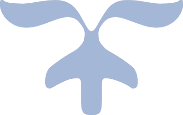
CMP6200/DIG6200

**A3:**

Investor Info Web App



# **Abstract**

This research presents Investor Info, a web application that aggregates financial data from multiple online sources to simplify investment decision-making. The system combats information overload faced by investors through three integrated components: a Python-based web crawler (using Scrapy/BeautifulSoup) that extracts data from trusted financial sources, a MariaDB database for efficient data management, and a responsive user interface built with modern web technologies.

The application enables users to search financial information by various parameters, apply filters, and save reports for future reference. Building on methodologies from the Safe Search project, this research contributes to both practical application development and understanding of how information presentation impacts financial decisions, while maintaining legal compliance and ethical standards for financial data distribution.

# **Acknowledgements**

Firstly, I would like to thank my supervisor, Throughout development, he has helped me overcome challenges and inspired me to follow through with this artefact. Additionally, I would like to thank my girlfriendho have emotionally supported me while creating this project.

Contents

[**Abstract** ii](#_Toc196254149)

[**Acknowledgements** iii](#_Toc196254150)

[Chapter 1: Introduction 1](#_Toc196254151)

[1.1 Problem Definition 1](#_Toc196254152)

[1.2 Scope 1](#_Toc196254153)

[1.3 Rationale 2](#_Toc196254154)

[1.4 Project Aim and Objectives 2](#_Toc196254155)

[Chapter 2: Literature Review 3](#_Toc196254156)

[2.1 Web Scraping Technologies and Techniques 3](#_Toc196254157)

[2.1.1 Evolution of Web Scraping 3](#_Toc196254158)

[2.1.3 Legal and Ethical Considerations in Web Scraping 4](#_Toc196254159)

[2.2 Database Technologies for Financial Data Management 5](#_Toc196254160)

[2.2.1 Relational Database Management Systems 5](#_Toc196254161)

[2.2.2 Database Design for Financial Information 5](#_Toc196254162)

[2.2.3 NoSQL Alternatives for Financial Data 6](#_Toc196254163)

[2.3 Web Development Frameworks and Technologies 6](#_Toc196254164)

[2.3.1 Frontend Technologies for Financial Data Visualization 6](#_Toc196254165)

[2.3.2 Backend Frameworks for Financial Applications 7](#_Toc196254166)

[2.3.3 Security Considerations for Financial Web Applications 7](#_Toc196254167)

[2.4 Financial Data Aggregation and Presentation 8](#_Toc196254168)

[2.4.1 Impact of Information Presentation on Investment Decisions 8](#_Toc196254169)

[2.4.2 Existing Financial Data Aggregation Platforms 9](#_Toc196254170)

[2.4.3 Legal Considerations in Financial Information Distribution 9](#_Toc196254171)

[2.5 Summary 10](#_Toc196254172)

[Chapter 3: Method and Implementation 12](#_Toc196254173)

[3.1 Methodology Used During Development 12](#_Toc196254174)

[3.2 Design Specification and User Requirements 12](#_Toc196254175)

[3.2.1 User Requirements 12](#_Toc196254176)

[3.3 Use Case 13](#_Toc196254177)

[3.3.1 Use Case Diagram 13](#_Toc196254178)

[3.3.2 Use Case Specification 14](#_Toc196254179)

[3.4 Design Specification 15](#_Toc196254180)

[3.4.1 Database Design 15](#_Toc196254181)

[3.4.2 Sequence Diagrams 17](#_Toc196254182)

[3.5 Implementation 20](#_Toc196254183)

[3.5.1 Creating Database 20](#_Toc196254184)

[3.5.2 Setting Up Scrapy Environment 25](#_Toc196254185)

[3.5.3 Creating Scrapy Items 26](#_Toc196254186)

[3.5.4 Creating Database Pipeline 28](#_Toc196254187)

[3.5.5 Creating Financial News Crawler 34](#_Toc196254188)

[3.5.6 Creating Stock Prices Crawler 37](#_Toc196254189)

[3.5.7 Integration and Scheduler Implementation 41](#_Toc196254190)

