Multi-Year Financial Data API Guide

Learn how to access multiple years of financial statements and serve them through FastAPI endpoints for web applications and data analysis.

Overview

This guide demonstrates how to retrieve historical financial data (income statement, balance sheet, and cash flow) for multiple years and expose it through professional FastAPI endpoints. Perfect for building financial dashboards, analysis tools, or data APIs.

Quick Start

```
from edgar import Company
from fastapi import FastAPI, HTTPException
from typing import List, Dict, Any
import pandas as pd

app = FastAPI(title="Financial Data API")

# Get multi-year financial data
company = Company('AAPL')

# Income statement - 5 years of annual data
income_stmt = company.income_statement(periods=5, annual=True)

# Balance sheet - 5 years of annual data
balance_sheet = company.balance_sheet(periods=5, annual=True)

# Cash flow - 5 years of annual data
cash_flow = company.cash_flow(periods=5, annual=True)

print(f"Retrieved {len(income_stmt.periods)} periods of data")
```

Core Data Retrieval Patterns

Multi-Year Annual Statements

```
def get_multi_year_financials(ticker: str, years: int = 5):
    """Get multiple years of financial statements"""
    company = Company(ticker)

    if not company.facts:
```

latest

```
        return None

    # Get annual data for specified years
    financial_data = {
        'company': {
            'name': company.name,
            'ticker': ticker,
            'shares_outstanding': company.shares_outstanding,
            'public_float': company.public_float
        },
        'income_statement': company.income_statement(periods=years,
annual=True),
        'balance_sheet': company.balance_sheet(periods=years, annual=True),
        'cash_flow': company.cash_flow(periods=years, annual=True)
    }

    return financial_data

# Example usage
aapl_data = get_multi_year_financials('AAPL', 7)
print(f"Retrieved data for periods: {aapl_data['income_statement'].periods}")
```

Mixed Period Analysis

```
def get_comprehensive_data(ticker: str):
    """Get both annual and quarterly data for trend analysis"""
    company = Company(ticker)

    if not company.facts:
        return None

    return {
        'annual': {
            'income': company.income_statement(periods=5, annual=True),
            'balance': company.balance_sheet(periods=5, annual=True),
            'cashflow': company.cash_flow(periods=5, annual=True)
        },
        'quarterly': {
            'income': company.income_statement(periods=12, annual=False),
            'balance': company.balance_sheet(periods=12, annual=False),
            'cashflow': company.cash_flow(periods=12, annual=False)
        }
    }

# Get comprehensive view
comprehensive = get_comprehensive_data('MSFT')
```

