Full-Stack Python Portfolio Optimization Application

Generate a full-stack application for portfolio optimization with a Python web frontend and Python backend. The application should allow users to evaluate a list of stock tickers based on market capitalization and average daily volume, then visualize the Efficient Frontier and portfolio analysis results.

Frontend Requirements (Python Web Framework):

Framework Options:

- **Streamlit** (Recommended for rapid development)
- **Dash by Plotly** (For advanced interactive visualizations)
- **Flask with Jinja2 templates** (For more control over HTML/CSS)
- **Django** (For enterprise-level applications)

User Interface Features:

1. **User Input Options:**
    - Allow users to upload a `.txt` file containing tickers (one per line) **or** manually input tickers in a text area
    - Provide input fields for:
        - Minimum/maximum market capitalization (in billions)
        - Minimum/maximum average daily volume (in millions)
    - Include validation for inputs (e.g., numeric values, non-empty ticker list)
2. **Interactive UI:**
    - Display a responsive layout with:
        - A form for ticker input and filter criteria
        - A loading spinner/progress bar while fetching/processing data
        - Interactive visualizations for the Efficient Frontier using Plotly
        - Tabular results showing:
            - Filtered tickers with market cap and volume data
            - Optimal portfolios (min volatility, max Sharpe ratio)
    - Allow users to toggle between graphical and tabular views
    - Responsive design that works on desktop and mobile

3. **Export Functionality:**
    - Provide buttons to download:
        - The Efficient Frontier plot as PNG/HTML
        - Portfolio analysis results as CSV
        - Complete analysis report as PDF

Backend Requirements (Python):

API Structure:

- **For Streamlit/Dash:** Built-in state management and callbacks
- **For Flask/Django:** RESTful API endpoints:
    - `/api/optimize` (POST): Accepts tickers, market cap, and volume filters; returns optimization results
    - `/api/generate_report` (POST): Generates downloadable reports from results
    - `/api/validate_tickers` (POST): Validates ticker symbols before processing

Portfolio Optimization Logic:

1. **Data Retrieval:**
    - Use `yfinance` or `Alpha Vantage` for historical stock data
    - Fetch current market cap and average volume data
    - Handle missing data and API rate limits gracefully
2. **Filtering and Processing:**
    - Filter tickers based on user-provided market cap (billions) and volume (millions)
    - Calculate expected returns using historical data
    - Compute covariance matrix for risk assessment
    - Generate the Efficient Frontier using portfolio optimization techniques
3. **Optimization Calculations:**
    - Identify optimal portfolios:
        - Minimum volatility portfolio
        - Maximum Sharpe ratio portfolio
        - Risk parity portfolio (optional)
    - Calculate portfolio metrics (returns, volatility, Sharpe ratio)
4. **Error Handling:**

- Validate ticker symbols and return meaningful error messages
- Handle API rate limits with retry logic and caching
- Graceful degradation for missing data points

Required Python Libraries:

Frontend (depending on chosen framework):

- **Streamlit:** `streamlit`, `streamlit-plotly`, `streamlit-aggrid`
- **Dash:** `dash`, `dash-bootstrap-components`, `dash-table`
- **Flask:** `flask`, `wtforms`, `flask-wtf`, `jinja2`
- **Django:** `django`, `django-crispy-forms`, `django-tables2`

Backend & Analysis:

- **Data & APIs:** `yfinance`, `pandas`, `numpy`, `requests`
- **Optimization:** `scipy`, `cvxpy`, `pypfopt` (Portfolio Optimization library)
- **Visualization:** `plotly`, `matplotlib`, `seaborn`
- **Report Generation:** `reportlab`, `weasyprint`, `fpdf`
- **Utilities:** `python-dotenv`, `cachetools`, `asyncio`

Optional Enhancements:

- **Database:** `sqlite3`, `sqlalchemy` (for caching historical data)
- **Task Queue:** `celery`, `redis` (for background processing)
- **Testing:** `pytest`, `unittest`

Application Architecture:

Recommended Structure:

```
portfolio_optimizer/
├── app/
│   ├── __init__.py
│   ├── main.py              # Main application entry point
│   ├── data_fetcher.py      # Stock data retrieval logic
│   ├── optimizer.py         # Portfolio optimization algorithms
│   ├── visualizer.py        # Chart and plot generation
│   ├── report_generator.py  # PDF/CSV export functionality
│   └── utils.py            # Helper functions
├── data/
│   └── cache/              # Cached stock data
├── static/                 # CSS, JS, images (if using Flask/Django)
├── templates/              # HTML templates (if using Flask/Django)
├── tests/
├── requirements.txt
├── config.py
└── README.md
```

Key Features to Implement:

1. **Real-time Data Processing:**
    - Asynchronous data fetching for multiple tickers
    - Progress tracking for long-running optimizations
    - Caching mechanism for frequently requested data
2. **Advanced Visualizations:**
    - Interactive Efficient Frontier plot with hover information
    - Risk-return scatter plot for individual stocks
    - Portfolio composition pie charts
    - Historical performance comparison charts
3. **User Experience:**
    - Input validation with helpful error messages
    - Responsive design for mobile devices
    - Dark/light theme toggle (optional)
    - Save/load analysis sessions
4. **Performance Optimization:**
    - Data caching to reduce API calls
    - Lazy loading for large datasets

- Optimized mathematical calculations using NumPy

Deliverables:

1. **Complete Python Application:**
   - Fully functional web interface using chosen Python framework
   - Clean, documented code following Python best practices
   - Configuration file for easy deployment
2. **Documentation:**
   - Comprehensive README with setup instructions
   - API documentation (if using Flask/Django)
   - Code comments explaining optimization algorithms
   - User guide with screenshots
3. **Educational Content:**
   - Explanation of how market cap and volume influence filtering/optimization
   - Documentation of the mathematical models used
   - Interpretation guide for the Efficient Frontier results
4. **Testing & Examples:**
   - Unit tests for core optimization functions
   - Example datasets and configuration files
   - Demo screenshots or video walkthrough

Deployment Considerations:

- **Local Development:** Instructions for running locally with sample data
- **Cloud Deployment:** Configuration for platforms like Heroku, Railway, or Streamlit Cloud
- **Docker Support:** Containerization for consistent deployment
- **Environment Variables:** Secure handling of API keys and configuration

This Python-based approach leverages the rich ecosystem of financial and data science libraries available in Python while providing multiple options for the web interface depending on your specific needs and preferences.