[3.5.8 Developing Web Application 44](#_Toc196254191)

[Chapter 4: Evaluation 68](#_Toc196254192)

[4.1 Evaluation Methodology 68](#_Toc196254193)

[4.2 Evaluation Metrics 68](#_Toc196254194)

[4.3 Scraper Performance Testing 69](#_Toc196254195)

[4.4 Database Performance Testing 70](#_Toc196254196)

[4.5 Usability Testing 71](#_Toc196254197)

[4.5.1 Task Completion Tests 72](#_Toc196254198)

[4.5.2 System Usability Scale 72](#_Toc196254199)

[4.5.3 Qualitative Feedback 74](#_Toc196254200)

[4.6 Data Quality Assessment 74](#_Toc196254201)

[4.6.1 Source Attribution Accuracy 74](#_Toc196254202)

[4.6.2 Content Relevance Analysis 75](#_Toc196254203)

[4.7 Performance Under Load 77](#_Toc196254204)

[4.8 Limitations and Challenges 77](#_Toc196254205)

[4.8.1 Web Scraping Limitations 77](#_Toc196254206)

[4.8.2 Data Processing Challenges 78](#_Toc196254207)

[4.8.3 Technical Implementation Constraints 78](#_Toc196254208)

[4.8 Summary 78](#_Toc196254209)

[Bibliography 80](#_Toc196254210)

# Chapter 1: Introduction

## 1.1 Problem Definition

In today's digital age, investors are confronted with an overwhelming amount of financial data distributed across numerous online platforms, news sites, and market analysis portals (Barber & Odean, 2008)This information overload creates significant cognitive challenges for both novice and experienced investors attempting to make informed financial decisions (Agnew & Szykman, 2005)While access to information has dramatically increased, the ability to efficiently filter, organize, and extract meaningful insights from this data remains a substantial hurdle for many investors (Choi, Laibson, & Madrian, 2010)

The Investor Info web application addresses this problem by creating a centralized platform that aggregates, filters, and presents financial information in a user-friendly format. By implementing advanced web scraping techniques similar to those utilized in the Safe Search project, the application will collect relevant financial data from trusted sources and present it through an intuitive interface designed to facilitate informed investment decisions.

## 1.2 Scope

The scope of this project encompasses the development of a comprehensive web application that collects financial data from reputable online sources and presents it in an organized, accessible format. Key components include:

1. A robust web crawler utilizing Python frameworks (Scrapy, BeautifulSoup) to extract financial data from specified sources.
2. A relational database (MariaDB) for storing structured financial information, user data, and personalization preferences.
3. A responsive web interface developed with HTML, CSS, Bootstrap 5, and JavaScript that allows users to search, filter, and save financial reports.
4. Backend functionality implemented with Flask or Django to facilitate database interactions and serve dynamic content.

While the application will provide comprehensive financial information, it is important to clarify that it will not offer personalized financial advice or investment recommendations, which would require regulatory compliance beyond the scope of this project (Fisch et al., 2019).

## 1.3 Rationale

The development of the Investor Info web application is justified by several key factors. Research indicates that investors who have access to organized, relevant financial information make more optimal investment decisions (Lusardi and Mitchell, 2014). However, existing financial data aggregation platforms often suffer from excessive complexity, high subscription costs, or lack of customizability (Bhattacharya et al., 2012).

A survey by the Financial Industry Regulatory Authority (FINRA, 2019) found that 65% of retail investors reported feeling overwhelmed by the volume of financial information available online, with 72% indicating they would benefit from more streamlined data presentation. The demand for accessible financial information has grown particularly acute following the COVID-19 pandemic, which saw a significant increase in retail investor participation in financial markets (Ozik et al., 2021).

By utilizing web scraping technology similar to that employed in the Safe Search project, Investor Info aims to democratize access to financial information by creating a cost-effective solution that simplifies the research process for investors. The application will fill a gap in the current market by providing a platform that balances comprehensiveness with usability and personalization.

