Stock Data Web Scraper - How It Works

# Stock Data Web Scraper - How It Works

## Table of Contents

1. [Application Overview](#application-overview)
2. [How the Web Scraping Works](#how-the-web-scraping-works)
3. [User Journey and Workflows](#user-journey-and-workflows)
4. [API Processing Flow](#api-processing-flow)
5. [Data Extraction Process](#data-extraction-process)
6. [Background Task Processing](#background-task-processing)
7. [Data Storage and Retrieval](#data-storage-and-retrieval)
8. [Error Handling and Recovery](#error-handling-and-recovery)
9. [Performance Optimization](#performance-optimization)
10. [Security and Stealth Features](#security-and-stealth-features)

* --

## Application Overview

The Stock Data Web Scraper is an intelligent automation system that extracts financial data from Morningstar.com in real-time. The application serves as a bridge between users seeking financial data and the complex web-based financial information systems.

### What the Application Does

* Primary Function\*: Automatically navigates to Morningstar.com, locates specific stock information, extracts financial statements, and delivers the data in user-friendly formats (JSON API responses and Excel files).
* Key Capabilities\*:
* 🤖 Automated Browser Control: Uses Chrome browser automation to interact with websites like a human user
* 📊 Multi-Format Data Extraction: Supports 8 different types of financial data
* 🔄 Asynchronous Processing: Handles multiple requests simultaneously without blocking
* 📱 Dual Interface: Both web interface for manual use and API for programmatic access
* 🛡️ Anti-Detection Technology: Advanced stealth features to avoid bot detection
* 🌐 Global Accessibility: Open access API available worldwide
* --

## How the Web Scraping Works

### Step-by-Step Scraping Process

### 1. Browser Initialization

┌─────────────────────────────────────────────────────────────────┐  
│ BROWSER STARTUP SEQUENCE │  
├─────────────────────────────────────────────────────────────────┤  
│ 1. Launch Chrome with stealth configuration │  
│ 2. Set user agent to mimic real browser │  
│ 3. Configure download directory for files │  
│ 4. Apply anti-detection measures │  
│ 5. Set window size and display options │  
└─────────────────────────────────────────────────────────────────┘

* Technical Details\*:
* Uses undetected-chromedriver to avoid bot detection
* Randomizes user agents from a pool of realistic options
* Configures Chrome flags for container optimization
* Sets up download preferences for Excel file handling

### 2. Website Navigation

Navigation Flow:  
User Input (AAPL, XNAS) → Build Morningstar URL → Navigate to Page  
 ↓  
Check Page Load Success → Wait for Elements → Verify Stock Found  
 ↓  
Click Financial Tab → Wait for Content → Navigate to Statements

* URL Construction Example\*:

Base URL: https://www.morningstar.com/stocks/  
Market Code: XNAS (NASDAQ)  
Ticker: AAPL  
Final URL: https://www.morningstar.com/stocks/xnas/aapl/quote

### 3. Data Extraction Workflow

graph TB  
 A[Navigate to Stock Page] --> B[Click Financials Tab]  
 B --> C[Select Statement Type]  
 C --> D[Click 'Expand Detail View']  
 D --> E[Wait for Data Load]  
 E --> F[Click 'Export Data']  
 F --> G[Wait for Excel Download]  
 G --> H[Verify File Downloaded]  
 H --> I[Process Excel Data]  
 I --> J[Convert to JSON]

* Interactive Elements\*:
* Financials Tab: Located in the main navigation menu
* Statement Selector: Dropdown for Income Statement, Balance Sheet, Cash Flow
* Expand Detail View: Button to show complete data
* Export Data: Downloads Excel file with raw financial data

### 4. Data Processing Pipeline

Excel File → Pandas DataFrame → Data Cleaning → JSON Conversion  
 ↓ ↓ ↓ ↓  
File Validation → Column Mapping → Remove NaN → Format Output  
 ↓ ↓ ↓ ↓  
Size Check → Standardize Names → Fill Blanks → API Response

* --

## User Journey and Workflows

### Web Interface User Journey

### Step 1: User Accesses the Application

┌────────────────────────────────────────────────────────────────┐  
│ WEB INTERFACE │  
├────────────────────────────────────────────────────────────────┤  
│ Data Type: [Income Statement ▼] │  
│ Ticker: [AAPL ] │  
│ Market: [XNAS ] │  
│ │  
│ [ Scrape Data ] [ Download All ] │  
└────────────────────────────────────────────────────────────────┘

