Stock Data Web Scraper - Complete Project Documentation

# Stock Data Web Scraper - Complete Project Documentation

## Table of Contents

1. [Project Overview](#project-overview)
2. [System Architecture](#system-architecture)
3. [Technology Stack](#technology-stack)
4. [Project Structure](#project-structure)
5. [Core Components](#core-components)
6. [API Endpoints](#api-endpoints)
7. [Data Flow](#data-flow)
8. [Deployment Architecture](#deployment-architecture)
9. [Usage Examples](#usage-examples)
10. [Performance & Scaling](#performance--scaling)

* --

## Project Overview

* Stock Data Web Scraper\* is a sophisticated Django-based web application that automatically scrapes financial data from Morningstar.com. The application provides both web interface and REST API access to comprehensive stock financial information including income statements, balance sheets, cash flow statements, dividends, key metrics, and valuation data.

### Key Features

* 🤖 Automated Web Scraping: Real-time data extraction from Morningstar.com
* 📊 Multiple Data Types: Income statements, balance sheets, cash flow, dividends, key metrics, valuation
* 🔄 Asynchronous Processing: Celery-powered background task execution
* 📱 REST API: JSON endpoints for programmatic access
* 🌐 Web Interface: User-friendly HTML interface for manual data retrieval
* 📈 Export Options: Excel file downloads with comprehensive data formatting
* 🔓 Open Access: Public API endpoints without authentication requirements
* 🚀 Cloud Deployment: Optimized for Fly.io container platform
* --

## System Architecture

The application follows a microservices-like architecture with clear separation of concerns:

### System Architecture Diagram

graph TB  
 subgraph "Client Layer"  
 WEB[Web Browser]  
 API[API Clients]  
 CURL[cURL/Scripts]  
 end  
   
 subgraph "Load Balancer"  
 FLY[Fly.io Load Balancer]  
 end  
   
 subgraph "Application Layer"  
 DJANGO[Django Web Framework]  
 VIEWS[View Controllers]  
 URLS[URL Routing]  
 end  
   
 subgraph "Task Processing"  
 CELERY[Celery Workers]  
 REDIS[Redis Message Broker]  
 SELENIUM[Selenium Chrome Driver]  
 end  
   
 subgraph "Data Storage"  
 FIREBASE[Firebase Realtime DB]  
 SQLITE[SQLite Local DB]  
 EXCEL[Excel File Storage]  
 end  
   
 subgraph "External Services"  
 MORNINGSTAR[Morningstar.com]  
 IPAPI[IP Geolocation API]  
 end  
   
 WEB --> FLY  
 API --> FLY  
 CURL --> FLY  
 FLY --> DJANGO  
 DJANGO --> VIEWS  
 DJANGO --> URLS  
 VIEWS --> CELERY  
 CELERY --> REDIS  
 CELERY --> SELENIUM  
 SELENIUM --> MORNINGSTAR  
 CELERY --> FIREBASE  
 DJANGO --> SQLITE  
 DJANGO --> EXCEL  
 VIEWS --> IPAPI

* --

## Technology Stack

### Backend Framework

* Django 4.0.6: Web framework with MVC architecture
* Python 3.11: Core programming language
* Gunicorn: WSGI HTTP Server for production

### Task Queue & Processing

* Celery 5.2.7: Distributed task queue for asynchronous processing
* Redis: Message broker and result backend
* celery-progress: Real-time task progress tracking

### Web Scraping

* Selenium 4.9.0: Web browser automation
* undetected-chromedriver 3.5.4: Stealth Chrome driver to avoid detection
* BeautifulSoup4: HTML parsing and data extraction
* ChromeDriver: Automated Chrome browser control

### Data Processing & Storage

* Pandas: Data manipulation and analysis
* Firebase Realtime Database: Cloud-based JSON data storage
* SQLite: Local relational database for user management
* openpyxl & XlsxWriter: Excel file generation

### Frontend & UI

* Bootstrap 4: Responsive CSS framework
* django-crispy-forms: Enhanced form rendering
* Whitenoise: Static file serving
* HTML5/CSS3/JavaScript: Frontend technologies

