



<Context>

Last time we agreed upon the tech stack for this project:

Backend: Node.js + Express

Frontend: Vue.js + chart.js

Scheduler: nodechron

Database: SQLite

Stock API: Financial Modeling Prep

Hosting: Railway

</Context>

After reviewing different Stock API Alternatives, I realized that free tiers are quite restrictive for my intention to follow prices of passive long-term investments primarily driven by ETFs and some stocks in global markets. To access broad market instruments, you are likely to pay at least 20USD monthly, which is not viable for a future MVP based on this PoC.

<Solution>

I decided to choose yfinance python library to mitigate those restrictions in my PoC, which means I need to move my stack to a python based backend.

Given all the context I provided to you and including attachments of previous technical reports and planning. Following the same PoC goals of : * Simplicity of requirements * 0\$ or near 0\$ cost of development * Relaxed requirements around delayed prices for long investors (no trading requirements, complex technical analysis data, nor deep historical data) * Imperfect PoC in favor of faster development What are your technical suggestions of a Python based backend What are your technical suggestions for a frontend? (keep Vue.js as separated frontnd project? make a full stack python project with Streamlit , ReactPY>) Always give pros and cons on each of your suggestions.

Perfect. I have your full context. Now let me provide Python-based backend recommendations while staying true to your POC goals and the technical language preference stated in your space instructions.

Python Backend Options for Your POC

Given your constraint (yfinance requires Python), here's the technical breakdown:

Option A: FastAPI (★ Recommended)

Technical fit for yfinance:

- Native async support (`async def` endpoints) pairs well with yfinance's I/O patterns (network calls to fetch price data)
- Can wrap yfinance calls in thread pool if needed (yfinance has some blocking calls), using `asyncio.run_in_executor()`
- Built-in OpenAPI docs auto-generated from type hints

Pros:

- Fast (~3x faster than Flask) due to Uvicorn ASGI server; matters if you scale polling
- Type hints throughout codebase reduce bugs in financial calculations
- Minimal boilerplate; Express-like simplicity
- Python libraries shine: pandas for ETF/stock data manipulation, numpy for calculations
- Deployment footprint: ~60-100 MB RAM (vs Node.js ~80 MB, so comparable)

Cons:

- Younger ecosystem than Django; fewer third-party integrations
- Less mature ORM tooling (but SQLAlchemy + Alembic migrations handle it)
- Requires ASGI awareness (Uvicorn + Gunicorn setup can feel less "magical" than Node's single server model)

Verdict: Best for "fast imperfect" Python. Highest performance-to-simplicity ratio.

Option B: Flask + APScheduler

Technical fit for yfinance:

- Flask is simpler but synchronous by default; APScheduler handles the polling loop separately
- yfinance works seamlessly (no async wrapping needed)
- Familiar pattern if you know minimal Python web frameworks

Pros:

- Lighter learning curve; Flask is incredibly simple (~200 lines minimal app)
- APScheduler integrated directly into Flask app; easy to understand
- Rock-solid for single-purpose POCs
- Memory footprint: ~40-60 MB RAM

Cons:

- Synchronous Flask routes block on I/O; if you later add many concurrent users, becomes bottleneck

- APScheduler inside single process = single point of failure (same as node-cron, acceptable for POC)
- Less "modern" Python (no type hints by default)

Verdict: Safest bet if you want ultra-simple. Trade off some flexibility for less boilerplate.

Option C: Quart (Async Flask-like)

Technical fit for yfinance:

- Drop-in Flask replacement with async/await support
- yfinance calls become non-blocking if wrapped properly

Pros:

- Familiar Flask API surface; easier migration than FastAPI
- True async throughout the stack

Cons:

- Less mature than FastAPI; smaller community
- Marginal benefit over FastAPI (you might as well learn FastAPI's newer patterns)

Verdict: Skip this. Either go Flask (simple, sync) or FastAPI (modern, async). Quart is the "middle ground" nobody needs.

Option D: Django + Celery

Technical fit for yfinance:

- Django is MVC-heavy; Celery is distributed task queue
- Overkill for polling architecture

Pros:

- Excellent ORM (Django ORM), admin interface, migrations out-of-box
- Industry-standard for larger Python projects

Cons:

- **Way too heavy for POC.** Django minimum setup requires understanding models, settings, middleware. Adds 30+ min setup time.
- Celery + Redis adds distributed complexity you don't need yet (polling is simple in-process job)
- Memory footprint: 150+ MB RAM

Verdict: No. Save Django for when your POC becomes a real product with multiple services.

Comparison Table: Python Backends

Aspect	FastAPI	Flask + APScheduler	Quart	Django + Celery
Setup Time	10 min	5 min	10 min	45 min
yfinance Integration	Excellent (can async-wrap)	Natural fit	Good	Overkill
Scheduler Simplicity	DIY or APScheduler	Native APScheduler	DIY	Celery (complex)
Memory	~80 MB	~50 MB	~80 MB	150+ MB
Type Safety	✓ Built-in	✗ Manual	✓ Possible	⚠ Partial
Concurrency Model	Async-first	Sync (async possible)	Async-first	Sync (Celery async)
Deployment Complexity	Moderate (Uvicorn)	Low (built-in server)	Moderate	High (Celery + broker)
POC Fit	★★★★★	★★★★☆	★★★☆☆	★☆☆☆☆

Frontend Options (Keeping Vue.js vs Full-Stack Python)

Here's where it gets interesting. You said "keep Vue.js as separated frontend project?" Let me address that:

Option 1: Separate Vue.js Frontend + Python API (Recommended)

Architecture:

```

frontend/ (Vue 3 + Vite)
├── src/
│   ├── components/
│   │   └── StockChart.vue (using chart.js)
│   ├── pages/
│   │   ├── Watchlist.vue
│   │   └── Alerts.vue
│   └── api/ (fetch calls to Python backend)

backend/ (FastAPI)
├── routes/
│   ├── watchlist.py
│   ├── alerts.py
│   └── prices.py
├── scheduler/
│   └── price_monitor.py
└── services/
    └── yfinance_service.py

```

Pros:

- **Clean separation of concerns:** Frontend handles UI, backend handles yfinance + scheduling. Each can be deployed independently.
- **Vue.js strengths shine:** Reactive data binding for real-time chart updates, easy component reuse.
- **Chart.js works perfectly:** Standard workflow (fetch data from API → render in Vue component).
- **Skill reuse:** You already planned Vue + chart.js; no learning curve spike.
- **Scalability:** If POC succeeds, swapping frontend framework (to React, Svelte) doesn't touch backend.
- **Development experience:** Vue dev server hot-reloads independently; Python backend hot-reloads separately.

Cons:

- **CORS setup:** Frontend (localhost:5173) talking to backend (localhost:8000) requires CORS headers. Minor friction during dev.
- **Two deployment pipelines:** Frontend static hosting (Vercel, Netlify, Render) + backend on Railway. Slightly more ops overhead than single monolith.
- **Coordination needed:** API contract changes require frontend + backend work.

Verdict: ✓ **Best choice.** Maintains your original plan, leverages Vue expertise, separates concerns cleanly.

Option 2: Streamlit (Full-Stack Python, No Frontend Code)

Architecture:

```
app.py (single file, ~300 lines)
├── Streamlit UI components
├── yfinance calls
├── SQLite queries
├── Chart rendering (built-in or plotly)
└── Scheduler in background
```

Pros:

- **Zero frontend code:** Write everything in Python. No HTML/CSS/JS.
- **Rapid prototyping:** Streamlit's st. API is incredibly fast (st.dataframe(), st.chart(), st.number_input()).
- **Built-in chart library:** Streamlit has st.line_chart() or integrates plotly seamlessly.
- **Single deploy:** One container, one command.
- **Dashboard-like UX:** Familiar to data scientists; looks professional.

Cons:

- **Limited customization:** Streamlit UI is opinionated; hard to build custom designs (buttons, modals, complex layouts).
- **Not real-time:** Each interaction reruns entire script from top (page reload). Awkward for continuous price monitoring.
- **Scheduler complexity:** Running background jobs (polling yfinance every 60s) while serving web UI is possible but hacky (requires threading + state management).
- **Learning curve:** Different mental model than traditional web framework; not transferable to other Python frameworks.
- **Scaling limitations:** Streamlit is great for internal dashboards, not production user-facing apps. Session management is minimal.

Verdict: \triangle **Fast for prototype, but wrong tool for an alert system.** Streamlit's reruns-on-interaction model doesn't mesh with continuous price polling. Better for "dashboard to view alerts" than "background poller + UI."

Option 3: ReactPy (Python React-like, Experimental)

What it is: Write React-like components in Python; transpiles to JavaScript.

Pros:

- Theoretically bridges Python backend + JS frontend in one language.

Cons:

- **Immature ecosystem:** Still experimental; few production apps use it.
- **Not actually faster:** You're writing Python that becomes JavaScript anyway; you still need JavaScript knowledge.
- **Tooling overhead:** Build step (Webpack/Vite), debugging is convoluted.
- **No real benefit:** You're not saving time vs. separate Vue frontend.

Verdict: \times **Skip.** Solves a problem you don't have (Python frontend expertise). Not mature enough for POC risk.

Option 4: FastAPI + Jinja2 Templates (Server-Rendered HTML)

Architecture:

```
backend/ (FastAPI)
  ├── routes/
  |   ├── pages.py (render HTML)
  |   └── api.py (JSON endpoints)
  └── templates/
      ├── watchlist.html
      └── alerts.html (with embedded chart.js)
```

```

└── services/
    └── yfinance_service.py

```

Pros:

- **Monolithic simplicity:** Single Python codebase handles UI + business logic.
- **No CORS issues:** Server renders HTML directly.
- **Traditional MVC pattern:** Familiar if you've built web apps before.

Cons:

- **Limited interactivity:** Page reloads on every action (add watchlist item, update threshold). Feels sluggish vs. Vue's reactivity.
- **Chart updates are hard:** Embedding chart.js in Jinja templates works but is clunky (need to serialize data server-side, inject into script tags).
- **Harder to test UI:** No component model; HTML templates are harder to unit test.
- **Frontloader → Vue later:** If you need to switch to SPA later, you're rewriting the UI layer entirely.