* User Actions\*:

1. Select Data Type: Choose from 8 available financial data types
2. Enter Ticker: Input stock symbol (e.g., AAPL, TSLA, MSFT)
3. Specify Market: Enter market code (e.g., XNAS, NYSE)
4. Initiate Scraping: Click "Scrape Data" button

### Step 2: Processing Screen

┌────────────────────────────────────────────────────────────────┐  
│ PROGRESS TRACKING │  
├────────────────────────────────────────────────────────────────┤  
│ Status: Scraping Financial Data... │  
│ │  
│ Progress: ████████████░░░░░░░░ 60% │  
│ │  
│ Current Step: Downloading Excel file from Morningstar │  
│ Elapsed Time: 1 minute 23 seconds │  
│ │  
│ [ Cancel Process ] │  
└────────────────────────────────────────────────────────────────┘

* Real-time Updates\*:
* Progress bar showing completion percentage
* Current step description
* Elapsed time counter
* Option to cancel long-running processes

### Step 3: Results and Download

┌────────────────────────────────────────────────────────────────┐  
│ RESULTS READY │  
├────────────────────────────────────────────────────────────────┤  
│ ✅ Data extraction completed successfully! │  
│ │  
│ Company: Apple Inc. (AAPL) │  
│ Market: NASDAQ (XNAS) │  
│ Data Type: Income Statement │  
│ Records: 45 financial metrics over 5 years │  
│ │  
│ [ Download Excel ] [ View JSON ] [ Start New Scrape ] │  
└────────────────────────────────────────────────────────────────┘

### API User Journey

### Direct API Access Flow

API Request → Parameter Validation → Task Creation → Queue Processing  
 ↓ ↓ ↓ ↓  
HTTP GET → Check Required Fields → Celery Task → Worker Assignment  
 ↓ ↓ ↓ ↓  
JSON Response ← Data Formatting ← Scraping Complete ← Browser Automation

* API Request Example\*:

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

* Processing Steps\*:

1. Request Validation: Check for required parameters (ticker, market, type)
2. Task Queue: Add scraping job to Celery background queue
3. Worker Processing: Dedicated worker performs browser automation
4. Response Delivery: Return JSON data or error message

* --

## API Processing Flow

### Request Processing Architecture

sequenceDiagram  
 participant Client as API Client  
 participant Django as Django Server  
 participant Celery as Celery Worker  
 participant Chrome as Chrome Browser  
 participant Morningstar as Morningstar.com  
 participant Firebase as Firebase DB  
  
 Client->>Django: GET /financial-statements-json  
 Django->>Django: Validate Parameters  
 Django->>Celery: Create Scraping Task  
 Celery->>Chrome: Launch Browser  
 Chrome->>Morningstar: Navigate to Stock Page  
 Morningstar->>Chrome: Return Financial Page  
 Chrome->>Morningstar: Click Export Button  
 Morningstar->>Chrome: Download Excel File  
 Chrome->>Celery: Process Downloaded Data  
 Celery->>Firebase: Store JSON Data  
 Celery->>Django: Task Complete Signal  
 Django->>Client: Return JSON Response

### Parameter Processing

* Required Parameters\*:
* ticker: Stock symbol (validated against common patterns)
* market: Market identifier (e.g., XNAS, NYSE, LSE)
* type: Data type (is, bs, cf for Income Statement, Balance Sheet, Cash Flow)
* Parameter Validation Logic\*:

def validate\_parameters(request):  
 errors = []  
   
 # Check ticker format  
 if not re.match(r'^[A-Z]{1,10}$', ticker.upper()):  
 errors.append("Invalid ticker format")  
   
 # Validate market code  
 valid\_markets = ['XNAS', 'NYSE', 'LSE', 'TSE', 'ASX']  
 if market not in valid\_markets:  
 errors.append("Unsupported market code")  
   
 # Check data type  
 if type not in ['is', 'bs', 'cf']:  
 errors.append("Invalid data type")  
   
 return errors

### Response Format Standardization

* Success Response Structure\*:

{  
 "status": "success",  
 "company": "Apple Inc.",  
 "ticker": "AAPL",  
 "market": "XNAS",  
 "data\_type": "Income Statement",  
 "periods": ["TTM", "2022-12", "2021-12", "2020-12", "2019-12"],  
 "data": {  
 "Revenue": [394328000000, 365817000000, 294135000000, 260174000000, 265595000000],  
 "Gross Profit": [170782000000, 152836000000, 152836000000, 104956000000, 98392000000],  
 "Net Income": [99803000000, 94680000000, 57411000000, 55256000000, 48351000000]  
 },  
 "currency": "USD",  
 "last\_updated": "2024-01-15T10:30:00Z"  
}

* Error Response Structure\*:

{  
 "status": "error",  
 "error\_code": "INVALID\_TICKER",  
 "message": "Ticker symbol not found or invalid",  
 "details": "The ticker 'XYZ123' could not be found on Morningstar",  
 "timestamp": "2024-01-15T10:30:00Z"  
}

* --

## Data Extraction Process

### Selenium Automation Workflow

### Browser Control Sequence

1. Initialize Chrome Driver  
 ├── Set stealth options  
 ├── Configure user agent  
 ├── Set download directory  
 └── Apply container optimizations  
  
2. Navigate to Target URL  
 ├── Construct Morningstar URL  
 ├── Handle redirects  
 ├── Wait for page load  
 └── Verify stock exists  
  
3. Interact with Page Elements  
 ├── Click Financials tab  
 ├── Select data type  
 ├── Click expand view  
 └── Trigger export function  
  
4. Handle File Download  
 ├── Wait for download start  
 ├── Monitor file creation  
 ├── Verify file completion  
 └── Process downloaded data

### Element Selection Strategy

* Robust Element Identification\*:

# Multiple selector strategies for reliability  
SELECTORS = {  
 'financials\_tab': [  
 "//a[contains(text(), 'Financials')]",  
 "//button[@data-tab='financials']",  
 ".tab-financials",  
 "#financials-tab"  
 ],  
   
 'export\_button': [  
 "//button[contains(text(), 'Export')]",  
 "//a[contains(@href, 'export')]",  
 ".export-data-btn",  
 "[data-action='export']"  
 ]  
}  
  
def find\_element\_with\_fallback(driver, selectors):  
 for selector in selectors:  
 try:  
 if selector.startswith('//'):  
 element = driver.find\_element(By.XPATH, selector)  
 else:  
 element = driver.find\_element(By.CSS\_SELECTOR, selector)  
 return element  
 except NoSuchElementException:  
 continue  
 raise Exception("Element not found with any selector")

### Data Type Mapping

* Financial Statement Types\*:

DATA\_TYPE\_MAPPING = {  
 'is': {  
 'full\_name': 'Income Statement',  
 'tab\_selector': 'income-statement',  
 'export\_format': 'annual',  
 'expected\_columns': ['Revenue', 'Cost of Revenue', 'Gross Profit', 'Operating Income', 'Net Income']  
 },  
   
 'bs': {  
 'full\_name': 'Balance Sheet',  
 'tab\_selector': 'balance-sheet',  
 'export\_format': 'annual',  
 'expected\_columns': ['Total Assets', 'Total Liabilities', 'Total Equity', 'Cash and Equivalents']  
 },  
   
 'cf': {  
 'full\_name': 'Cash Flow Statement',  
 'tab\_selector': 'cash-flow',  
 'export\_format': 'annual',  
 'expected\_columns': ['Operating Cash Flow', 'Investing Cash Flow', 'Financing Cash Flow']  
 }  
}

* --

## Background Task Processing

### Celery Task Management

### Task Queue Architecture

┌─────────────────────────────────────────────────────────────────┐  
│ TASK QUEUE SYSTEM │  
├─────────────────────────────────────────────────────────────────┤  
│ │  
│ Web Requests → Django Views → Celery Tasks → Redis Queue │  
│ ↓ │  
│ Task Distribution │  
│ ↓ │  
│ Worker 1: Income Statement Worker 2: Balance Sheet │  
│ Worker 3: Cash Flow Worker 4: Dividends │  
│ ↓ │  
│ Results Collection │  
│ ↓ │  
│ Firebase Storage ← JSON Formatting ← Data Processing │  
│ │  
└─────────────────────────────────────────────────────────────────┘