### Deployment & Infrastructure

* Fly.io: Container hosting platform
* Docker: Containerization
* GitHub: Version control and CI/CD
* --

## Project Structure

StockDataWebAppV2/  
├── 📁 core/ # Main Django application  
│ ├── 📄 scraperVersionTwo.py # Primary scraping logic & API endpoints  
│ ├── 📄 tasks.py # Celery background tasks  
│ ├── 📄 views.py # Web interface controllers  
│ ├── 📄 urls.py # URL routing configuration  
│ ├── 📄 models.py # Database models  
│ ├── 📁 templates/ # HTML templates  
│ │ ├── 📄 stockData.html # Main scraping interface  
│ │ ├── 📄 loadScreen.html # Progress tracking page  
│ │ └── 📄 loadScreenAll.html # Comprehensive scraping progress  
│ └── 📄 admin.py # Django admin configuration  
│  
├── 📁 register/ # User management application  
│ ├── 📄 models.py # User profile models  
│ ├── 📄 views.py # Authentication views  
│ ├── 📄 forms.py # User registration forms  
│ └── 📄 urls.py # Registration URL routing  
│  
├── 📁 stock\_scraper/ # Django project configuration  
│ ├── 📄 settings.py # Application settings  
│ ├── 📄 urls.py # Root URL configuration  
│ ├── 📄 celery.py # Celery configuration  
│ ├── 📄 wsgi.py # WSGI application  
│ └── 📄 asgi.py # ASGI application  
│  
├── 📄 Dockerfile # Container configuration  
├── 📄 fly.toml # Fly.io deployment config  
├── 📄 requirements.txt # Python dependencies  
├── 📄 manage.py # Django management script  
├── 📄 scraper\_config.py # Scraping configuration  
└── 📄 db.sqlite3 # Local SQLite database

* --

## Core Components

### 1. Scraping Engine (`core/tasks.py`)

The heart of the application, containing Celery tasks for data extraction:

### Main Scraper Tasks:

* scraper(): Primary task for financial statements (Income Statement, Balance Sheet, Cash Flow)
* scraper\_dividends(): Dedicated dividend data extraction
* scraper\_valuation(): Valuation metrics and ratios
* all\_scraper(): Comprehensive data collection (8 data types)
* scraper\_operating\_performance(): Operating efficiency metrics

### Stealth Configuration:

def create\_stealth\_driver():  
 """Creates undetected Chrome driver with anti-bot measures"""  
 options = uc.ChromeOptions()  
 options.add\_argument("--disable-blink-features=AutomationControlled")  
 options.add\_argument("--no-sandbox")  
 options.add\_argument("--disable-dev-shm-usage")  
 # Production optimizations for containers  
 if IS\_PRODUCTION:  
 options.add\_argument("--headless=new")  
 options.add\_argument("--memory-pressure-off")  
 return uc.Chrome(options=options)

### 2. API Layer (`core/scraperVersionTwo.py`)

### Primary Functions:

* scrape(): Main web interface controller
* download(): Excel file generation and download
* financial\_statements\_json(): Open access JSON API
* get\_task\_info(): Real-time task status tracking

### Open Access API Example:

def financial\_statements\_json(request):  
 """Open access financial statements API"""  
 if 'ticker' in request.GET and 'market' in request.GET and 'type' in request.GET:  
 # Type mapping: is=Income Statement, bs=Balance Sheet, cf=Cash Flow  
 type\_mapping = {'is': 'INCOME\_STATEMENT', 'bs': 'BALANCE\_SHEET', 'cf': 'CASH\_FLOW'}  
   