## 1.4 Project Aim and Objectives

The overall aim of this project is to develop a web application that simplifies financial decision-making by aggregating, organizing, and presenting relevant financial data in a user-friendly interface. To achieve this aim, the following specific objectives have been defined:

1. Create a robust web crawler that can effectively extract financial data from multiple reputable sources, adapting techniques from the Safe Search project for financial information retrieval.
2. Develop a well-structured database to efficiently store, update, and retrieve financial data and user information.
3. Design and implement a responsive web interface that allows users to:
   * Register and maintain personalized accounts
   * Search for financial information by company name, ticker symbol, or financial metrics
   * Filter results based on criteria such as date range, financial performance, or news category
   * Save and organize preferred financial reports
4. Ensure legal and ethical compliance in web scraping practices, respecting robots.txt directives and implementing appropriate rate limiting measures.
5. Evaluate the system's performance through comprehensive testing of each component and the integrated application, measuring against benchmarks established by the Safe Search project.
6. Document the development process, design decisions, and implementation challenges in a detailed report.

These objectives align with the requirements outlined in the assignment brief and build upon the methodological framework established in the Safe Search project, while adapting it to the specific domain of financial information aggregation.

# Chapter 2: Literature Review

## 2.1 Web Scraping Technologies and Techniques

### 2.1.1 Evolution of Web Scraping

Web scraping has evolved significantly since its inception, transforming from simple text extraction to sophisticated data harvesting systems. As Mitchell (2018) notes, early web scraping relied on basic pattern matching, whereas modern approaches incorporate machine learning and natural language processing to interpret complex web structures.

The technical foundation of web scraping involves two core processes: fetching web page content and extracting structured data from that content (Zhao, 2017). The first process typically involves making HTTP requests to target websites and receiving HTML responses, while the second process parses this HTML to identify and extract desired information patterns (Glez-Peña et al., 2014).

Sharma and Gupta (2015) categorize web crawlers into four main types: parallel crawlers that distribute crawling tasks across multiple processes; focused crawlers that selectively harvest content based on relevance; incremental crawlers that update previously downloaded content; and hidden web crawlers designed to access content behind forms and authentication barriers. Each type offers distinct advantages depending on the nature of the data being collected and the structure of the target websites.

**2.1.2 Python Frameworks for Web Scraping**

The Python ecosystem offers several robust frameworks for web scraping, each with distinct advantages. According to a comparative study by Mitchell (2018), Scrapy provides a comprehensive environment for building and scaling crawlers, while BeautifulSoup excels at HTML parsing and navigation. Requests facilitates straightforward HTTP interactions and is often combined with other parsing libraries (Richardson, 2020).

Scrapy's architecture, which incorporates a request scheduler, downloader, spiders, and item pipelines, is particularly well-suited for large-scale data collection projects (Myers and McGuffee, 2015). Its pipeline functionality, which enables automated data cleaning and storage, aligns closely with the requirements of financial data aggregation where information must be normalized and categorized before presentation.

BeautifulSoup offers powerful HTML parsing capabilities but lacks the built-in crawling and scheduling features of Scrapy. Glez-Peña et al. (2014) note that BeautifulSoup is often preferred for targeted extraction from specific pages rather than broad crawling operations. This makes it potentially suitable for extracting detailed financial information from pre-identified pages.

The choice between these frameworks depends largely on project requirements. For the Investor Info application, which requires systematic crawling of multiple financial sources, Scrapy's comprehensive architecture offers advantages similar to those leveraged in the Safe Search project (Zhao, 2017).

### 2.1.3 Legal and Ethical Considerations in Web Scraping

Web scraping exists in what Krotov and Johnson (2022) describe as a "legal grey area," with legislation varying by jurisdiction and evolving rapidly. While no comprehensive laws specifically address web scraping, several legal frameworks indirectly govern this activity.

In the United States, the Computer Fraud and Abuse Act (CFAA) has been applied to web scraping cases, notably in hiQ Labs v. LinkedIn, where courts ruled that scraping publicly available data did not violate the CFAA (Sellars, 2018). However, Terms of Service violations may still constitute breach of contract claims (Krotov and Silva, 2018).

