/
│       ├── __init__.py
│       ├── base.py
│       ├── moving_averages.py
│       └── oscillators.py
├── frontend/
│   ├── templates/
│   │   └── index.html
│   └── static/
│       └── css/
│           └── style.css
└── data/
    └── cache/              # Historical data cache (parquet files)
```

requirements.txt

```
ib_insync>=0.9.86
fastapi>=0.109.0
uvicorn[standard]>=0.27.0
websockets>=12.0
pandas>=2.0.0
```

```
pyarrow>=14.0.0
pyyaml>=6.0
```

Single-command startup

```python
#!/usr/bin/env python
# run.py
import uvicorn
import webbrowser
import threading
import time

def open_browser():
    time.sleep(1.5)  # Wait for server to start
    webbrowser.open('http://localhost:8000')

if __name__ == '__main__':
    # Open browser in background thread
    threading.Thread(target=open_browser, daemon=True).start()

    # Start server
    uvicorn.run(
        "backend.app:app",
        host="0.0.0.0",
        port=8000,
        reload=False,
        log_level="info"
    )
```

Launch the application:

```bash
python run.py
```

Performance optimization for 375,000 candles

Lightweight Charts v5.1 introduced **data conflation** for large datasets. (TradingView) Enable it for smooth scrolling through a year of minute data:

```javascript
const chart = createChart(container, {
  // ... other options
  timeScale: {
    // Reduces rendered points when zoomed out
    uniformDistribution: true,
  },
});
```

Additional strategies:

- **Lazy loading**: Load only 5,000–10,000 most recent candles initially; fetch more on scroll
- **Local caching**: Store historical data in parquet format; only fetch new bars on startup
- **Typed arrays**: Use `Float64Array` for price storage in memory-intensive scenarios
- **Batch updates**: During fast markets, batch WebSocket messages to reduce render cycles

---

Rate limits and subscription requirements

Constraint	Limit
Historical requests	6 per 2 seconds, 60 per 10 minutes
Identical requests	15-second wait required
Max bars per request	~2,000 for 1-minute data
Concurrent market data lines	100 (default)

Required market data subscriptions (via IB Account Management):

- CME Real-Time Data: MES, MNQ (~$10/month non-professional)
- COMEX Real-Time Data: MGC (~$10/month non-professional)

Conclusion

This implementation provides a production-ready path to TradingView-quality charting. The **ib_insync** + **keepUpToDate=True** pattern delivers true real-time candlestick formation with sub-second updates, while

Lightweight Charts guarantees visual parity with TradingView at a fraction of the complexity of building from scratch.

Key architectural decisions that drive success: chunk historical requests into 1-day segments with proper pacing, use WebSocket for sub-100ms data delivery to the frontend, and leverage Lightweight Charts' native `update()` method which handles both new bars and in-progress bar updates automatically. The indicator plugin system allows unlimited extensibility while keeping the core implementation clean.

For an experienced quant with WebSocket and API experience, expect **2–3 days** to a fully functional implementation, with most time spent on IB Gateway configuration and historical data backfill optimization.