FastAPI Endpoint Implementation

latest

Basic Financial Data Endpoints

```
from fastapi import FastAPI, HTTPException, Query
from pydantic import BaseModel
from typing import Optional, Dict, Any, List
from edgar import Company
import json

app = FastAPI(
    title="Financial Data API",
    description="Access multi-year financial statements from SEC data",
    version="1.0.0"
)

class FinancialResponse(BaseModel):
    company_info: Dict[str, Any]
    periods: List[str]
    data: Dict[str, Any]
    metadata: Dict[str, Any]

@app.get("/financial/{ticker}/income", response_model=FinancialResponse)
async def get_income_statement(
    ticker: str,
    periods: int = Query(4, description="Number of periods", ge=1, le=20),
    annual: bool = Query(True, description="Annual (True) or Quarterly
(False)"),
    concise_format: bool = Query(False, description="Use concise formatting
($1.0B vs $1,000,000,000)")
):
    """Get income statement data for multiple periods"""
    try:
        company = Company(ticker.upper())

        if not company.facts:
            raise HTTPException(status_code=404, detail=f"No financial data
available for {ticker}")

        # Get income statement
        stmt = company.income_statement(periods=periods, annual=annual,
concise_format=concise_format)

        if not stmt:
            raise HTTPException(status_code=404, detail=f"No income statement
data for {ticker}")

        # Convert to API response format
        response_data = {
            'company_info': {
                'name': company.name,
                'ticker': ticker.upper(),
                'shares_outstanding': company.shares_outstanding,
                'public_float': company.public_float
            },
            'periods': stmt.periods,
            'data': _convert_statement_to_dict(stmt),
            'metadata': {
                'period_type': 'annual' if annual else 'quarterly',
                'concise_format': concise_format,
                'total_items': len(list(stmt.iter_with_values()))
```

```
            }
        }

        return FinancialResponse(**response_data)

    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@app.get("/financial/{ticker}/balance", response_model=FinancialResponse)
async def get_balance_sheet(
    ticker: str,
    periods: int = Query(4, description="Number of periods", ge=1, le=20),
    annual: bool = Query(True, description="Annual (True) or Quarterly
(False)"),
    concise_format: bool = Query(False, description="Use concise formatting")
):
    """Get balance sheet data for multiple periods"""
    try:
        company = Company(ticker.upper())

        if not company.facts:
            raise HTTPException(status_code=404, detail=f"No financial data
available for {ticker}")

        stmt = company.balance_sheet(periods=periods, annual=annual,
concise_format=concise_format)

        if not stmt:
            raise HTTPException(status_code=404, detail=f"No balance sheet
data for {ticker}")

        response_data = {
            'company_info': {
                'name': company.name,
                'ticker': ticker.upper(),
                'shares_outstanding': company.shares_outstanding,
                'public_float': company.public_float
            },
            'periods': stmt.periods,
            'data': _convert_statement_to_dict(stmt),
            'metadata': {
                'period_type': 'annual' if annual else 'quarterly',
                'concise_format': concise_format,
                'total_items': len(list(stmt.iter_with_values()))
            }
        }

        return FinancialResponse(**response_data)

    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@app.get("/financial/{ticker}/cashflow", response_model=Fin
async def get_cash_flow(
    ticker: str,
    periods: int = Query(4, description="Number of periods", ge=1, le=20),
    annual: bool = Query(True, description="Annual (True) or Quarterly
```

```
(False)"),
    concise_format: bool = Query(False, description="Use concise formatting")
):
    """Get cash flow statement data for multiple periods"""
    try:
        company = Company(ticker.upper())

        if not company.facts:
            raise HTTPException(status_code=404, detail=f"No financial data
available for {ticker}")

        stmt = company.cash_flow(periods=periods, annual=annual,
concise_format=concise_format)

        if not stmt:
            raise HTTPException(status_code=404, detail=f"No cash flow data
for {ticker}")

        response_data = {
            'company_info': {
                'name': company.name,
                'ticker': ticker.upper(),
                'shares_outstanding': company.shares_outstanding,
                'public_float': company.public_float
            },
            'periods': stmt.periods,
            'data': _convert_statement_to_dict(stmt),
            'metadata': {
                'period_type': 'annual' if annual else 'quarterly',
                'concise_format': concise_format,
                'total_items': len(list(stmt.iter_with_values()))
            }
        }

        return FinancialResponse(**response_data)

    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

def _convert_statement_to_dict(stmt):
    """Convert statement to API-friendly dictionary format"""
    data = {}

    for item in stmt.iter_with_values():
        # Create item data with all periods
        item_data = {
            'label': item.label,
            'concept': item.concept,
            'values': {},
            'is_total': getattr(item, 'is_total', False),
            'depth': getattr(item, 'depth', 0)
        }

        # Add values for each period
        for period in stmt.periods:
            value = item.values.get(period)
            if value is not None:
```

```
                item_data['values'][period] = {
                    'raw_value': value,
                    'display_value': item.get_display_value(period)
                }

        data[item.concept] = item_data

    return data
```