Verdict: \triangle Simpler to start, but wrong direction for your goals. You wanted Vue for interactivity; server-rendered HTML throws that away.

Frontend Options Comparison

Aspect	Vue.js (Separate)	Streamlit	ReactPy	FastAPI + Jinja
Customization	\checkmark Full control	\times Limited	\triangle Possible	\triangle HTML-only
Interactivity	\checkmark Real-time	\triangle Page-reload	\checkmark Real-time	\times Page-reload
Chart.js Integration	\checkmark Natural	\triangle Streamlit charts	\triangle Via npm?	\triangle Embedded
Background Polling	\checkmark Handled by backend	\times Clunky	\triangle Possible	\triangle Hard
Setup Time	15 min	5 min	20 min	10 min
Learning Curve	\checkmark Reuse existing	\times New mental model	\times Experimental	\triangle Familiar
Skill Transfer	\checkmark Portable (Vue ecosystem)	\times Streamlit-specific	\times Experimental	\triangle Old-school
POC Fit	★★★★★	★★★☆☆	★★☆☆☆	★★★☆☆

Final Recommendation (Python Stack)

Backend: FastAPI + yfinance

- **Reasoning:** Async-native fits polling patterns, type hints catch bugs early, minimal setup. yfinance integrates naturally.
- **Scheduler:** APScheduler inside FastAPI (same as node-cron approach). Simple, in-process, POC-appropriate.
- **Database:** Keep SQLite (no changes). Use SQLAlchemy ORM for clean migrations when/if you scale to PostgreSQL.

Frontend: Vue.js (separate project) + chart.js

- **Reasoning:** Preserves your original plan, leverages your existing Vue knowledge, maintains clean separation. Chart.js works out-of-the-box.
- **Deployment:** Vue on Netlify/Vercel (free static hosting). FastAPI backend on Railway.

Scheduler: APScheduler in FastAPI process

- **Polling:** Every 60-120s, APScheduler runs `check_watchlist()` → yfinance fetch → SQLite query → Telegram alert.
- **Tradeoff:** Single point of failure (same as node-cron). Acceptable for POC. Upgrade to Celery + Redis if scaled.

Why NOT the other options:

- Streamlit: Reruns-on-interaction model breaks continuous price monitoring.
- ReactPy: Immature, solves no problem you have.
- FastAPI + Ninja: Sacrifices the Vue interactivity you wanted.
- Django: Overkill, 5x setup time.

Tech Stack Summary (Revised)

```
Backend: FastAPI + yfinance + SQLAlchemy
Scheduler: APScheduler (inside FastAPI)
Frontend: Vue.js 3 + Vite + chart.js
Database: SQLite (→ PostgreSQL on scale)
Alerts: Telegram (unchanged)
Hosting: FastAPI on Railway | Vue on Netlify/Vercel
Cost: ~$0-15/month
Setup: ~6-8 hours total
```

This keeps the spirit of your original Node.js plan (simple, fast, cost-effective) but gains yfinance's free, unrestricted access to global ETF/stock data—the constraint that drove this pivot.

Trade-off clarity: You lose Node.js's tiny memory footprint (~5 MB savings), but gain Python ecosystem maturity for financial data (pandas, numpy). On Railway's pay-per-compute,

negligible cost difference (\$0.50/month max).

Implementation Roadmap (Week 1-2)

Days 1-2: FastAPI skeleton + yfinance basic fetch

Days 3-4: SQLite schema (watchlist, alerts) + SQLAlchemy ORM

Days 5-6: APScheduler polling loop + Telegram integration

Days 7-8: Vue frontend + API integration + chart.js rendering

Days 9-10: Deploy to Railway + Netlify, manual testing

Days 11-14: Edge case handling, documentation

Does this Python-based direction align with your POC goals? Any specific concerns on FastAPI vs Flask, or the frontend separation?

**

what do you mean by ASGI awareness?

ASGI awareness basically means: being conscious that your Python web app is running on an **asynchronous server interface** (ASGI) instead of the older synchronous one (WSGI), and understanding the implications:

1. ASGI vs WSGI in one sentence

- **WSGI:** Old standard, **sync-only**, one request = one blocking function (`def app(environ, start_response)`).
- **ASGI:** New standard, **async-capable**, supports `async def` handlers, websockets, long-lived connections.

FastAPI, Starlette, etc. are **ASGI frameworks**.

2. What “awareness” means in practice

When building with FastAPI (ASGI):

1. Async vs sync functions

- You can (and usually should) declare routes as:

```
@app.get("/prices")
async def get_prices():
    ...
```

- Inside `async def`, **you must not block the event loop** with heavy CPU work or long blocking I/O.

2. Blocking libraries (like yfinance)

- `yfinance` is synchronous. If you call it directly in `async` handlers:

```
async def get_prices():
    data = yf.Ticker("VOO").history(period="1d")  # blocking
```

This blocks the event loop and defeats the point of async.

- ASGI awareness means: wrap blocking calls in a threadpool:

```
import asyncio
import yfinance as yf

async def get_price():
    loop = asyncio.get_running_loop()
    data = await loop.run_in_executor(
        None, lambda: yf.Ticker("VOO").history(period="1d")
    )
    return data
```

3. Server choice and config

- You serve an ASGI app with something like:

```
uvicorn main:app --workers 1 --host 0.0.0.0 --port 8000
```

- ASGI awareness here: understanding that:

- Uvicorn is an **ASGI server**, not WSGI (like gunicorn+WSGI).
- If you increase workers, you're spawning multiple processes, each with its own event loop and scheduler.

4. Middleware / lifespan

- Middlewares and startup/shutdown hooks follow ASGI patterns:

```
@app.on_event("startup")
async def startup():
    # open DB, init scheduler, etc.
```

- ASGI awareness: use these hooks to start background tasks (APScheduler, etc.) correctly so they tie into the app lifecycle.

3. Why it matters for your project

Because you're leaning towards **FastAPI + yfinance + APScheduler**, ASGI awareness means:

- Using `async def` for your HTTP endpoints to keep them non-blocking.
- Ensuring **heavy / blocking work (yfinance calls, maybe DB)** are either:
 - Run in a thread executor, or
 - Done in background jobs (APScheduler) not on the request path.
- Running the app on an ASGI server like Uvicorn (or Uvicorn+Gunicorn combo) with correct settings for Railway.

In short:

ASGI awareness = "know that the framework is async, and don't accidentally turn it into a blocking, single-threaded bottleneck by calling synchronous code the wrong way."

**

is Node.js a runtime server ASGI or WSGI?

Neither. ASGI and WSGI are **Python-only** interface standards; Node.js lives in a completely different world.

- **WSGI** = Web Server Gateway Interface (Python, sync).
- **ASGI** = Asynchronous Server Gateway Interface (Python, async-capable).

Node.js:

- Is a **JavaScript runtime** with its own **event loop** and built-in http server.
- Frameworks like **Express, Fastify, Koa** run directly on top of Node's HTTP server.
- There is no WSGI/ASGI equivalent spec; the "interface" is basically:

```
const http = require('http');

const server = http.createServer((req, res) => {
    // handle request
});

server.listen(3000);
```

So in Python you choose **WSGI vs ASGI** and a server (Gunicorn/Uvicorn/etc.).

In Node you just run node app.js, and the runtime + framework together act as both app and server—no WSGI/ASGI layer involved.

**

I hesitate between FastAPI vs Flask. I don't have experience at all in any of those so are unknown to me.

My concerns are around how much benefit is having FastAPI handling async if it's just a PoC with one user (me) so far. In hypothetical case if PoC became an MVP, then I'd force change from yfinance to a more robust solution like a free tier/paid API like Finnhub or [StockData.org](#)

Excellent clarification question. Let me cut through the noise with a **practical decision framework** for your specific constraints.

The Real Question: Does Async Matter for Your POC?

Short answer: No. Not yet.

Here's why:

1. POC = Single User (You)

- One person triggering requests.
- One price-polling loop running in background (every 60-120s).
- No concurrent load; no "thundering herd" of users.
- **Async buys you nothing here.** Sync code runs just fine.

2. Your Actual Bottleneck: yfinance + Network I/O

- yfinance fetches data from Yahoo Finance API (~100-500ms per call).
- This is **I/O-bound, not CPU-bound.**
- Flask (sync) handles this fine:

```
# Flask, synchronous - perfectly acceptable for POC
@app.get("/prices/<symbol>")
def get_price(symbol):
    data = yf.Ticker(symbol).history(period="1d")  # blocks, but who cares? It's me 😊
    return {"price": data['Close'].iloc[-1]}
```

- Async would make it slightly faster if you were juggling 100 requests at once. You're not.

3. Your Scheduler is the Actual Workload

- APScheduler runs a polling loop every 60-120s: fetch 5-10 ETFs/stocks, compare thresholds, send Telegram.
- This happens **in the background**, independent of HTTP requests.
- APScheduler + Flask work together cleanly; no special async needed:

```
# Flask app + APScheduler
scheduler = APScheduler()
scheduler.init_app(app)
scheduler.start()

@scheduler.task('interval', id='price_check', seconds=60)
def check_prices():
    # runs every 60s, blocks during yfinance calls, nobody notices
    pass
```

Decision Tree: Flask vs FastAPI for Your POC

Question: "Do I need async?"

- Do you have 100s of concurrent users?
 - └ No → Move to FastAPI only if you want to learn it
- Do you have complex I/O coordination (fetch 3 APIs in parallel, merge results)?
 - └ No → Sync (Flask) is simpler
- Is your bottleneck yfinance network calls?
 - └ Yes → Async helps marginally; not critical for POC
- Will your POC change APIs soon (yfinance → Finnhub)?
 - └ Yes → API layer is abstracted either way; language choice doesn't matter
- Result: FLASK IS FINE

Head-to-Head: Flask vs FastAPI for Your Actual POC

Criterion	Flask	FastAPI	Winner
Setup time	5 min	10 min	Flask ✓
Learning curve	Ultra-gentle	Moderate (decorators, type hints, docs)	Flask ✓
Single-user POC speed	~100ms per request	~90ms per request	Tie (irrelevant difference)
Code clarity	Simple, obvious	Modern, type-hinted	FastAPI (minor)
Future scaling to MVP	Need rewrite if concurrent load	Can add async later	FastAPI (tiny advantage)
yfinance integration	Direct call (easy)	Need <code>run_in_executor</code> wrapper	Flask ✓
APScheduler integration	Native, seamless	Also native, seamless	Tie
Deployment on Railway	Works great	Also works great	Tie
"Feel" for a 1-person project	Lightweight, focused	Maybe overengineered	Flask ✓

My Honest Take

For your POC? Use Flask.