### Task Lifecycle Management

* Task States\*:

TASK\_STATES = {  
 'PENDING': 'Task is waiting to be processed',  
 'STARTED': 'Task execution has begun',  
 'PROGRESS': 'Task is making progress (with percentage)',  
 'SUCCESS': 'Task completed successfully',  
 'FAILURE': 'Task failed with error',  
 'RETRY': 'Task is being retried after failure',  
 'REVOKED': 'Task was cancelled by user'  
}

* Progress Tracking Implementation\*:

@shared\_task(bind=True)  
def scraper(self, ticker\_value, market\_value, download\_type):  
 progress\_recorder = ProgressRecorder(self)  
   
 # Step 1: Initialize browser  
 progress\_recorder.set\_progress(10, 100, 'Initializing browser...')  
 driver = create\_stealth\_driver()  
   
 # Step 2: Navigate to website  
 progress\_recorder.set\_progress(25, 100, 'Navigating to Morningstar...')  
 driver.get(construct\_url(ticker\_value, market\_value))  
   
 # Step 3: Find and click elements  
 progress\_recorder.set\_progress(50, 100, 'Locating financial data...')  
 click\_financials\_tab(driver)  
   
 # Step 4: Download data  
 progress\_recorder.set\_progress(75, 100, 'Downloading Excel file...')  
 trigger\_export(driver)  
   
 # Step 5: Process results  
 progress\_recorder.set\_progress(90, 100, 'Processing data...')  
 data = process\_downloaded\_file()  
   
 # Step 6: Complete  
 progress\_recorder.set\_progress(100, 100, 'Complete!')  
 return data

### Timeout and Error Management

* Task Timeout Configuration\*:

CELERY\_TASK\_TIME\_LIMIT = 300 # 5 minutes hard limit  
CELERY\_TASK\_SOFT\_TIME\_LIMIT = 240 # 4 minutes soft limit  
  
@shared\_task(bind=True, time\_limit=300, soft\_time\_limit=240)  
def scraper\_with\_timeout(self, ticker, market, data\_type):  
 try:  
 # Perform scraping operations  
 result = perform\_scraping(ticker, market, data\_type)  
 return result  
 except SoftTimeLimitExceeded:  
 # Clean up resources before hard timeout  
 cleanup\_browser\_resources()  
 raise  
 except Exception as e:  
 # Log error and prepare for retry  
 logger.error(f"Scraping failed: {str(e)}")  
 raise self.retry(countdown=60, max\_retries=3)

* --

## Data Storage and Retrieval

### Firebase Integration

### Database Structure

Firebase Realtime Database Schema:  
/  
├── financial\_data/  
│ ├── {ticker}\_{market}\_{data\_type}/  
│ │ ├── company\_name: "Apple Inc."  
│ │ ├── ticker: "AAPL"  
│ │ ├── market: "XNAS"  
│ │ ├── data\_type: "INCOME\_STATEMENT"  
│ │ ├── last\_updated: "2024-01-15T10:30:00Z"  
│ │ ├── periods: ["TTM", "2022-12", "2021-12"]  
│ │ └── data: {  
│ │ ├── "Revenue": [394328000000, 365817000000]  
│ │ ├── "Net Income": [99803000000, 94680000000]  
│ │ └── ...  
│ │ }  
│ └── ...  
├── request\_logs/  
│ ├── {timestamp}/  
│ │ ├── ip\_address: "192.168.1.1"  
│ │ ├── user\_agent: "Mozilla/5.0..."  
│ │ ├── ticker: "AAPL"  
│ │ ├── market: "XNAS"  
│ │ ├── endpoint: "/financial-statements-json"  
│ │ └── status: "success"  
│ └── ...  
└── error\_logs/  
 └── {timestamp}/  
 ├── error\_type: "TICKER\_NOT\_FOUND"  
 ├── message: "Stock symbol not found"  
 └── stack\_trace: "..."