Comprehensive Multi-Statement Endpoint

```
class ComprehensiveFinancialResponse(BaseModel):
    company_info: Dict[str, Any]
    periods: List[str]
    income_statement: Dict[str, Any]
    balance_sheet: Dict[str, Any]
    cash_flow: Dict[str, Any]
    key_metrics: Dict[str, Any]
    metadata: Dict[str, Any]

@app.get("/financial/{ticker}/comprehensive",
response_model=ComprehensiveFinancialResponse)
async def get_comprehensive_financials(
    ticker: str,
    periods: int = Query(5, description="Number of periods", ge=1, le=10),
    annual: bool = Query(True, description="Annual (True) or Quarterly
(False)"),
    concise_format: bool = Query(False, description="Use concise formatting"),
    include_ratios: bool = Query(True, description="Calculate financial
ratios")
):
    """Get comprehensive financial data including all statements and key
metrics"""
    try:
        company = Company(ticker.upper())

        if not company.facts:
            raise HTTPException(status_code=404, detail=f"No financial data
available for {ticker}")

        # Get all three statements
        income_stmt = company.income_statement(periods=periods, annual=annual,
concise_format=concise_format)
        balance_sheet = company.balance_sheet(periods=periods, annual=annual,
concise_format=concise_format)
        cash_flow = company.cash_flow(periods=periods, annual=annual,
concise_format=concise_format)

        # Get periods from the first available statement
        available_periods = []
        if income_stmt:
            available_periods = income_stmt.periods
        elif balance_sheet:
            available_periods = balance_sheet.periods
        elif cash_flow:
```

```
            available_periods = cash_flow.periods

        if not available_periods:
            raise HTTPException(status_code=404, detail=f"No financial
statement data available for {ticker}")

        # Calculate key metrics if requested
        key_metrics = {}
        if include_ratios and income_stmt and balance_sheet:
            key_metrics = _calculate_financial_ratios(income_stmt,
balance_sheet, cash_flow)

        response_data = {
            'company_info': {
                'name': company.name,
                'ticker': ticker.upper(),
                'shares_outstanding': company.shares_outstanding,
                'public_float': company.public_float
            },
            'periods': available_periods,
            'income_statement': _convert_statement_to_dict(income_stmt) if
income_stmt else {},
            'balance_sheet': _convert_statement_to_dict(balance_sheet) if
balance_sheet else {},
            'cash_flow': _convert_statement_to_dict(cash_flow) if cash_flow
else {},
            'key_metrics': key_metrics,
            'metadata': {
                'period_type': 'annual' if annual else 'quarterly',
                'concise_format': concise_format,
                'statements_available': {
                    'income_statement': income_stmt is not None,
                    'balance_sheet': balance_sheet is not None,
                    'cash_flow': cash_flow is not None
                }
            }
        }

        return ComprehensiveFinancialResponse(**response_data)

    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

def _calculate_financial_ratios(income_stmt, balance_sheet, cash_flow):
    """Calculate key financial ratios from statements"""
    ratios = {}

    try:
        # Get latest period
        if not income_stmt.periods:
            return ratios

        latest_period = income_stmt.periods[0]

        # Find key items
        revenue_item = income_stmt.find_item('Revenue')
        net_income_item = income_stmt.find_item('Net Income')
```

```
        total_assets_item = balance_sheet.find_item('Assets') if balance_sheet
else None
        total_equity_item = balance_sheet.find_item('Equity') if balance_sheet
else None

        # Calculate ratios for latest period
        if revenue_item and net_income_item:
            revenue = revenue_item.values.get(latest_period)
            net_income = net_income_item.values.get(latest_period)

            if revenue and net_income and revenue != 0:
                ratios[f'profit_margin_{latest_period.lower().replace(" ",
"_")}'] = net_income / revenue

        if net_income_item and total_assets_item:
            net_income = net_income_item.values.get(latest_period)
            total_assets = total_assets_item.values.get(latest_period)

            if net_income and total_assets and total_assets != 0:
                ratios[f'roa_{latest_period.lower().replace(" ", "_")}'] =
net_income / total_assets

        if net_income_item and total_equity_item:
            net_income = net_income_item.values.get(latest_period)
            total_equity = total_equity_item.values.get(latest_period)

            if net_income and total_equity and total_equity != 0:
                ratios[f'roe_{latest_period.lower().replace(" ", "_")}'] =
net_income / total_equity

    except Exception as e:
        # Log error but don't fail the request
        print(f"Error calculating ratios: {e}")

    return ratios
```