Why:

1. **No learning tax.** Flask is the most beginner-friendly Python web framework. Literally:

```

from flask import Flask
app = Flask(__name__)

@app.route("/api/prices/<symbol>")
def get_price(symbol):
    price = yf.Ticker(symbol).history(period="1d")['Close'].iloc[-1]
    return {"price": price}

if __name__ == "__main__":
    app.run(debug=True)

```

That's a working API in 10 lines. You can reason about it immediately.

2. **Zero async complexity.** No threading, no event loops, no `run_in_executor`. Just write normal Python.
3. **APScheduler fits naturally.** One Flask app, one scheduler, one SQLite file. Done.
4. **yfinance works as-is.** No wrapper functions needed. Call it directly.
5. **"Fast enough."** 100ms response time for a personal POC? Who cares. It's you using it.

What Changes When You Scale POC → MVP?

Switching from Flask to FastAPI later is trivial (if needed at all):

Change	Scale Trigger	Effort
Single sync Flask → async FastAPI	50+ concurrent users	2-3 hours refactor (add <code>async def</code> , move blocking calls to executors)
yfinance → Finnhub API	Hit yfinance rate limits	Swap 3-4 lines in <code>services/stock_service.py</code>
SQLite → PostgreSQL	10,000+ watchlist entries	1 hour (connection string + ORM migrations, if using SQLAlchemy)
Single Flask process → load balancer	Horizontal scaling	Add Gunicorn workers, slap NGINX in front

Key insight: None of these require you to pick FastAPI now. Flask + clean architecture (services layer, ORM) scales fine up to "small MVP." You only rewrite to FastAPI when profiling shows I/O concurrency is actually your bottleneck (spoiler: it won't be).

Recommendation: Flask POC Path

Tech stack (final):

```

Backend:  Flask + APScheduler + yfinance
Frontend: Vue.js 3 + chart.js (no changes)
Database: SQLite + SQLAlchemy ORM (for migrations)
Scheduler: APScheduler (built-in)
Alerts:    Telegram (unchanged)

```

Hosting:	Railway
Cost:	\$0-15/month
Setup:	~4-5 hours (simpler than FastAPI)

Minimal Flask skeleton:

```
backend/
├── app.py          (Flask init, routes)
├── config.py       (config from env)
└── models/
    └── watchlist.py (SQLAlchemy models)
├── services/
    ├── yfinance_service.py
    ├── alert_service.py
    └── telegram_service.py
└── scheduler/
    └── price_monitor.py (APScheduler tasks)
└── requirements.txt
└── .env
```

Trade-off clarity:

- ✓ Faster to write, easier to debug, one less thing to learn simultaneously.
- △ If you *do* scale to many concurrent users (unlikely for personal tool), you migrate to FastAPI—but that's a "good problem to have" decision.

When You'd Switch to FastAPI

Only if:

1. POC becomes MVP with 10+ users, *and*
2. Profiling shows Flask handling 50+ concurrent requests per second, *and*
3. Response times degrade noticeably.

For a personal stock alerting tool? Probably never. Flask runs forever on a cheap server.

Does this settle the hesitation? Flask for POC, zero regrets, clean enough to refactor if needed. You'll learn the Python web fundamentals faster, and can focus on yfinance integration instead of async/await semantics.

**

Alright , with the final decision of tech stack:

Backend: Flask + APScheduler + yfinance

Frontend: Vue.js 3 + chart.js (no changes)

Database: SQLite + SQLAlchemy ORM (for migrations)

Scheduler: APScheduler (built-in)

Alerts: Telegram (unchanged)

Hosting: Railway

Cost: \$0-15/month

Setup: ~4-5 hours (simpler than FastAPI)

Help me scaffolding 2 projects, one for the python backend , the other for the Vue.js frontend.

For Backend project:

- Initial setup files for Flask,
- Basic REST API crud endpoints to cover alerts, watchlist and price fetching using yfinance
- Sqlite database setup with alerts and watchlist schemas
- Starter scheduler logic to pool fetching of prices every 5 minutes

For Frontend project

- Basic setup of Vue.js project and chart.js
- Basic Vue template that holds Single Page Application with similar layout in the StockAlarm layout image I attached.

Perfect. I'll scaffold both projects based on the StockAlarm layout you provided. Let me create complete, working starter code for both backend and frontend.

Backend Project Scaffold

I'll create a complete Flask backend with SQLite, yfinance integration, APScheduler, and REST endpoints.

Backend Directory Structure

```
stock-price-alert-backend/
├── app.py                                # Flask app entry point
├── config.py                             # Configuration (dev/prod)
├── requirements.txt                      # Python dependencies
├── .env.example                          # Environment variables template
└── .gitignore

├── models/
│   ├── __init__.py                         # Watchlist ORM models
│   ├── watchlist.py                        # Alert ORM models
│   └── alert.py

├── services/
│   ├── __init__.py                         # yfinance wrapper
│   ├── yfinance_service.py                 # Alert logic (price comparisons)
│   ├── alert_service.py                   # Telegram notifications
│   └── telegram_service.py

└── routes/
    ├── __init__.py                         # GET/POST/PUT/DELETE /api/watchlist
    └── watchlist_routes.py
```

```

    └── alert_routes.py          # GET/POST /api/alerts
        └── price_routes.py      # GET /api/prices/:symbol

    └── scheduler/
        ├── __init__.py
        └── price_monitor.py     # APScheduler job logic

    └── db/
        ├── __init__.py
        └── database.py          # SQLite setup & session management

    └── logs/                  # Scheduler/app logs (created at runtime)

```

Backend: Core Files

1. requirements.txt

```

Flask==3.0.0
Flask-SQLAlchemy==3.1.1
SQLAlchemy==2.0.23
yfinance==0.2.32
APScheduler==3.10.4
python-dotenv==1.0.0
requests==2.31.0

```

2. .env.example

```

# Flask
FLASK_ENV=development
FLASK_APP=app.py
SECRET_KEY=your-secret-key-here-change-in-production

# Database
DATABASE_URL=sqlite:///stock_alert.db

# Telegram
TELEGRAM_BOT_TOKEN=your-telegram-bot-token-here
TELEGRAM_CHAT_ID=your-telegram-chat-id-here

# Scheduler
SCHEDULER_INTERVAL_SECONDS=300

# CORS (for Vue frontend)
FRONTEND_URL=http://localhost:5173

```

3. config.py

```
import os
from dotenv import load_dotenv

load_dotenv()

class Config:
    """Base configuration"""
    SECRET_KEY = os.getenv('SECRET_KEY', 'dev-secret-key-change-me')
    SQLALCHEMY_DATABASE_URI = os.getenv('DATABASE_URL', 'sqlite:///stock_alert.db')
    SQLALCHEMY_TRACK_MODIFICATIONS = False

    # Telegram
    TELEGRAM_BOT_TOKEN = os.getenv('TELEGRAM_BOT_TOKEN', '')
    TELEGRAM_CHAT_ID = os.getenv('TELEGRAM_CHAT_ID', '')

    # Scheduler
    SCHEDULER_INTERVAL_SECONDS = int(os.getenv('SCHEDULER_INTERVAL_SECONDS', 300))

    # CORS
    FRONTEND_URL = os.getenv('FRONTEND_URL', 'http://localhost:5173')

class DevelopmentConfig(Config):
    """Development configuration"""
    DEBUG = True

class ProductionConfig(Config):
    """Production configuration"""
    DEBUG = False

config = {
    'development': DevelopmentConfig,
    'production': ProductionConfig,
    'default': DevelopmentConfig
}
```

4. db/database.py

```
from flask_sqlalchemy import SQLAlchemy
from contextlib import contextmanager

db = SQLAlchemy()

def init_db(app):
    """Initialize database with Flask app"""
    db.init_app(app)
    with app.app_context():
        db.create_all()
        print("✓ Database initialized")

@contextmanager
def get_db_session():
    """Context manager for DB session"""
```

```

session = db.session
try:
    yield session
except Exception as e:
    session.rollback()
    print(f"X Database error: {e}")
    raise
finally:
    session.close()

```

5. models/watchlist.py

```

from db.database import db
from datetime import datetime

class Watchlist(db.Model):
    """Store user's watched stocks/ETFs"""
    __tablename__ = 'watchlist'

    id = db.Column(db.Integer, primary_key=True)
    symbol = db.Column(db.String(10), unique=True, nullable=False, index=True)
    name = db.Column(db.String(100)) # e.g., "Vanguard VOO ETF"
    price_at_add = db.Column(db.Float) # Price when added
    created_at = db.Column(db.DateTime, default=datetime.utcnow)
    updated_at = db.Column(db.DateTime, default=datetime.utcnow, onupdate=datetime.utcnow)

    # Relationship to alerts
    alerts = db.relationship('Alert', backref='watchlist', lazy=True, cascade='all, delete')

    def to_dict(self):
        return {
            'id': self.id,
            'symbol': self.symbol,
            'name': self.name,
            'price_at_add': self.price_at_add,
            'created_at': self.created_at.isoformat(),
            'updated_at': self.updated_at.isoformat(),
            'alerts_count': len(self.alerts)
        }

    def __repr__(self):
        return f'<Watchlist {self.symbol}>'

```

6. models/alert.py

```

from db.database import db
from datetime import datetime

class Alert(db.Model):
    """Store price alerts for watchlist items"""
    __tablename__ = 'alert'

    id = db.Column(db.Integer, primary_key=True)