 # Start async scraping task  
 task = scraper.delay(ticker\_value, market\_value, download\_type)  
   
 # Wait for completion with timeout  
 while not task.ready() and wait\_time < 120:  
 time.sleep(5)  
   
 # Return JSON data from Firebase  
 return HttpResponse(pretty\_json, content\_type='text/json')

### 3. Data Models (`core/models.py` & `register/models.py`)

### API Request Tracking:

class APIRequest(models.Model):  
 title = models.CharField(max\_length=200) # Request type  
 endpoint = models.TextField() # API endpoint used  
 ticker = models.TextField() # Stock ticker  
 market = models.TextField() # Market identifier  
 location = models.TextField() # User location data  
 user\_email = models.TextField() # User email (if authenticated)  
 created = models.DateTimeField(default=timezone.now())

### User Profile Management:

class Profile(models.Model):  
 user = models.OneToOneField(User, on\_delete=models.CASCADE)  
 country = models.CharField(max\_length=100)  
 user\_key = models.CharField(max\_length=16) # API authentication key

### 4. Web Interface (`core/templates/`)

### Main Scraping Interface (`stockData.html`):

* Dropdown Selection: Choose from 8 data types
* Input Fields: Ticker symbol and market selection
* Progress Tracking: Real-time scraping status
* Download Options: Excel file generation

### Progress Tracking (`loadScreen.html`):

* Live Updates: WebSocket-based progress monitoring
* Error Handling: Detailed error messages and retry options
* Multi-step Progress: Visual progress bar for complex operations
* --

## API Endpoints

### Public Endpoints (Open Access)

### 1. Financial Statements JSON API

GET /financial-statements-json

* Parameters:\*
* ticker (required): Stock ticker symbol (e.g., "AAPL")
* market (required): Market code (e.g., "XNAS")
* type (required): Statement type (is, bs, cf)
* Response:\*

{  
 "TTM": "2023",  
 "2022-12": "2022",  
 "2021-12": "2021",  
 "Revenue": [394328000000, 365817000000, 294135000000],  
 "Net Income": [99803000000, 94680000000, 57411000000]  
}

### 2. Task Status Tracking

GET /get\_task\_info/?task\_id={TASK\_ID}

* Response:\*

{  
 "task\_id": "abc123",  
 "task\_status": "SUCCESS",  
 "task\_result": "COMPLETED",  
 "progress": {  
 "current": 8,  
 "total": 8,  
 "description": "Processing Valuation"  
 }  
}

### Web Interface Endpoints

### 1. Main Scraping Interface

GET /  
POST / (with scraping parameters)

### 2. Excel Download

POST /stockDataDownload

* --

## Data Flow

### Web Scraping Workflow

sequenceDiagram  
 participant U as User/API Client  
 participant D as Django View  
 participant C as Celery Worker  
 participant S as Selenium Driver  
 participant M as Morningstar.com  
 participant F as Firebase DB  
 participant E as Excel Export  
  
 U->>D: Request financial data  
 D->>C: Create async scraping task  
 C->>S: Launch Chrome browser  
 S->>M: Navigate to financial page  
 S->>M: Click financial statement tab  
 S->>M: Click "Expand Detail View"  
 S->>M: Click "Export Data"  
 M->>S: Download Excel file  
 S->>C: Process downloaded file  
 C->>F: Store JSON data  
 C->>D: Task completion signal  
 D->>E: Generate formatted Excel  
 D->>U: Return data/download link

### Data Processing Pipeline

graph LR  
 subgraph "Data Extraction"  
 WEB[Web Scraping] --> XLS[Excel Download]  
 XLS --> PARSE[Pandas Processing]  
 end  
   
 subgraph "Data Transformation"  
 PARSE --> CLEAN[Data Cleaning]  
 CLEAN --> JSON[JSON Conversion]  
 JSON --> VALIDATE[Validation]  
 end  
   
 subgraph "Data Storage"  
 VALIDATE --> FIREBASE[Firebase Storage]  
 VALIDATE --> LOCAL[Local Excel Files]  
 end  
   
 subgraph "Data Delivery"  
 FIREBASE --> API[JSON API Response]  
 LOCAL --> DOWNLOAD[Excel Download]  
 API --> CLIENT[API Clients]  
 DOWNLOAD --> USER[Web Users]  
 end