### Data Storage Process

def store\_financial\_data(ticker, market, data\_type, processed\_data):  
 # Create unique key  
 key = f"{ticker}\_{market}\_{data\_type}"  
   
 # Prepare data structure  
 firebase\_data = {  
 'company\_name': processed\_data.get('company\_name'),  
 'ticker': ticker,  
 'market': market,  
 'data\_type': data\_type,  
 'last\_updated': datetime.utcnow().isoformat(),  
 'periods': processed\_data.get('periods', []),  
 'data': processed\_data.get('financial\_data', {}),  
 'currency': processed\_data.get('currency', 'USD'),  
 'source': 'morningstar.com'  
 }  
   
 # Store in Firebase  
 database.child('financial\_data').child(key).set(firebase\_data)  
   
 # Set expiration (24 hours)  
 database.child('financial\_data').child(key).update({  
 'expires\_at': (datetime.utcnow() + timedelta(hours=24)).isoformat()  
 })

### Local SQLite Integration

* Request Tracking Model\*:

class APIRequest(models.Model):  
 title = models.CharField(max\_length=200)  
 endpoint = models.TextField()  
 ticker = models.TextField()  
 market = models.TextField()  
 location = models.TextField() # IP geolocation  
 request\_type = models.TextField()  
 user\_email = models.TextField(null=True, blank=True)  
 user\_country = models.TextField()  
 response\_time = models.FloatField(null=True)  
 status\_code = models.IntegerField(null=True)  
 created = models.DateTimeField(default=timezone.now)  
   
 def \_\_str\_\_(self):  
 return f"{self.ticker} - {self.endpoint} - {self.created}"

* --

## Error Handling and Recovery

### Error Classification System

### Error Types and Responses

ERROR\_TYPES = {  
 'NETWORK\_ERROR': {  
 'code': 'NET\_001',  
 'message': 'Network connection failed',  
 'retry': True,  
 'max\_retries': 3,  
 'http\_status': 503  
 },  
   
 'TICKER\_NOT\_FOUND': {  
 'code': 'DATA\_001',  
 'message': 'Stock ticker not found',  
 'retry': False,  
 'max\_retries': 0,  
 'http\_status': 404  
 },  
   
 'WEBSITE\_CHANGED': {  
 'code': 'SCRAPE\_001',  
 'message': 'Website structure changed',  
 'retry': True,  
 'max\_retries': 2,  
 'http\_status': 502  
 },  
   
 'DOWNLOAD\_TIMEOUT': {  
 'code': 'TIME\_001',  
 'message': 'File download timeout',  
 'retry': True,  
 'max\_retries': 3,  
 'http\_status': 408  
 }  
}

### Recovery Strategies

### Automatic Retry Logic

def handle\_scraping\_error(error, task, ticker, market, data\_type):  
 error\_type = classify\_error(error)  
   
 if error\_type['retry'] and task.request.retries < error\_type['max\_retries']:  
 # Calculate exponential backoff  
 countdown = min(60 \* (2 \*\* task.request.retries), 300)  
   
 # Log retry attempt  
 logger.info(f"Retrying {ticker} scraping in {countdown} seconds")  
   
 # Schedule retry  
 raise task.retry(  
 countdown=countdown,  
 max\_retries=error\_type['max\_retries'],  
 exc=error  
 )  
 else:  
 # Log final failure  
 log\_scraping\_failure(ticker, market, data\_type, error)  
   
 # Return error response  
 return {  
 'error': error\_type['code'],  
 'message': error\_type['message'],  
 'ticker': ticker,  
 'market': market  
 }

### Graceful Degradation

Primary Strategy → Fallback Strategy → Final Option  
 ↓ ↓ ↓  
Stealth Chrome → Regular Chrome → Cached Data  
 ↓ ↓ ↓  
Full Automation → Simplified → Error Response

* --

## Performance Optimization

### Caching Strategy

### Multi-Level Caching

Level 1: In-Memory Cache (Redis)  
├── Recently requested data (1 hour TTL)  
├── Popular stock symbols (6 hours TTL)  
└── API response templates (24 hours TTL)  
  
Level 2: Database Cache (Firebase)  
├── Processed financial data (24 hours TTL)  
├── Company metadata (7 days TTL)  
└── Market information (30 days TTL)  
  
Level 3: File System Cache  
├── Downloaded Excel files (1 hour TTL)  
├── Processed JSON data (6 hours TTL)  
└── Error logs and debugging data (7 days TTL)