```

```

watchlist_id = db.Column(db.Integer, db.ForeignKey('watchlist.id'), nullable=False, index=True)

# Alert type: 'price_above' or 'price_below'
alert_type = db.Column(db.String(20), nullable=False)

# Threshold price
threshold_price = db.Column(db.Float, nullable=False)

# Alert state
is_active = db.Column(db.Boolean, default=True, index=True)
triggered = db.Column(db.Boolean, default=False)
triggered_at = db.Column(db.DateTime)

created_at = db.Column(db.DateTime, default=datetime.utcnow)
updated_at = db.Column(db.DateTime, default=datetime.utcnow, onupdate=datetime.utcnow)

def to_dict(self):
    return {
        'id': self.id,
        'watchlist_id': self.watchlist_id,
        'symbol': self.watchlist.symbol,
        'alert_type': self.alert_type,
        'threshold_price': self.threshold_price,
        'is_active': self.is_active,
        'triggered': self.triggered,
        'triggered_at': self.triggered_at.isoformat() if self.triggered_at else None,
        'created_at': self.created_at.isoformat(),
        'updated_at': self.updated_at.isoformat()
    }

def __repr__(self):
    return f'<Alert {self.watchlist.symbol} {self.alert_type} {self.threshold_price}>'

```

7. services/yfinance_service.py

```

import yfinance as yf
from datetime import datetime, timedelta
import logging

logger = logging.getLogger(__name__)

class YFinanceService:
    """Wrapper around yfinance library"""

    @staticmethod
    def get_current_price(symbol):
        """
        Fetch current price for a symbol (stock or ETF)
        Returns: float or None
        """
        try:
            ticker = yf.Ticker(symbol)
            # Get last day's close price
            hist = ticker.history(period="1d")
        except Exception as e:
            logger.error(f"Error fetching current price for {symbol}: {e}")
            return None
        return hist['Close'].iloc[-1]

```

```

        if hist.empty:
            logger.warning(f"No data found for {symbol}")
            return None

        price = hist['Close'].iloc[-1]
        return float(price)

    except Exception as e:
        logger.error(f"Error fetching price for {symbol}: {e}")
        return None

    @staticmethod
    def get_stock_info(symbol):
        """
        Fetch basic stock/ETF info (name, sector, etc.)
        Returns: dict with metadata or None
        """
        try:
            ticker = yf.Ticker(symbol)
            info = ticker.info

            return {
                'symbol': symbol,
                'name': info.get('longName', symbol),
                'currency': info.get('currency', 'USD'),
                'exchange': info.get('exchange', 'UNKNOWN'),
                'market_cap': info.get('marketCap'),
                'sector': info.get('sector')
            }
        except Exception as e:
            logger.error(f"Error fetching info for {symbol}: {e}")
            return None

    @staticmethod
    def get_historical_prices(symbol, period="1mo"):
        """
        Fetch historical prices for charting
        period: "1mo", "3mo", "6mo", "1y"
        Returns: list of [date, price] pairs
        """
        try:
            ticker = yf.Ticker(symbol)
            hist = ticker.history(period=period)

            if hist.empty:
                return []

            # Return date (ISO format) and closing price
            data = []
            for idx, row in hist.iterrows():
                data.append({
                    'date': idx.strftime('%Y-%m-%d'),
                    'close': float(row['Close']),
                    'high': float(row['High']),
                    'low': float(row['Low']),
                    'volume': int(row['Volume'])
                })
        
```

```

        })

    return data

except Exception as e:
    logger.error(f"Error fetching historical prices for {symbol}: {e}")
    return []

# Singleton instance
yfinance_service = YFinanceService()

```

8. services/alert_service.py

```

from models.alert import Alert
from models.watchlist import Watchlist
from db.database import db
from datetime import datetime
import logging

logger = logging.getLogger(__name__)

class AlertService:
    """Business logic for price alerts"""

    @staticmethod
    def check_price_alert(watchlist_item, current_price):
        """
        Check if any active alerts for this watchlist item should trigger
        Returns: list of triggered alerts
        """
        triggered = []

        for alert in watchlist_item.alerts:
            if not alert.is_active or alert.triggered:
                continue

            should_trigger = False

            if alert.alert_type == 'price_above':
                should_trigger = current_price >= alert.threshold_price
            elif alert.alert_type == 'price_below':
                should_trigger = current_price <= alert.threshold_price

            if should_trigger:
                alert.triggered = True
                alert.triggered_at = datetime.utcnow()
                db.session.add(alert)
                triggered.append(alert)
                logger.info(f"✓ Alert triggered: {watchlist_item.symbol} {alert.alert_ty}

        if triggered:
            db.session.commit()

    return triggered

```

```

@staticmethod
def get_active_alerts(watchlist_id=None):
    """Get all active (untriggered) alerts, optionally filtered by watchlist"""
    query = Alert.query.filter(Alert.is_active == True, Alert.triggered == False)

    if watchlist_id:
        query = query.filter(Alert.watchlist_id == watchlist_id)

    return query.all()

@staticmethod
def reset_alert(alert_id):
    """Reset a triggered alert back to active state"""
    alert = Alert.query.get(alert_id)
    if alert:
        alert.triggered = False
        alert.triggered_at = None
        db.session.commit()
        logger.info(f"✓ Alert reset: {alert.id}")
        return alert
    return None

alert_service = AlertService()

```

9. services/telegram_service.py

```

import requests
from flask import current_app
import logging

logger = logging.getLogger(__name__)

class TelegramService:
    """Send alerts via Telegram Bot"""

    @staticmethod
    def send_alert(watchlist_symbol, alert_type, threshold, current_price):
        """
        Send alert message to Telegram
        alert_type: "price_above" or "price_below"
        """

        token = current_app.config.get('TELEGRAM_BOT_TOKEN')
        chat_id = current_app.config.get('TELEGRAM_CHAT_ID')

        if not token or not chat_id:
            logger.warning("Telegram credentials not configured")
            return False

        # Format message
        if alert_type == 'price_above':
            message = f"\u25b6 {watchlist_symbol} reached ${current_price:.2f} (above ${threshold:.2f})"
        else:
            message = f"\u25b6 {watchlist_symbol} reached ${current_price:.2f} (below ${threshold:.2f})"

        try:
            response = requests.post(
                f"https://api.telegram.org/bot{token}/sendMessage",
                json={"chat_id": chat_id, "text": message}
            )
            if response.status_code != 200:
                logger.error(f"Error sending alert: {response.text}")
        except requests.exceptions.RequestException as e:
            logger.error(f"Error sending alert: {e}")

```

```

url = f"https://api.telegram.org/bot{token}/sendMessage"
payload = {
    'chat_id': chat_id,
    'text': message,
    'parse_mode': 'HTML'
}

response = requests.post(url, json=payload, timeout=10)

if response.status_code == 200:
    logger.info(f"✓ Telegram message sent: {message}")
    return True
else:
    logger.error(f"✗ Telegram error: {response.text}")
    return False

except Exception as e:
    logger.error(f"✗ Error sending Telegram message: {e}")
    return False

telegram_service = TelegramService()

```

10. scheduler/price_monitor.py

```

from apscheduler.schedulers.background import BackgroundScheduler
from flask import current_app
from models.watchlist import Watchlist
from services.yfinance_service import yfinance_service
from services.alert_service import alert_service
from services.telegram_service import telegram_service
from db.database import db
import logging
from datetime import datetime

logger = logging.getLogger(__name__)

scheduler = BackgroundScheduler(daemon=True)

def check_all_prices():
    """
    Main scheduler job: fetch prices for all watchlist items
    Check alerts, send Telegram notifications
    Runs every N seconds (configured in .env)
    """
    try:
        with current_app.app_context():
            logger.info(f"[{datetime.now().isoformat()}] Starting price check...")

            watchlist_items = Watchlist.query.all()

            if not watchlist_items:
                logger.info("No watchlist items to check")
                return

            for item in watchlist_items:

```

```

# Fetch current price from yfinance
current_price = yfinance_service.get_current_price(item.symbol)

if current_price is None:
    logger.warning(f"Could not fetch price for {item.symbol}")
    continue

logger.info(f"{item.symbol}: ${current_price:.2f}")

# Check if any alerts should trigger
triggered_alerts = alert_service.check_price_alert(item, current_price)

# Send Telegram notification for each triggered alert
for alert in triggered_alerts:
    telegram_service.send_alert(
        item.symbol,
        alert.alert_type,
        alert.threshold_price,
        current_price
    )

logger.info("✓ Price check complete")

except Exception as e:
    logger.error(f"✗ Error in price monitor: {e}", exc_info=True)

def init_scheduler(app):
    """Initialize and start the scheduler"""
    interval = app.config.get('SCHEDULER_INTERVAL_SECONDS', 300)

    scheduler.add_job(
        func=check_all_prices,
        trigger="interval",
        seconds=interval,
        id='price_monitor',
        name='Check prices and send alerts',
        replace_existing=True
    )

    scheduler.start()
    logger.info(f"✓ Scheduler started (interval: {interval}s)")

```

11. routes/watchlist_routes.py

```

from flask import Blueprint, request, jsonify
from models.watchlist import Watchlist
from services.yfinance_service import yfinance_service
from db.database import db
import logging

logger = logging.getLogger(__name__)

watchlist_bp = Blueprint('watchlist', __name__, url_prefix='/api/watchlist')

@watchlist_bp.route('', methods=['GET'])

```

```

def get_watchlist():
    """Get all watchlist items"""
    items = Watchlist.query.all()
    return jsonify([item.to_dict() for item in items]), 200

@watchlist_bp.route('/<int:watchlist_id>', methods=['GET'])
def get_watchlist_item(watchlist_id):
    """Get single watchlist item with current price"""
    item = Watchlist.query.get(watchlist_id)

    if not item:
        return jsonify({'error': 'Watchlist item not found'}), 404

    # Fetch current price
    current_price = yfinance_service.get_current_price(item.symbol)

    result = item.to_dict()
    result['current_price'] = current_price

    return jsonify(result), 200

@watchlist_bp.route('', methods=['POST'])
def add_to_watchlist():
    """Add a new item to watchlist"""
    data = request.get_json()

    if not data or 'symbol' not in data:
        return jsonify({'error': 'symbol is required'}), 400

    symbol = data['symbol'].upper()

    # Check if already in watchlist
    existing = Watchlist.query.filter_by(symbol=symbol).first()
    if existing:
        return jsonify({'error': f'{symbol} already in watchlist'}), 409

    # Fetch current price and info
    current_price = yfinance_service.get_current_price(symbol)
    stock_info = yfinance_service.get_stock_info(symbol)

    if current_price is None:
        return jsonify({'error': f'Could not fetch data for {symbol}'}), 400

    # Create watchlist item
    item = Watchlist(
        symbol=symbol,
        name=stock_info.get('name') if stock_info else symbol,
        price_at_add=current_price
    )

    db.session.add(item)
    db.session.commit()

    logger.info(f"✓ Added {symbol} to watchlist")
    return jsonify(item.to_dict()), 201