Historical Trend Analysis Endpoints

```
from datetime import datetime, timedelta

@app.get("/financial/{ticker}/trends")
async def get_financial_trends(
    ticker: str,
    metric: str = Query(..., description="Metric to analyze (revenue,
net_income, assets)"),
    years: int = Query(5, description="Number of years", ge=2, le=10)
):
    """Get historical trends for specific financial metrics"""
    try:
        company = Company(ticker.upper())

        if not company.facts:
            raise HTTPException(status_code=404, detail=f"No financial data
available for {ticker}")
```

```
        # Determine which statement to use based on metric
        if metric.lower() in ['revenue', 'net_income', 'operating_income']:
            stmt = company.income_statement(periods=years, annual=True)
        elif metric.lower() in ['assets', 'liabilities', 'equity']:
            stmt = company.balance_sheet(periods=years, annual=True)
        elif metric.lower() in ['operating_cash_flow', 'free_cash_flow']:
            stmt = company.cash_flow(periods=years, annual=True)
        else:
            raise HTTPException(status_code=400, detail=f"Unknown metric:
{metric}")

        if not stmt:
            raise HTTPException(status_code=404, detail=f"No data available
for metric: {metric}")

        # Find the requested metric
        metric_item = stmt.find_item(metric)
        if not metric_item:
            # Try alternative names
            alt_names = {
                'revenue': ['Revenues', 'Total Revenue', 'Net Sales'],
                'net_income': ['Net Income', 'Net Earnings', 'Net Income
(Loss)'],
                'assets': ['Total Assets', 'Assets'],
                'equity': ['Total Equity', 'Stockholders Equity', 'Total
Stockholders Equity']
            }

            for alt_name in alt_names.get(metric.lower(), []):
                metric_item = stmt.find_item(alt_name)
                if metric_item:
                    break

        if not metric_item:
            raise HTTPException(status_code=404, detail=f"Metric '{metric}'
not found in financial statements")

        # Calculate trends
        trend_data = []
        values = []

        for period in reversed(stmt.periods):  # Chronological order
            value = metric_item.values.get(period)
            if value is not None:
                trend_data.append({
                    'period': period,
                    'value': value,
                    'display_value': metric_item.get_display_value(period)
                })
                values.append(value)

        # Calculate growth rates
        growth_rates = []
        if len(values) > 1:
            for i in range(1, len(values)):
                if values[i-1] != 0:
                    growth = ((values[i] - values[i-1]) / abs(values[i-1])) *
```

```
100
                    growth_rates.append(growth)

        return {
            'company': company.name,
            'ticker': ticker.upper(),
            'metric': metric,
            'periods_analyzed': len(trend_data),
            'trend_data': trend_data,
            'analytics': {
                'average_growth_rate': sum(growth_rates) / len(growth_rates)
if growth_rates else None,
                'total_growth': ((values[-1] - values[0]) / abs(values[0])) *
100 if len(values) > 1 and values[0] != 0 else None,
                'compound_annual_growth_rate': (((values[-1] / values[0]) **
(1/(len(values)-1))) - 1) * 100 if len(values) > 1 and values[0] > 0 else None
            }
        }

    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
```