### Cache Implementation

def get\_cached\_data(ticker, market, data\_type):  
 # Try Redis first (fastest)  
 redis\_key = f"stock\_data:{ticker}:{market}:{data\_type}"  
 cached\_data = redis\_client.get(redis\_key)  
   
 if cached\_data:  
 return json.loads(cached\_data)  
   
 # Try Firebase (medium speed)  
 firebase\_key = f"{ticker}\_{market}\_{data\_type}"  
 firebase\_data = database.child('financial\_data').child(firebase\_key).get()  
   
 if firebase\_data.val() and not is\_expired(firebase\_data.val()):  
 # Cache in Redis for next time  
 redis\_client.setex(redis\_key, 3600, json.dumps(firebase\_data.val()))  
 return firebase\_data.val()  
   
 # No cache available - need fresh scraping  
 return None

### Resource Management

### Browser Instance Pooling

class BrowserPool:  
 def \_\_init\_\_(self, max\_instances=3):  
 self.pool = []  
 self.max\_instances = max\_instances  
 self.active\_count = 0  
   
 def get\_browser(self):  
 if self.pool:  
 return self.pool.pop()  
 elif self.active\_count < self.max\_instances:  
 self.active\_count += 1  
 return create\_stealth\_driver()  
 else:  
 # Wait for available browser  
 time.sleep(5)  
 return self.get\_browser()  
   
 def return\_browser(self, driver):  
 # Clean browser state  
 driver.delete\_all\_cookies()  
 driver.get('about:blank')  
   
 # Return to pool  
 self.pool.append(driver)

* --

## Security and Stealth Features

### Anti-Detection Measures

### Browser Fingerprinting Prevention

STEALTH\_OPTIONS = {  
 'user\_agents': [  
 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36',  
 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_15\_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36',  
 'Mozilla/5.0 (X11; Linux x86\_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36'  
 ],  
   
 'screen\_resolutions': [  
 (1920, 1080), (1366, 768), (1440, 900), (1536, 864)  
 ],  
   
 'viewport\_sizes': [  
 (1200, 800), (1366, 768), (1440, 900)  
 ]  
}  
  
def randomize\_browser\_profile(options):  
 # Random user agent  
 options.add\_argument(f"--user-agent={random.choice(STEALTH\_OPTIONS['user\_agents'])}")  
   
 # Random screen size  
 width, height = random.choice(STEALTH\_OPTIONS['screen\_resolutions'])  
 options.add\_argument(f"--window-size={width},{height}")  
   
 # Disable automation indicators  
 options.add\_argument("--disable-blink-features=AutomationControlled")  
 options.add\_experimental\_option("excludeSwitches", ["enable-automation"])  
 options.add\_experimental\_option('useAutomationExtension', False)  
   
 return options

### Request Rate Limiting

### Intelligent Throttling

class RateLimiter:  
 def \_\_init\_\_(self):  
 self.request\_times = {}  
 self.min\_interval = 30 # seconds between requests  
   
 def can\_make\_request(self, ticker, market):  
 key = f"{ticker}\_{market}"  
 last\_request = self.request\_times.get(key, 0)  
 current\_time = time.time()  
   
 if current\_time - last\_request >= self.min\_interval:  
 self.request\_times[key] = current\_time  
 return True  
   
 return False  
   
 def wait\_time(self, ticker, market):  
 key = f"{ticker}\_{market}"  
 last\_request = self.request\_times.get(key, 0)  
 elapsed = time.time() - last\_request  
 return max(0, self.min\_interval - elapsed)

### Data Privacy and Compliance

### User Data Protection

def anonymize\_request\_data(request\_data):  
 # Hash IP addresses  
 if 'ip\_address' in request\_data:  
 request\_data['ip\_hash'] = hashlib.sha256(  
 request\_data['ip\_address'].encode()  
 ).hexdigest()[:16]  
 del request\_data['ip\_address']  
   
 # Remove sensitive headers  
 sensitive\_headers = ['authorization', 'cookie', 'x-api-key']  
 if 'headers' in request\_data:  
 request\_data['headers'] = {  
 k: v for k, v in request\_data['headers'].items()  
 if k.lower() not in sensitive\_headers  
 }  
   
 return request\_data

* --

This comprehensive documentation explains exactly how the Stock Data Web Scraper application works, from initial user request to final data delivery. The system combines web automation, asynchronous processing, intelligent caching, and robust error handling to provide reliable financial data extraction services.