```

```

@watchlist_bp.route('/<int:watchlist_id>', methods=['DELETE'])
def remove_from_watchlist(watchlist_id):
    """Remove item from watchlist"""
    item = Watchlist.query.get(watchlist_id)

    if not item:
        return jsonify({'error': 'Watchlist item not found'}), 404

    symbol = item.symbol
    db.session.delete(item)
    db.session.commit()

    logger.info(f"✓ Removed {symbol} from watchlist")
    return jsonify({'message': f'{symbol} removed from watchlist'}), 200

```

12. routes/alert_routes.py

```

from flask import Blueprint, request, jsonify
from models.alert import Alert
from models.watchlist import Watchlist
from services.alert_service import alert_service
from db.database import db
import logging

logger = logging.getLogger(__name__)

alert_bp = Blueprint('alert', __name__, url_prefix='/api/alerts')

@alert_bp.route('', methods=['GET'])
def get_alerts():
    """Get all alerts, optionally filtered by watchlist_id"""
    watchlist_id = request.args.get('watchlist_id', type=int)

    query = Alert.query
    if watchlist_id:
        query = query.filter_by(watchlist_id=watchlist_id)

    alerts = query.all()
    return jsonify([alert.to_dict() for alert in alerts]), 200

@alert_bp.route('', methods=['POST'])
def create_alert():
    """Create a new price alert"""
    data = request.get_json()

    # Validation
    required = ['watchlist_id', 'alert_type', 'threshold_price']
    if not all(key in data for key in required):
        return jsonify({'error': f'Required fields: {required}'}), 400

    if data['alert_type'] not in ['price_above', 'price_below']:
        return jsonify({'error': 'alert_type must be "price_above" or "price_below"'}), 400

    # Check watchlist item exists
    watchlist = Watchlist.query.get(data['watchlist_id'])

```

```

if not watchlist:
    return jsonify({'error': 'Watchlist item not found'}), 404

# Create alert
alert = Alert(
    watchlist_id=data['watchlist_id'],
    alert_type=data['alert_type'],
    threshold_price=float(data['threshold_price']),
    is_active=True
)

db.session.add(alert)
db.session.commit()

logger.info(f"✓ Created alert: {alert}")
return jsonify(alert.to_dict()), 201

@alert_bp.route('/<int:alert_id>', methods=['DELETE'])
def delete_alert(alert_id):
    """Delete an alert"""
    alert = Alert.query.get(alert_id)

    if not alert:
        return jsonify({'error': 'Alert not found'}), 404

    db.session.delete(alert)
    db.session.commit()

    logger.info(f"✓ Deleted alert {alert_id}")
    return jsonify({'message': 'Alert deleted'}), 200

@alert_bp.route('/<int:alert_id>/reset', methods=['POST'])
def reset_alert(alert_id):
    """Reset a triggered alert back to active"""
    alert = alert_service.reset_alert(alert_id)

    if not alert:
        return jsonify({'error': 'Alert not found'}), 404

    return jsonify(alert.to_dict()), 200

```

13. routes/price_routes.py

```

from flask import Blueprint, request, jsonify
from services.yfinance_service import yfinance_service
import logging

logger = logging.getLogger(__name__)

price_bp = Blueprint('price', __name__, url_prefix='/api/prices')

@price_bp.route('/<symbol>', methods=['GET'])
def get_price(symbol):
    """Get current price for a symbol"""
    symbol = symbol.upper()

```

```

price = yfinance_service.get_current_price(symbol)

if price is None:
    return jsonify({'error': f'Could not fetch price for {symbol}'}), 400

return jsonify({'symbol': symbol, 'price': price}), 200

@price_bp.route('/<symbol>/historical', methods=['GET'])
def get_historical_prices(symbol):
    """Get historical prices for charting"""
    symbol = symbol.upper()
    period = request.args.get('period', '1mo')  # 1mo, 3mo, 6mo, 1y

    prices = yfinance_service.get_historical_prices(symbol, period=period)

    if not prices:
        return jsonify({'error': f'Could not fetch historical data for {symbol}'}), 400

    return jsonify({'symbol': symbol, 'period': period, 'data': prices}), 200

@price_bp.route('/<symbol>/info', methods=['GET'])
def get_stock_info(symbol):
    """Get stock/ETF info"""
    symbol = symbol.upper()
    info = yfinance_service.get_stock_info(symbol)

    if not info:
        return jsonify({'error': f'Could not fetch info for {symbol}'}), 400

    return jsonify(info), 200

```

14. app.py (Flask Entry Point)

```

from flask import Flask, jsonify
from flask_cors import CORS
import logging
import os
from config import config
from db.database import init_db, db
from scheduler.price_monitor import init_scheduler

# Logging setup
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s [%(levelname)s] %(name)s: %(message)s'
)
logger = logging.getLogger(__name__)

def create_app(config_name='development'):
    """Flask app factory"""

    app = Flask(__name__)

    # Load configuration
    app.config.from_object(config[config_name])

```

```

# Initialize extensions
db.init_app(app)

# CORS setup for Vue frontend
CORS(app, origins=[app.config['FRONTEND_URL']])

with app.app_context():
    # Initialize database
    init_db(app)

    # Register blueprints
    from routes.watchlist_routes import watchlist_bp
    from routes.alert_routes import alert_bp
    from routes.price_routes import price_bp

    app.register_blueprint(watchlist_bp)
    app.register_blueprint(alert_bp)
    app.register_blueprint(price_bp)

    # Initialize scheduler
    init_scheduler(app)

    # Health check endpoint
    @app.route('/health', methods=['GET'])
    def health():
        return jsonify({'status': 'ok'}), 200

logger.info("✓ Flask app initialized")
return app

if __name__ == '__main__':
    app = create_app(os.getenv('FLASK_ENV', 'development'))
    app.run(host='0.0.0.0', port=5000, debug=True)

```

15. .gitignore

```

# Environment
.env
.env.local

# Database
*.db
*.sqlite
*.sqlite3

# Python
__pycache__/
*.py[cod]
*$py.class
*.so
.Python
env/
venv/
ENV/

```

```
build/
develop-eggs/
dist/
downloads/
eggs/
.eggs/
lib/
lib64/
parts/
sdist/
var/
wheels/
*.egg-info/
.installed.cfg
*.egg

# IDE
.vscode/
.idea/
*.swp
*.SWO

# Logs
logs/
*.log

# OS
.DS_Store
Thumbs.db
```

FRONTEND PROJECT SCAFFOLD

Now let's create the Vue.js frontend with chart.js integration.

Frontend Directory Structure

```
stock-price-alert-frontend/
├── src/
│   ├── main.js                  # Entry point
│   ├── App.vue                  # Root component
│   └── api.js                   # HTTP client (fetch to backend)

        ├── components/
        │   ├── WatchlistPanel.vue    # Left sidebar - watchlist + search
        │   ├── PriceChart.vue       # Center - chart.js chart
        │   ├── AlertsPanel.vue      # Right sidebar - alerts for selected stock
        │   └── StockInfoCard.vue    # Header - stock name, price, change %

        ├── assets/
        │   └── styles.css           # Global styles

        └── utils/
            └── formatters.js        # Number formatting, date utils
```

```
|  
|   index.html  
|   package.json  
|   vite.config.js  
|   .env.example  
|   .gitignore  
|  
└── README.md
```

Frontend: Core Files

1. package.json

```
{  
  "name": "stock-price-alert-frontend",  
  "version": "0.1.0",  
  "type": "module",  
  "scripts": {  
    "dev": "vite",  
    "build": "vite build",  
    "preview": "vite preview"  
  },  
  "dependencies": {  
    "vue": "^3.3.4",  
    "chart.js": "^4.4.0",  
    "axios": "^1.6.2"  
  },  
  "devDependencies": {  
    "@vitejs/plugin-vue": "^4.5.0",  
    "vite": "^5.0.0"  
  }  
}
```

2. .env.example

```
VITE_API_URL=http://localhost:5000
```

3. vite.config.js

```
import { defineConfig } from 'vite'  
import vue from '@vitejs/plugin-vue'  
  
export default defineConfig({  
  plugins: [vue()],  
  server: {  
    port: 5173,  
    host: true  
  }  
})
```

4. index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Stock Price Alert</title>
</head>
<body>
  <div id="app"></div>
  <script type="module" src="/src/main.js"></script>
</body>
</html>
```

5. src/main.js

```
import { createApp } from 'vue'
import App from './App.vue'
import './assets/styles.css'

createApp(App).mount('#app')
```

6. src/api.js

```
import axios from 'axios'

const API_URL = import.meta.env.VITE_API_URL || 'http://localhost:5000'

const apiClient = axios.create({
  baseURL: API_URL,
  headers: {
    'Content-Type': 'application/json'
  }
})

export const watchlistAPI = {
  getAll: () => apiClient.get('/api/watchlist'),
  getOne: (id) => apiClient.get(`/api/watchlist/${id}`),
  add: (symbol) => apiClient.post('/api/watchlist', { symbol }),
  remove: (id) => apiClient.delete(`/api/watchlist/${id}`)
}

export const alertAPI = {
  getAll: (watchlistId = null) => {
    const params = watchlistId ? { watchlist_id: watchlistId } : {}
    return apiClient.get('/api/alerts', { params })
  },
  create: (watchlistId, alertType, thresholdPrice) =>
    apiClient.post('/api/alerts', {
      watchlist_id: watchlistId,
      alert_type: alertType,
    })
}
```