Bulk Company Comparison Endpoint

```
@app.post("/financial/compare")
async def compare_companies(
    tickers: List[str],
    periods: int = Query(3, description="Number of periods", ge=1, le=5),
    metrics: List[str] = Query(['revenue', 'net_income'], description="Metrics
to compare")
):
    """Compare financial metrics across multiple companies"""
    try:
        comparison_data = []

        for ticker in tickers:
            try:
                company = Company(ticker.upper())

                if not company.facts:
                    continue

                company_data = {
                    'ticker': ticker.upper(),
                    'name': company.name,
                    'metrics': {}
                }

                # Get statements based on requested metrics
                income_stmt = company.income_statement(periods=periods,
annual=True)
                balance_sheet = company.balance_sheet(perio
annual=True)

                for metric in metrics:
```

```
                    if metric.lower() in ['revenue', 'net_income']:
                        stmt = income_stmt
                    else:
                        stmt = balance_sheet

                    if stmt:
                        metric_item = stmt.find_item(metric)
                        if metric_item:
                            company_data['metrics'][metric] = {
                                'periods': stmt.periods,
                                'values': metric_item.values
                            }

                comparison_data.append(company_data)

            except Exception as e:
                print(f"Error processing {ticker}: {e}")
                continue

        return {
            'comparison': comparison_data,
            'metadata': {
                'companies_compared': len(comparison_data),
                'periods_requested': periods,
                'metrics_requested': metrics
            }
        }

    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
```

Running the API Server

```
# Save the above code as financial_api.py and run:

if __name__ == "__main__":
    import uvicorn

    uvicorn.run(
        "financial_api:app",
        host="0.0.0.0",
        port=8000,
        reload=True,
        title="Financial Data API"
    )

# Or run from command line:
# uvicorn financial_api:app --reload --host 0.0.0.0 --port 8000
```


API Usage Examples

Get 7 Years of Income Statement Data

```
# Get comprehensive income statement data
curl "http://localhost:8000/financial/AAPL/income?
periods=7&annual=true&concise_format=false"

# Get quarterly data for recent periods
curl "http://localhost:8000/financial/AAPL/income?
periods=12&annual=false&concise_format=true"
```

Get Complete Financial Profile

```
# Get all three statements plus ratios
curl "http://localhost:8000/financial/AAPL/comprehensive?
periods=5&include_ratios=true"
```

Analyze Revenue Trends

```
# Get 10-year revenue trend analysis
curl "http://localhost:8000/financial/AAPL/trends?metric=revenue&years=10"
```

Compare Multiple Companies

```
# Compare revenue and profits across companies
curl -X POST "http://localhost:8000/financial/compare" \
     -H "Content-Type: application/json" \
     -d '{"tickers": ["AAPL", "MSFT", "GOOGL"], "periods": 5, "metrics":
["revenue", "net_income"]}'
```

Client Integration Examples

Python Client

```
import requests
import pandas as pd

class FinancialDataClient:
    def __init__(self, base_url="http://localhost:8000"):
        self.base_url = base_url

    def get_income_statement(self, ticker, periods=5, annual=True):
        """Get income statement data"""
        response = requests.get(
            f"{self.base_url}/financial/{ticker}/income",
            params={'periods': periods, 'annual': annual}
        )
```

```
        return response.json() if response.status_code == 200 else None

    def get_comprehensive_data(self, ticker, periods=5):
        """Get all financial statements"""
        response = requests.get(
            f"{self.base_url}/financial/{ticker}/comprehensive",
            params={'periods': periods, 'include_ratios': True}
        )
        return response.json() if response.status_code == 200 else None

    def get_trends(self, ticker, metric, years=5):
        """Get trend analysis"""
        response = requests.get(
            f"{self.base_url}/financial/{ticker}/trends",
            params={'metric': metric, 'years': years}
        )
        return response.json() if response.status_code == 200 else None

# Usage
client = FinancialDataClient()

# Get Apple's financial data
aapl_data = client.get_comprehensive_data('AAPL', periods=7)
print(f"Retrieved data for periods: {aapl_data['periods']}")

# Get revenue trends
revenue_trends = client.get_trends('AAPL', 'revenue', years=10)
print(f"10-year revenue CAGR: {revenue_trends['analytics']
['compound_annual_growth_rate']:.1f}%")
```