### Available Data Types

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Type** | **Description** | **API Code** | **Excel Sheet** |
| \*\*Income Statement\*\* | Revenue, expenses, profit/loss | `is` | Income Statement |
| \*\*Balance Sheet\*\* | Assets, liabilities, equity | `bs` | Balance Sheet |
| \*\*Cash Flow\*\* | Operating, investing, financing cash flows | `cf` | Cash Flow |
| \*\*Dividends\*\* | Dividend history and yields | N/A | Dividends |
| \*\*Key Metrics - Cash Flow\*\* | FCF, FCF yield, working capital | N/A | Key Metrics Cash Flow |
| \*\*Key Metrics - Growth\*\* | Revenue growth, earnings growth | N/A | Key Metrics Growth |
| \*\*Key Metrics - Financial Health\*\* | Debt ratios, liquidity ratios | N/A | Key Metrics Financial Health |
| \*\*Valuation\*\* | P/E, P/B, EV/EBITDA ratios | N/A | Valuation |

* --

## Deployment Architecture

### Fly.io Container Configuration

graph TB  
 subgraph "Internet"  
 USERS[Users Worldwide]  
 end  
   
 subgraph "Fly.io Edge Network"  
 LB[Global Load Balancer]  
 CDN[Edge Caching]  
 end  
   
 subgraph "Fly.io Data Center"  
 subgraph "Web Machines"  
 WEB1[Web Instance 1<br/>Django + Gunicorn]  
 WEB2[Web Instance 2<br/>Django + Gunicorn]  
 end  
   
 subgraph "Worker Machines"  
 WORKER1[Celery Worker 1<br/>Chrome + Selenium]  
 end  
   
 subgraph "Storage"  
 VOLUME[Persistent Volume<br/>Excel Files]  
 end  
 end  
   
 subgraph "External Services"  
 REDIS[Redis Labs<br/>Message Broker]  
 FIREBASE[Firebase<br/>JSON Database]  
 MORNINGSTAR[Morningstar.com<br/>Data Source]  
 end  
   
 USERS --> LB  
 LB --> CDN  
 CDN --> WEB1  
 CDN --> WEB2  
 WEB1 --> WORKER1  
 WEB2 --> WORKER1  
 WORKER1 --> VOLUME  
 WORKER1 --> REDIS  
 WORKER1 --> FIREBASE  
 WORKER1 --> MORNINGSTAR

### Container Specifications

### Web Machines (2 instances):

* CPU: 1 shared vCPU
* Memory: 512MB RAM
* Network: Global anycast IPs
* Purpose: Handle HTTP requests, serve web interface

### Worker Machines (1 instance):

* CPU: 1 dedicated vCPU
* Memory: 1GB RAM (Chrome optimization)
* Storage: 10GB persistent volume
* Purpose: Execute Celery tasks, run Chrome browser
* --

## Usage Examples

### 1. Web Interface Usage

### Step-by-Step Process:

1. Navigate to https://stockdata-scraper.fly.dev/
2. Select data type from dropdown (Income Statement, Balance Sheet, etc.)
3. Enter ticker symbol (e.g., "AAPL")
4. Choose market (e.g., "XNAS")
5. Click "Scrape" to start extraction
6. Monitor progress on loading screen
7. Download Excel file when complete

### 2. API Usage Examples

### Income Statement API Call:

curl "https://stockdata-scraper.fly.dev/financial-statements-json?ticker=AAPL&market=XNAS&type=is"

* Response:\*

{  
 "TTM": "2023",  
 "2022-12": "2022",   
 "2021-12": "2021",  
 "Revenue": [394328000000, 365817000000, 294135000000],  
 "Cost of Revenue": [223546000000, 212981000000, 169559000000],  
 "Gross Profit": [170782000000, 152836000000, 124576000000],  
 "Operating Expense": [70851000000, 51344000000, 43887000000],  
 "Operating Income": [99931000000, 101492000000, 80689000000],  
 "Net Income": [99803000000, 94680000000, 57411000000]  
}