```

        threshold_price: thresholdPrice
    },
    delete: (id) => apiClient.delete(`/api/alerts/${id}`),
    reset: (id) => apiClient.post(`/api/alerts/${id}/reset`)
}

export const priceAPI = {
    getCurrent: (symbol) => apiClient.get(`/api/prices/${symbol}`),
    getHistorical: (symbol, period = '1mo') =>
        apiClient.get(`/api/prices/${symbol}/historical`, { params: { period } }),
    getInfo: (symbol) => apiClient.get(`/api/prices/${symbol}/info`)
}

export default apiClient

```

7. src/utils/formatters.js

```

export const formatCurrency = (value) => {
    if (!value) return '$0.00'
    return new Intl.NumberFormat('en-US', {
        style: 'currency',
        currency: 'USD',
        minimumFractionDigits: 2
    }).format(value)
}

export const formatPercent = (value) => {
    if (!value) return '0.00%'
    return (value * 100).toFixed(2) + '%'
}

export const formatDate = (dateStr) => {
    if (!dateStr) return ''
    const date = new Date(dateStr)
    return date.toLocaleDateString('en-US', {
        month: 'short',
        day: 'numeric',
        hour: '2-digit',
        minute: '2-digit'
    })
}

export const formatNumber = (value) => {
    if (!value) return '0'
    return Number(value).toLocaleString('en-US', {
        minimumFractionDigits: 2,
        maximumFractionDigits: 2
    })
}

```

8. src/App.vue (Root Component)

```
<template>
<div class="app-container">
  <!-- Header -->
  <header class="header">
    <div class="header-content">
      <h1> Stock Price Alert</h1>
      <p class="subtitle">Track ETFs and stocks • Get price alerts on Telegram</p>
    </div>
  </header>

  <!-- Main Grid Layout -->
  <main class="main-content">
    <!-- Left: Watchlist Panel -->
    <aside class="left-panel">
      <WatchlistPanel
        :items="watchlist"
        :selected-id="selectedWatchlistId"
        @select="selectWatchlistItem"
        @add="addToWatchlist"
        @remove="removeFromWatchlist"
      />
    </aside>

    <!-- Center: Price Chart & Stock Info -->
    <section class="center-panel">
      <StockInfoCard
        v-if="selectedItem"
        :item="selectedItem"
        :current-price="currentPrice"
      />

      <PriceChart
        v-if="selectedItem"
        :symbol="selectedItem.symbol"
        :chart-data="chartData"
      />
    </section>

    <!-- Right: Alerts Panel -->
    <aside class="right-panel">
      <AlertsPanel
        v-if="selectedItem"
        :watchlist-item="selectedItem"
        :alerts="selectedAlerts"
        @add-alert="addAlert"
        @delete-alert="deleteAlert"
        @reset-alert="resetAlert"
      />
    </aside>
  </main>

  <!-- Loading/Error States -->
  <div v-if="loading" class="loading">Loading...</div>
  <div v-if="error" class="error">{{ error }}</div>
```

```
</div>
</template>

<script>
import { ref, computed, onMounted, onUnmounted } from 'vue'
import WatchlistPanel from './components/WatchlistPanel.vue'
import PriceChart from './components/PriceChart.vue'
import AlertsPanel from './components/AlertsPanel.vue'
import StockInfoCard from './components/StockInfoCard.vue'
import { watchlistAPI, alertAPI, priceAPI } from './api'

export default {
  name: 'App',
  components: {
    WatchlistPanel,
    PriceChart,
    AlertsPanel,
    StockInfoCard
  },
  setup() {
    const watchlist = ref([])
    const alerts = ref([])
    const selectedWatchlistId = ref(null)
    const currentPrice = ref(null)
    const chartData = ref([])
    const loading = ref(false)
    const error = ref(null)
    let refreshInterval = null

    // Computed
    const selectedItem = computed(() => {
      return watchlist.value.find(item => item.id === selectedWatchlistId.value)
    })

    const selectedAlerts = computed(() => {
      if (!selectedWatchlistId.value) return []
      return alerts.value.filter(a => a.watchlist_id === selectedWatchlistId.value)
    })

    // Methods
    const loadWatchlist = async () => {
      try {
        loading.value = true
        const response = await watchlistAPI.getAll()
        watchlist.value = response.data

        // Auto-select first item
        if (watchlist.value.length > 0 && !selectedWatchlistId.value) {
          selectedWatchlistId.value = watchlist.value[^5_0].id
        }
      } catch (err) {
        error.value = 'Failed to load watchlist: ' + err.message
      } finally {
        loading.value = false
      }
    }
  }
}
```

```

const loadAlerts = async () => {
  try {
    const response = await alertAPI.getAll()
    alerts.value = response.data
  } catch (err) {
    console.error('Failed to load alerts:', err)
  }
}

const loadChartData = async (symbol) => {
  try {
    const response = await priceAPI.getHistorical(symbol, '1mo')
    chartData.value = response.data.data || []
  }

  // Update current price
  if (chartData.value.length > 0) {
    currentPrice.value = chartData.value[chartData.value.length - 1].close
  }
} catch (err) {
  console.error('Failed to load chart data:', err)
}
}

const selectWatchlistItem = (id) => {
  selectedWatchlistId.value = id
  const item = watchlist.value.find(w => w.id === id)
  if (item) {
    loadChartData(item.symbol)
  }
}

const addToWatchlist = async (symbol) => {
  try {
    const response = await watchlistAPI.add(symbol)
    watchlist.value.push(response.data)
    selectedWatchlistId.value = response.data.id
    loadChartData(symbol)
  } catch (err) {
    error.value = 'Failed to add to watchlist: ' + (err.response?.data?.error || err.message)
  }
}

const removeFromWatchlist = async (id) => {
  try {
    await watchlistAPI.remove(id)
    watchlist.value = watchlist.value.filter(w => w.id !== id)
    if (selectedWatchlistId.value === id) {
      selectedWatchlistId.value = watchlist.value.length > 0 ? watchlist.value[^5_0].id : null
    }
  } catch (err) {
    error.value = 'Failed to remove from watchlist: ' + err.message
  }
}

const addAlert = async (alertType, thresholdPrice) => {

```

```

    try {
      const response = await alertAPI.create(selectedWatchlistId.value, alertType, thresholds)
      alerts.value.push(response.data)
    } catch (err) {
      error.value = 'Failed to create alert: ' + (err.response?.data?.error || err.message)
    }
  }

  const deleteAlert = async (id) => {
    try {
      await alertAPI.delete(id)
      alerts.value = alerts.value.filter(a => a.id !== id)
    } catch (err) {
      error.value = 'Failed to delete alert: ' + err.message
    }
  }

  const resetAlert = async (id) => {
    try {
      const response = await alertAPI.reset(id)
      const idx = alerts.value.findIndex(a => a.id === id)
      if (idx >= 0) {
        alerts.value[idx] = response.data
      }
    } catch (err) {
      error.value = 'Failed to reset alert: ' + err.message
    }
  }

  // Lifecycle
  onMounted(async () => {
    await loadWatchlist()
    await loadAlerts()

    // Refresh every 30 seconds
    refreshInterval = setInterval(async () => {
      await loadAlerts()
      if (selectedItem.value) {
        await loadChartData(selectedItem.value.symbol)
      }
    }, 30000)
  })

  onUnmounted(() => {
    if (refreshInterval) clearInterval(refreshInterval)
  })

  return {
    watchlist,
    alerts,
    selectedWatchlistId,
    selectedItem,
    selectedAlerts,
    currentPrice,
    chartData,
    loading,
  }
}

```

```
        error,
        selectWatchlistItem,
        addToWatchlist,
        removeFromWatchlist,
        addAlert,
        deleteAlert,
        resetAlert
    }
}
}

</script>

<style scoped>
.app-container {
    display: flex;
    flex-direction: column;
    height: 100vh;
    background: #f5f7fa;
    font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto, sans-serif;
}

.header {
    background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
    color: white;
    padding: 2rem;
    box-shadow: 0 2px 8px rgba(0, 0, 0, 0.1);
}

.header-content h1 {
    margin: 0;
    font-size: 28px;
    font-weight: 600;
}

.subtitle {
    margin: 0.5rem 0 0 0;
    opacity: 0.9;
    font-size: 14px;
}

.main-content {
    display: grid;
    grid-template-columns: 250px 1fr 300px;
    gap: 1.5rem;
    padding: 1.5rem;
    flex: 1;
    overflow: hidden;
}

.left-panel {
    overflow-y: auto;
    background: white;
    border-radius: 8px;
    box-shadow: 0 1px 3px rgba(0, 0, 0, 0.1);
}
```

```

.center-panel {
  display: flex;
  flex-direction: column;
  gap: 1rem;
  overflow: hidden;
}

.right-panel {
  overflow-y: auto;
  background: white;
  border-radius: 8px;
  box-shadow: 0 1px 3px rgba(0, 0, 0, 0.1);
}

.loading, .error {
  position: fixed;
  bottom: 2rem;
  right: 2rem;
  padding: 1rem 1.5rem;
  border-radius: 8px;
  font-weight: 500;
  z-index: 1000;
}

.loading {
  background: #667eea;
  color: white;
}

.error {
  background: #ef4444;
  color: white;
}

@media (max-width: 1200px) {
  .main-content {
    grid-template-columns: 1fr;
    gap: 1rem;
  }

  .left-panel, .right-panel {
    max-height: 300px;
  }
}
</style>

```

9. src/components/WatchlistPanel.vue

```

<template>
  <div class="watchlist-panel">
    <!-- Add New Stock -->
    <div class="add-stock-form">
      <input
        v-model="newSymbol"
        @keyup.enter="handleAdd"

```

```

        type="text"
        placeholder="Add symbol (e.g., VOO)"
        class="search-input"
      />
      <button @click="handleAdd" class="btn-add">+</button>
    </div>

    <!-- Watchlist Items -->
    <div class="watchlist-items">
      <div
        v-for="item in items"
        :key="item.id"
        :class="['watchlist-item', { active: selectedId === item.id }]"
        @click="$emit('select', item.id)"
      >
        <div class="item-header">
          <span class="symbol">{{ item.symbol }}</span>
          <button
            @click.stop="$emit('remove', item.id)"
            class="btn-remove"
            title="Remove"
          >
            ×
          </button>
        </div>
        <div class="item-meta">
          <span class="name">{{ item.name }}</span>
          <span class="alerts-count" v-if="item.alerts_count > 0">
            {{ item.alerts_count }} alert{{ item.alerts_count !== 1 ? 's' : '' }}
          </span>
        </div>
      </div>
    </div>

    <div v-if="items.length === 0" class="empty-state">
      <p>No stocks in watchlist</p>
      <p class="hint">Add one above to get started</p>
    </div>
  </div>
</template>