JavaScript/Node.js Client

```
class FinancialDataClient {
    constructor(baseUrl = 'http://localhost:8000') {
        this.baseUrl = baseUrl;
    }

    async getIncomeStatement(ticker, periods = 5, annual = true) {
        const response = await fetch(
            `${this.baseUrl}/financial/${ticker}/income?
periods=${periods}&annual=${annual}`
        );
        return response.ok ? response.json() : null;
    }

    async getComprehensiveData(ticker, periods = 5) {
        const response = await fetch(
            `${this.baseUrl}/financial/${ticker}/comprehensive?
periods=${periods}&include_ratios=true`
        );
        return response.ok ? response.json() : null;
    }

    async compareCompanies(tickers, periods = 3, metrics = ['revenue',
'net_income']) {
```

```
        const response = await fetch(`${this.baseUrl}/financial/compare`, {
            method: 'POST',
            headers: {'Content-Type': 'application/json'},
            body: JSON.stringify({tickers, periods, metrics})
        });
        return response.ok ? response.json() : null;
    }
}

// Usage
const client = new FinancialDataClient();

// Get multi-year data
const msftData = await client.getComprehensiveData('MSFT', 7);
console.log(`Retrieved data for periods:`, msftData.periods);

// Compare tech giants
const comparison = await client.compareCompanies(['AAPL', 'MSFT', 'GOOGL',
'AMZN']);
console.log('Comparison data:', comparison);
```

Advanced Use Cases

Building a Financial Dashboard

```
# dashboard.py - Streamlit financial dashboard
import streamlit as st
import requests
import plotly.graph_objects as go
import plotly.express as px
import pandas as pd

st.title("Multi-Year Financial Analysis Dashboard")

# Input controls
ticker = st.text_input("Company Ticker", value="AAPL")
years = st.slider("Years of Data", min_value=2, max_value=10, value=5)

if st.button("Analyze"):
    # Get data from our API
    client = FinancialDataClient()
    data = client.get_comprehensive_data(ticker, periods=years)

    if data:
        st.subheader(f"{data['company_info']['name']} ({ticker})")

        # Revenue trend chart
        income_data = data['income_statement']
        revenue_concept = next((k for k in income_data.keys() if 'revenue' in
k.lower()), None)

        if revenue_concept:
            revenue_item = income_data[revenue_concept]
            periods = data['periods']
```

```
            values = [revenue_item['values'][p]['raw_value'] for p in periods
if p in revenue_item['values']]

            fig = go.Figure()
            fig.add_trace(go.Scatter(x=periods, y=values,
mode='lines+markers', name='Revenue'))
            fig.update_layout(title='Revenue Trend', xaxis_title='Period',
yaxis_title='Revenue ($)')
            st.plotly_chart(fig)

        # Key metrics
        if data['key_metrics']:
            st.subheader("Key Financial Ratios")
            for metric, value in data['key_metrics'].items():
                if isinstance(value, (int, float)):
                    st.metric(metric.replace('_', ' ').title(), f"{value:.2%}"
if 'margin' in metric or 'ro' in metric else f"{value:.2f}")

# Run with: streamlit run dashboard.py
```