### Balance Sheet API Call:

curl "https://stockdata-scraper.fly.dev/financial-statements-json?ticker=TSLA&market=XNAS&type=bs"

### Cash Flow API Call:

curl "https://stockdata-scraper.fly.dev/financial-statements-json?ticker=MSFT&market=XNAS&type=cf"

### 3. Python Integration Example

import requests  
import json  
import time  
  
class StockDataAPI:  
 BASE\_URL = "https://stockdata-scraper.fly.dev"  
   
 def get\_financial\_statement(self, ticker, market, statement\_type):  
 """  
 Fetch financial statement data  
 statement\_type: 'is' (income), 'bs' (balance), 'cf' (cash flow)  
 """  
 url = f"{self.BASE\_URL}/financial-statements-json"  
 params = {  
 'ticker': ticker,  
 'market': market,  
 'type': statement\_type  
 }  
   
 response = requests.get(url, params=params, timeout=180)  
   
 if response.status\_code == 200:  
 return response.json()  
 else:  
 raise Exception(f"API Error: {response.text}")  
   
 def get\_all\_statements(self, ticker, market):  
 """Get all three financial statements"""  
 statements = {}  
   
 for stmt\_type, name in [('is', 'income'), ('bs', 'balance'), ('cf', 'cashflow')]:  
 print(f"Fetching {name} statement...")  
 statements[name] = self.get\_financial\_statement(ticker, market, stmt\_type)  
 time.sleep(30) # Rate limiting  
   
 return statements  
  
# Usage example  
api = StockDataAPI()  
  
# Get Apple's income statement  
apple\_income = api.get\_financial\_statement('AAPL', 'XNAS', 'is')  
print(f"Apple Revenue: ${apple\_income['Revenue'][0]:,}")  
  
# Get all statements for Tesla  
tesla\_data = api.get\_all\_statements('TSLA', 'XNAS')

* --

## Performance & Scaling

### Current Performance Metrics

### Response Times:

* API Response: 90-120 seconds (includes scraping time)
* Web Interface: 2-5 minutes (comprehensive data)
* Excel Generation: 5-15 seconds

### Throughput:

* Concurrent Requests: 3-5 simultaneous scraping tasks
* Daily Capacity: 500-1000 successful scraping operations
* Error Rate: <5% (mostly due to website changes)

### Scaling Considerations

### Performance Optimizations:

1. Caching Layer:

# Redis caching for frequently requested data  
 cache\_key = f"financial\_data:{ticker}:{market}:{statement\_type}"  
 cached\_data = redis.get(cache\_key)  
 if cached\_data:  
 return json.loads(cached\_data)

1. Database Optimization:

# Implement data expiration and cleanup  
 def cleanup\_old\_data():  
 cutoff\_date = datetime.now() - timedelta(hours=24)  
 database.child('expired\_data').remove()

### Monitoring & Analytics

### Key Metrics to Track:

* Scraping Success Rate: Percentage of successful data extractions
* Response Time Distribution: P50, P95, P99 response times
* Error Categorization: Network, parsing, timeout errors
* Geographic Usage: Request origins and patterns
* Resource Utilization: CPU, memory, disk usage
* --

## Future Enhancements

### Planned Features

1. Rate Limiting: Per-IP request throttling
2. Authentication Tiers: Free vs premium access levels
3. Webhook Notifications: Async completion callbacks
4. Data Validation: Enhanced error detection and data quality checks
5. Historical Data: Multi-year data retention and trending
6. Real-time Updates: WebSocket-based live data streaming

### Technical Improvements

1. Microservices Architecture: Separate scraping, API, and web services
2. Container Orchestration: Kubernetes deployment
3. Advanced Caching: Multi-layer caching strategy
4. Machine Learning: Predictive scraping and data quality assessment
5. API Rate Management: Sophisticated usage analytics and controls

* --

This comprehensive documentation provides a complete overview of the Stock Data Web Scraper project, covering all major components, architecture decisions, and usage patterns. The system is designed for reliability, scalability, and ease of use while maintaining robust data extraction capabilities.