<script>
import { ref } from 'vue'

export default {
  name: 'WatchlistPanel',
  props: {
    items: Array,
    selectedId: Number
  },
  emits: ['select', 'add', 'remove'],
  setup(props, { emit }) {
    const newSymbol = ref('')

```

```
        emit('add', newSymbol.value.toUpperCase())
        newSymbol.value = ''
    }

}

return {
    newSymbol,
    handleAdd
}
}

</script>

<style scoped>
.watchlist-panel {
    display: flex;
    flex-direction: column;
    height: 100%;
    padding: 1rem;
}

.add-stock-form {
    display: flex;
    gap: 0.5rem;
    margin-bottom: 1rem;
}

.search-input {
    flex: 1;
    padding: 0.5rem;
    border: 1px solid #ddd;
    border-radius: 4px;
    font-size: 14px;
}

.btn-add {
    padding: 0.5rem 1rem;
    background: #667eea;
    color: white;
    border: none;
    border-radius: 4px;
    cursor: pointer;
    font-weight: 600;
    transition: background 0.2s;
}

.btn-add:hover {
    background: #5568d3;
}

.watchlist-items {
    flex: 1;
    overflow-y: auto;
}

.watchlist-item {
```

```
padding: 0.75rem;
margin-bottom: 0.5rem;
background: #f9fafb;
border-radius: 4px;
cursor: pointer;
transition: all 0.2s;
border: 2px solid transparent;
}

.watchlist-item:hover {
  background: #f3f4f6;
}

.watchlist-item.active {
  background: #ede9fe;
  border-color: #667eea;
}

.item-header {
  display: flex;
  justify-content: space-between;
  align-items: center;
  margin-bottom: 0.25rem;
}

.symbol {
  font-weight: 600;
  color: #111;
  font-size: 14px;
}

.btn-remove {
  background: none;
  border: none;
  color: #666;
  cursor: pointer;
  font-size: 18px;
  padding: 0;
  width: 24px;
  height: 24px;
  display: flex;
  align-items: center;
  justify-content: center;
  transition: color 0.2s;
}

.btn-remove:hover {
  color: #ef4444;
}

.item-meta {
  font-size: 12px;
  display: flex;
  justify-content: space-between;
}
```

```

.name {
  color: #666;
  white-space: nowrap;
  overflow: hidden;
  text-overflow: ellipsis;
}

.alerts-count {
  background: #ef4444;
  color: white;
  padding: 0.125rem 0.375rem;
  border-radius: 10px;
  font-weight: 600;
  flex-shrink: 0;
}

.empty-state {
  text-align: center;
  padding: 2rem 1rem;
  color: #999;
}

.empty-state p {
  margin: 0.5rem 0;
}

.hint {
  font-size: 12px;
}
</style>

```

10. src/components/PriceChart.vue

```

<template>
  <div class="price-chart">
    <div class="chart-header">
      <h2>{{ symbol }} Price Chart</h2>
      <div class="chart-controls">
        <button
          v-for="period in ['1mo', '3mo', '6mo', '1y']"
          :key="period"
          :class="['period-btn', { active: activePeriod === period }]"
          @click="changePeriod(period)"
        >
          {{ period }}
        </button>
      </div>
    </div>

    <div class="chart-container">
      <canvas ref="chartCanvas"></canvas>
    </div>
  </div>
</template>

```

```
<script>
import { ref, onMounted, watch } from 'vue'
import Chart from 'chart.js/auto'
import { priceAPI } from '../api'

export default {
  name: 'PriceChart',
  props: {
    symbol: String,
    chartData: Array
  },
  setup(props) {
    const chartCanvas = ref(null)
    let chartInstance = null
    const activePeriod = ref('1mo')

    const changePeriod = async (period) => {
      activePeriod.value = period
      try {
        const response = await priceAPI.getHistorical(props.symbol, period)
        updateChart(response.data.data)
      } catch (err) {
        console.error('Failed to fetch chart data:', err)
      }
    }

    const updateChart = (data) => {
      if (!chartInstance || !chartCanvas.value) return

      const dates = data.map(d => d.date)
      const prices = data.map(d => d.close)

      chartInstance.data.labels = dates
      chartInstance.data.datasets[0].data = prices
      chartInstance.update()
    }

    onMounted(() => {
      if (!chartCanvas.value) return

      const ctx = chartCanvas.value.getContext('2d')

      const dates = props.chartData.map(d => d.date)
      const prices = props.chartData.map(d => d.close)

      chartInstance = new Chart(ctx, {
        type: 'line',
        data: {
          labels: dates,
          datasets: [
            {
              label: `${props.symbol} Price (USD)`,
              data: prices,
              borderColor: '#667eea',
              backgroundColor: 'rgba(102, 126, 234, 0.1)',
              borderWidth: 2,
            }
          ]
        }
      })
    })
  }
}
```

```

        tension: 0.1,
        fill: true,
        pointRadius: 0,
        pointHoverRadius: 6
    }
]
},
options: {
    responsive: true,
    maintainAspectRatio: false,
    plugins: {
        legend: {
            display: true,
            position: 'top'
        }
    },
    scales: {
        y: {
            beginAtZero: false,
            ticks: {
                callback: function(value) {
                    return '$' + value.toFixed(2)
                }
            }
        }
    }
}
}

watch(() => props.chartData, (newData) => {
    if (newData && newData.length > 0) {
        updateChart(newData)
    }
})

return {
    chartCanvas,
    activePeriod,
    changePeriod
}
}
}
</script>

<style scoped>
.price-chart {
    display: flex;
    flex-direction: column;
    flex: 1;
    background: white;
    border-radius: 8px;
    padding: 1.5rem;
    box-shadow: 0 1px 3px rgba(0, 0, 0, 0.1);
    overflow: hidden;
}

```

```
.chart-header {
  display: flex;
  justify-content: space-between;
  align-items: center;
  margin-bottom: 1rem;
}

.chart-header h2 {
  margin: 0;
  font-size: 18px;
  color: #111;
}

.chart-controls {
  display: flex;
  gap: 0.5rem;
}

.period-btn {
  padding: 0.375rem 0.75rem;
  border: 1px solid #ddd;
  background: white;
  border-radius: 4px;
  cursor: pointer;
  font-size: 12px;
  font-weight: 500;
  transition: all 0.2s;
}

.period-btn:hover {
  border-color: #667eea;
  color: #667eea;
}

.period-btn.active {
  background: #667eea;
  color: white;
  border-color: #667eea;
}

.chart-container {
  flex: 1;
  position: relative;
  min-height: 300px;
}

canvas {
  max-height: 100%;
}

</style>
```

11. src/components/StockInfoCard.vue

```
<template>
  <div class="stock-info-card">
    <div class="info-header">
      <div class="stock-title">
        <h1>{{ item.symbol }}</h1>
        <p class="stock-name">{{ item.name }}</p>
      </div>
      <div class="price-display">
        <div class="current-price">{{ formatCurrency(currentPrice) }}</div>
        <div class="price-change">
          <span v-if="priceChange >= 0" class="positive">
            ▲ {{ formatNumber(priceChange) }}
          </span>
          <span v-else class="negative">
            ▼ {{ formatNumber(priceChange) }}
          </span>
        </div>
      </div>
    </div>
  </div>
</template>

<script>
import { computed } from 'vue'
import { formatCurrency, formatNumber } from '../utils/formatters'

export default {
  name: 'StockInfoCard',
  props: {
    item: Object,
    currentPrice: Number
  },
  setup(props) {
    const priceChange = computed(() => {
      if (!props.currentPrice || !props.item.price_at_add) return 0
      return props.currentPrice - props.item.price_at_add
    })

    return {
      priceChange,
      formatCurrency,
      formatNumber
    }
  }
}
</script>

<style scoped>
.stock-info-card {
  background: white;
  border-radius: 8px;
  padding: 1.5rem;
  box-shadow: 0 1px 3px rgba(0, 0, 0, 0.1);
}
```

```

.info-header {
  display: flex;
  justify-content: space-between;
  align-items: center;
}

.stock-title h1 {
  margin: 0;
  font-size: 24px;
  color: #111;
}

.stock-name {
  margin: 0.25rem 0 0 0;
  color: #666;
  font-size: 14px;
}

.price-display {
  text-align: right;
}

.current-price {
  font-size: 28px;
  font-weight: 700;
  color: #111;
}

.price-change {
  margin-top: 0.5rem;
  font-size: 14px;
  font-weight: 600;
}

.positive {
  color: #10b981;
}

.negative {
  color: #ef4444;
}

```

12. src/components/AlertsPanel.vue

```

<template>
  <div class="alerts-panel">
    <h3>Price Alerts</h3>

    <!-- Create New Alert -->
    <div class="create-alert">
      <div class="form-group">
        <label>Alert Type</label>
        <select v-model="newAlert.type" class="form-input">

```

```

        <option value="price_above">Price Above</option>
        <option value="price_below">Price Below</option>
    </select>
</div>

<div class="form-group">
    <label>Threshold ($)</label>
    <input
        v-model.number="newAlert.threshold"
        type="number"
        step="0.01"
        placeholder="150.00"
        class="form-input"
    />
</div>

<button @click="handleCreateAlert" class="btn-create">
    Create Alert
</button>
</div>

<!-- Alerts List --&gt;
&lt;div class="alerts-list"&gt;
    &lt;div
        v-for="alert in alerts"
        :key="alert.id"
        :class="['alert-item', { triggered: alert.triggered }]"
    &gt;
        &lt;div class="alert-info"&gt;
            &lt;span class="alert-type"&gt;
                {{ alert.alert_type === 'price_above' ? '↑' : '↓' }}
                {{ alert.alert_type === 'price_above' ? 'Above' : 'Below' }}
            &lt;/span&gt;
            &lt;span class="alert-price"&gt;${{ alert.threshold_price }}&lt;/span&gt;
        &lt;/div&gt;

        &lt;div class="alert-actions"&gt;
            &lt;button
                v-if="alert.triggered"
                @click="$emit('reset-alert', alert.id)"
                class="btn-reset"
                title="Reset"
            &gt;
                ✎
            &lt;/button&gt;
            &lt;button
                @click="$emit('delete-alert', alert.id)"
                class="btn-delete"
                title="Delete"
            &gt;
                ✘
            &lt;/button&gt;
        &lt;/div&gt;
    &lt;/div&gt;
&lt;/div&gt;