Data Export and Analysis

```
def export_financial_data_to_excel(tickers, periods=5,
filename="financial_data.xlsx"):
    """Export multi-company financial data to Excel"""
    import pandas as pd
    from openpyxl import Workbook
    from openpyxl.utils.dataframe import dataframe_to_rows

    client = FinancialDataClient()

    with pd.ExcelWriter(filename, engine='openpyxl') as writer:
        for ticker in tickers:
            data = client.get_comprehensive_data(ticker, periods)

            if data:
                # Income Statement
                income_df =
_convert_api_data_to_dataframe(data['income_statement'], data['periods'])
                income_df.to_excel(writer, sheet_name=f'{ticker}_Income')

                # Balance Sheet
                balance_df =
_convert_api_data_to_dataframe(data['balance_sheet'], data['periods'])
                balance_df.to_excel(writer, sheet_name=f'{ticker}_Balance')

                # Cash Flow
                cashflow_df =
_convert_api_data_to_dataframe(data['cash_flow'], data['periods'])
                cashflow_df.to_excel(writer, sheet_name=f'{ticker}_CashFlow')

    print(f"Financial data exported to {filename}")

def _convert_api_data_to_dataframe(statement_data, periods):
    """Convert API response to pandas DataFrame"""
```

```
    rows = []
    for concept, item_data in statement_data.items():
        row = {'concept': concept, 'label': item_data['label']}
        for period in periods:
            if period in item_data['values']:
                row[period] = item_data['values'][period]['raw_value']
        rows.append(row)

    return pd.DataFrame(rows)

# Export data for analysis
export_financial_data_to_excel(['AAPL', 'MSFT', 'GOOGL', 'AMZN'], periods=7)
```

Best Practices

Error Handling and Resilience

```
import logging
from functools import wraps

def handle_api_errors(f):
    """Decorator for consistent API error handling"""
    @wraps(f)
    async def wrapper(*args, **kwargs):
        try:
            return await f(*args, **kwargs)
        except Exception as e:
            logging.error(f"API error in {f.__name__}: {str(e)}")
            raise HTTPException(status_code=500, detail=f"Internal server
error: {str(e)}")
    return wrapper

@app.get("/financial/{ticker}/income")
@handle_api_errors
async def get_income_statement_safe(ticker: str, periods: int = 4):
    # Implementation with automatic error handling
    pass
```

Caching for Performance

```
from functools import lru_cache
from typing import Optional
import time

class CachedFinancialAPI:
    def __init__(self):
        self._cache = {}
        self._cache_ttl = 3600  # 1 hour

    def _get_cache_key(self, ticker: str, statement_type: str, periods: int,
annual: bool):
        return f"{ticker}_{statement_type}_{periods}_{annual}"
```

```
    def _is_cache_valid(self, timestamp: float) -> bool:
        return time.time() - timestamp < self._cache_ttl

    @lru_cache(maxsize=100)
    def get_cached_company(self, ticker: str):
        """Cache Company objects to avoid repeated API calls"""
        return Company(ticker)

    def get_financial_data(self, ticker: str, statement_type: str, periods:
int, annual: bool):
        cache_key = self._get_cache_key(ticker, statement_type, periods,
annual)

        # Check cache
        if cache_key in self._cache:
            data, timestamp = self._cache[cache_key]
            if self._is_cache_valid(timestamp):
                return data

        # Fetch new data
        company = self.get_cached_company(ticker)

        if statement_type == 'income':
            data = company.income_statement(periods=periods, annual=annual)
        elif statement_type == 'balance':
            data = company.balance_sheet(periods=periods, annual=annual)
        elif statement_type == 'cashflow':
            data = company.cash_flow(periods=periods, annual=annual)

        # Cache the result
        self._cache[cache_key] = (data, time.time())

        return data

# Use cached API in endpoints
cached_api = CachedFinancialAPI()

@app.get("/financial/{ticker}/income")
async def get_cached_income_statement(ticker: str, periods: int = 4, annual:
bool = True):
    data = cached_api.get_financial_data(ticker, 'income', periods, annual)
    # ... rest of endpoint logic
```