&lt;div v-if="alerts.length === 0" class="empty"&gt;
</pre>

```

```
        No alerts. Create one above!
    </div>
    </div>
</div>
</template>

<script>
import { ref } from 'vue'

export default {
  name: 'AlertsPanel',
  props: {
    watchlistItem: Object,
    alerts: Array
  },
  emits: ['add-alert', 'delete-alert', 'reset-alert'],
  setup(props, { emit }) {
    const newAlert = ref({
      type: 'price_above',
      threshold: null
    })

    const handleCreateAlert = () => {
      if (!newAlert.value.threshold) {
        alert('Please enter a threshold price')
        return
      }

      emit('add-alert', newAlert.value.type, newAlert.value.threshold)
      newAlert.value.threshold = null
    }

    return {
      newAlert,
      handleCreateAlert
    }
  }
}
</script>

<style scoped>
.alerts-panel {
  display: flex;
  flex-direction: column;
  height: 100%;
  padding: 1rem;
}

.alerts-panel h3 {
  margin: 0 0 1rem 0;
  font-size: 16px;
  color: #111;
}

.create-alert {
  padding: 1rem;
```

```
background: #f9fafb;
border-radius: 4px;
margin-bottom: 1rem;
}

.form-group {
  margin-bottom: 0.75rem;
}

.form-group label {
  display: block;
  font-size: 12px;
  font-weight: 600;
  margin-bottom: 0.25rem;
  color: #555;
}

.form-input {
  width: 100%;
  padding: 0.5rem;
  border: 1px solid #ddd;
  border-radius: 4px;
  font-size: 13px;
}

.btn-create {
  width: 100%;
  padding: 0.5rem;
  background: #667eea;
  color: white;
  border: none;
  border-radius: 4px;
  font-weight: 600;
  cursor: pointer;
  transition: background 0.2s;
}

.btn-create:hover {
  background: #5568d3;
}

.alerts-list {
  flex: 1;
  overflow-y: auto;
}

.alert-item {
  display: flex;
  justify-content: space-between;
  align-items: center;
  padding: 0.75rem;
  margin-bottom: 0.5rem;
  background: #f9fafb;
  border-radius: 4px;
  border-left: 3px solid #667eea;
  transition: all 0.2s;
}
```

```
}

.alert-item.triggered {
  background: #fef3c7;
  border-left-color: #f59e0b;
}

.alert-info {
  display: flex;
  flex-direction: column;
  gap: 0.25rem;
  flex: 1;
}

.alert-type {
  font-size: 12px;
  font-weight: 600;
  color: #555;
}

.alert-price {
  font-size: 14px;
  font-weight: 700;
  color: #111;
}

.alert-actions {
  display: flex;
  gap: 0.5rem;
}

.btn-reset, .btn-delete {
  background: none;
  border: none;
  cursor: pointer;
  font-size: 14px;
  padding: 0.25rem 0.5rem;
  transition: transform 0.2s;
}

.btn-reset:hover {
  transform: scale(1.2);
}

.btn-delete:hover {
  color: #ef4444;
  transform: scale(1.2);
}

.empty {
  text-align: center;
  padding: 1rem;
  color: #999;
  font-size: 13px;
}

</style>
```

13. src/assets/styles.css

```
* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}

html, body, #app {
  width: 100%;
  height: 100%;
}

body {
  font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
}

button {
  font-family: inherit;
}

input, select, textarea {
  font-family: inherit;
}

/* Scrollbar styling */
::-webkit-scrollbar {
  width: 6px;
  height: 6px;
}

::-webkit-scrollbar-track {
  background: #f1f5f9;
}

::-webkit-scrollbar-thumb {
  background: #cbd5e1;
  border-radius: 3px;
}

::-webkit-scrollbar-thumb:hover {
  background: #94a3b8;
}
```

14. .gitignore

```
# Dependencies
node_modules/
.pnp
.pnp.js

# Production
```

```
dist/  
build/  
  
# Env  
.env  
.env.local  
.env.*.local  
  
# Editor  
.vscode/  
.idea/  
*.swp  
*.swo  
  
# OS  
.DS_Store  

```

15. README.md

```
# Stock Price Alert - Frontend  
  
Vue.js + Chart.js frontend for real-time stock price alerting.  
  
## Setup  
  
```bash  
npm install
```

## Development

```
npm run dev
```

Runs on <http://localhost:5173> by default.

## Configuration

Create `.env` with:

```
VITE_API_URL=http://localhost:5000
```

## Build

```
npm run build
```

## Features

- ⌚ Real-time price charts with Chart.js
- ⌚ Create price alerts (above/below thresholds)
- ⌚ Track multiple stocks/ETFs simultaneously
- ⚡ Fast, responsive UI with Vue 3

## Project Structure

- src/components/ - Vue components
- src/api.js - Backend API client
- src/utils/ - Utility functions

```

```

```
Quick Start Guide

Backend Setup

```bash
# 1. Create project directory
mkdir stock-price-alert-backend
cd stock-price-alert-backend

# 2. Create Python virtual environment
python -m venv venv
source venv/bin/activate # On Windows: venv\Scripts\activate

# 3. Install dependencies
pip install -r requirements.txt

# 4. Create .env from template
cp .env.example .env
# Edit .env with your Telegram credentials

# 5. Run Flask app
python app.py

```

App runs on <http://localhost:5000>

Frontend Setup

```
# 1. Create project directory
mkdir stock-price-alert-frontend
cd stock-price-alert-frontend

# 2. Install dependencies
npm install

# 3. Create .env from template
cp .env.example .env

# 4. Run dev server
npm run dev
```

App runs on <http://localhost:5173>

API Endpoints (Backend Reference)

```
GET   /api/watchlist           # Get all watchlist items
POST  /api/watchlist          # Add to watchlist { symbol }
GET   /api/watchlist/:id       # Get single item with current price
DELETE /api/watchlist/:id     # Remove from watchlist

GET   /api/alerts              # Get all alerts
POST  /api/alerts              # Create alert { watchlist_id, alert_type, threshold_
DELETE /api/alerts/:id         # Delete alert
POST  /api/alerts/:id/reset    # Reset triggered alert

GET   /api/prices/:symbol      # Get current price
GET   /api/prices/:symbol/historical # Get historical prices (period=1mo|3mo|6mo|1y)
GET   /api/prices/:symbol/info  # Get stock info

GET   /health                  # Health check
```

Next Steps

1. Configure Telegram:

- Create bot with BotFather on Telegram
- Get token and chat ID
- Add to .env

2. Test Full Flow:

- Add stock to watchlist
- Create price alert
- Monitor scheduler logs
- Receive Telegram notification when price reaches threshold

3. Deploy:

- Backend → Railway (`git push railway main`)
- Frontend → Netlify (`npm run build` → drag dist/)

That's your complete POC scaffold! ☺

**

Now update the technical report and architecture diagram with the new stack selected and why we chose such technologies.

Perfect! I've successfully created comprehensive updated documentation for your tech stack transition. Here's what I've delivered:

▪ Summary of Updates

I've created **4 comprehensive documents** to replace and update your original technical report and architecture:

1. tech-report-updated.md ★ Main Document

- Complete technical decision report for the Flask + Python backend
- Detailed section on why yfinance over FMP API
- Why Flask over FastAPI for your specific POC constraints
- Database migration path (SQLite → PostgreSQL)
- Complete architecture breakdown
- Risk analysis and mitigation strategies
- Success metrics for POC completion

2. architecture-diagram.txt (ASCII Visual)

- Complete system architecture in ASCII format (printable/shareable)
- Shows all layers: CLIENT → PRESENTATION → APPLICATION → DATA
- Data flow annotations (HTTP/REST, SQL Queries, Telegram API)
- Deployment architecture for Railway
- Technology decision rationale table
- Migration path visualization for future scaling

3. stack-update-summary.md

- Quick reference: Original vs Updated stack
- Cost comparison (\$360/year → \$120/year savings)
- Risk analysis with mitigation strategies
- Migration effort estimates (if POC → MVP)
- Developer experience comparison
- Side-by-side setup instructions

4. stack-comparison.md

- Detailed side-by-side technical comparison
- Code examples showing differences
- Performance metrics (setup time, feature development)
- Scalability matrix
- Decision matrix with weighted scoring
- Complete files checklist

□ Key Points Documented

✓ Why Flask instead of FastAPI:

- Sync-first simplicity for single-user POC
- Direct yfinance integration (no async wrapper needed)
- 2 hours faster setup
- Proven for small apps

✓ Why yfinance over FMP API:

- **\$0 cost vs \$20+/month** (eliminates original constraint)
- 15,000+ global symbols (vs US-only FMP)
- Unrestricted rate limits
- Trade-off: 15-20 min delayed data (acceptable for long-term investors)

✓ What stayed the same:

- Vue.js 3 + Chart.js frontend (unchanged)
- SQLite database (ORM abstracts for future migration)
- APScheduler (same polling pattern as node-cron)
- Telegram alerts (unchanged)
- Railway hosting (unchanged)

✓ Clear migration path:

- If MVP grows: Flask → FastAPI (1-2 days)
- If data scales: SQLite → PostgreSQL (2-4 hours, ORM handles it)
- If scheduler needs distribution: APScheduler → Celery (1-2 days)

Cost Impact

Item	Original	Updated	Savings
Stock API	\$240/yr	\$0	\$240 ✓
Total POC Cost	\$360/yr	\$120/yr	\$240/yr savings

All documents are ready for sharing with your team or stakeholders. They comprehensively explain the technical justification for the Python backend decision while maintaining the architectural integrity of your original plan.

Next steps: Begin development using the scaffolding code provided in the previous response! ☀

**

1. Perplexity-planning.pdf
2. [technical-report.md](#)
3. Perplexity planning.pdf
4. [technical-report.md](#)
5. Perplexity planning.pdf
6. [technical-report.md](#)
7. Perplexity planning.pdf
8. [technical-report.md](#)
9. StockAlarm-layout.jpg
10. Perplexity planning.pdf
11. [technical-report.md](#)
12. [technical-report.md](#)
13. architecture_diagram.jpg
14. Perplexity planning.pdf
15. [technical-report.md](#)
16. StockAlarm-layout.jpg