Performance Optimization

Batch Processing Multiple Companies

```
import asyncio
import aiohttp
from concurrent.futures import ThreadPoolExecutor

async def process_companies_batch(tickers: List[str], periods: int = 5):
    """Process multiple companies in parallel"""
```

```
    def get_company_data(ticker):
        try:
            company = Company(ticker)
            if company.facts:
                return {
                    'ticker': ticker,
                    'income': company.income_statement(periods=periods,
annual=True),
                    'balance': company.balance_sheet(periods=periods,
annual=True),
                    'cashflow': company.cash_flow(periods=periods,
annual=True)
                }
        except Exception as e:
            print(f"Error processing {ticker}: {e}")
        return None

    # Use ThreadPoolExecutor for I/O bound operations
    with ThreadPoolExecutor(max_workers=10) as executor:
        loop = asyncio.get_event_loop()
        tasks = [
            loop.run_in_executor(executor, get_company_data, ticker)
            for ticker in tickers
        ]
        results = await asyncio.gather(*tasks)

    return [r for r in results if r is not None]

@app.post("/financial/batch")
async def process_batch(tickers: List[str], periods: int = Query(5, le=10)):
    """Process multiple companies in parallel"""
    results = await process_companies_batch(tickers, periods)
    return {
        'processed': len(results),
        'requested': len(tickers),
        'data': results
    }
```

Deployment Considerations

Production Setup

```
# production_config.py
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from fastapi.middleware.gzip import GZipMiddleware
import logging

def create_production_app():
    app = FastAPI(
        title="Financial Data API",
        description="Production financial data service",
        version="1.0.0",
```

```
        docs_url="/docs" if settings.DEBUG else None
    )

    # Add middleware
    app.add_middleware(GZipMiddleware, minimum_size=1000)
    app.add_middleware(
        CORSMiddleware,
        allow_origins=["https://yourdomain.com"],
        allow_credentials=True,
        allow_methods=["GET", "POST"],
        allow_headers=["*"],
    )

    # Configure logging
    logging.basicConfig(level=logging.INFO)

    return app

app = create_production_app()

# Add rate limiting
from slowapi import Limiter, _rate_limit_exceeded_handler
from slowapi.util import get_remote_address
from slowapi.errors import RateLimitExceeded

limiter = Limiter(key_func=get_remote_address)
app.state.limiter = limiter
app.add_exception_handler(RateLimitExceeded, _rate_limit_exceeded_handler)

@app.get("/financial/{ticker}/income")
@limiter.limit("10/minute")
async def rate_limited_income_statement(request: Request, ticker: str):
    # Rate-limited endpoint implementation
    pass
```

Testing

Unit Tests for API Endpoints

```
# test_financial_api.py
import pytest
from fastapi.testclient import TestClient
from financial_api import app

client = TestClient(app)

def test_get_income_statement():
    """Test income statement endpoint"""
    response = client.get("/financial/AAPL/income?periods=3&annual=true")
    assert response.status_code == 200

    data = response.json()
    assert data['company_info']['ticker'] == 'AAPL'
    assert len(data['periods']) <= 3
```

```
    assert 'income_statement' in data or 'data' in data

def test_comprehensive_endpoint():
    """Test comprehensive financial data endpoint"""
    response = client.get("/financial/MSFT/comprehensive?
periods=2&include_ratios=true")
    assert response.status_code == 200

    data = response.json()
    assert 'income_statement' in data
    assert 'balance_sheet' in data
    assert 'cash_flow' in data
    assert 'key_metrics' in data

def test_trends_analysis():
    """Test trends endpoint"""
    response = client.get("/financial/GOOGL/trends?metric=revenue&years=5")
    assert response.status_code == 200

    data = response.json()
    assert 'trend_data' in data
    assert 'analytics' in data
    assert data['metric'] == 'revenue'

def test_company_comparison():
    """Test company comparison endpoint"""
    payload = {
        "tickers": ["AAPL", "MSFT"],
        "periods": 2,
        "metrics": ["revenue"]
    }
    response = client.post("/financial/compare", json=payload)
    assert response.status_code == 200

    data = response.json()
    assert len(data['comparison']) <= 2

def test_invalid_ticker():
    """Test handling of invalid ticker"""
    response = client.get("/financial/INVALIDTICKER/income")
    assert response.status_code == 404

# Run with: pytest test_financial_api.py -v
```

This comprehensive guide provides everything needed to build a production-ready FastAPI service for accessing multi-year financial statement data using EdgarTools. The implementation includes error handling, caching, rate limiting, and extensive examples for building financial analysis applications.]