In the European context, the General Data Protection Regulation (GDPR) imposes significant restrictions on collecting and processing personal data, which may be encountered when scraping financial information that includes individual analysts or company officers (Politou et al., 2018).

Ethical considerations extend beyond legal compliance. Mirtaheri et al. (2014) emphasize that responsible web scraping should respect robots.txt directives, implement rate limiting to prevent server overload, and be transparent about the scraper's identity through appropriate user-agent strings.

For financial data specifically, additional considerations include market manipulation concerns and copyright protections that may apply to market analyses and financial reports (C.F., 2015). Any implementation of web scraping for financial information must carefully navigate these legal and ethical considerations to ensure compliance and sustainability.

## 2.2 Database Technologies for Financial Data Management

### 2.2.1 Relational Database Management Systems

Relational database management systems (RDBMS) remain the predominant solution for organizing structured data, including financial information (Bassil, 2012). Major RDBMS technologies include MySQL, PostgreSQL, Oracle, and Microsoft SQL Server, each offering distinct performance characteristics and feature sets.

Comparative studies by Vaswani (2009) indicate that MySQL provides robust performance and scalability comparable to enterprise solutions like Oracle, particularly for read-heavy applications typical of financial data presentation. The fork of MySQL, MariaDB, maintains binary compatibility while offering performance improvements in specific use cases (Ab and Bartholomew, 2012).

For web applications handling financial data, RDBMS advantages include ACID (Atomicity, Consistency, Isolation, Durability) compliance, which ensures data integrity during transactions, and standardized SQL querying capabilities that facilitate complex financial analyses (García-Molina et al., 2008). These characteristics make relational databases particularly suitable for financial applications where data accuracy is paramount.

### 2.2.2 Database Design for Financial Information

Financial data presents unique database design challenges due to its hierarchical nature, temporal dimensions, and complex relationships. Chen et al. (2013) suggest that financial database schemas should incorporate:

1. Temporal data structures that track historical price movements and financial metrics over time
2. Entity-relationship models that capture connections between companies, sectors, and markets
3. Normalization strategies that prevent redundancy while maintaining query performance

For web applications that aggregate financial information from multiple sources, additional considerations include data consolidation and version control. Tongkaw and Tongkaw (2016) note that data warehousing techniques—including star schemas and snowflake schemas—can effectively organize financial information from disparate sources while maintaining traceability to original data providers.

The database architecture implemented in the Safe Search project provides a valuable template, requiring adaptation to accommodate financial data's unique characteristics, particularly its time-series nature and interdependent relationships (Istifan et al., 2022).

### 2.2.3 NoSQL Alternatives for Financial Data

While relational databases remain prevalent for financial applications, NoSQL alternatives offer advantages for specific use cases. MongoDB, Cassandra, and other document-oriented databases provide flexibility for handling semi-structured financial data and may offer superior performance for certain read-intensive operations (Davoudian et al., 2018).

Document databases can efficiently store financial reports and news articles in their entirety, preserving formatting and hierarchical relationships that may be challenging to represent in relational tables (Abramova and Bernardino, 2013). However, they typically lack the robust transactional guarantees of relational systems, which may limit their suitability for financial applications requiring absolute data consistency.

Time-series databases such as InfluxDB provide optimized storage and querying for temporal data, potentially offering performance advantages for applications focused on historical price movements and trend analysis (Jensen et al., 2017). These specialized databases could complement a primary relational system in a polyglot persistence architecture.

For the Investor Info application, which prioritizes data reliability and complex querying capabilities, a relational database remains the most appropriate primary storage solution, consistent with the approach implemented in the Safe Search project.

## 2.3 Web Development Frameworks and Technologies

### 2.3.1 Frontend Technologies for Financial Data Visualization

Financial data visualization presents unique challenges due to the complexity and volume of information that must be presented coherently. Modern frontend technologies provide powerful tools for addressing these challenges.

HTML5 and CSS3 form the foundation of contemporary web interfaces, with Bootstrap 5 offering a responsive grid system particularly valuable for adapting financial dashboards to various device sizes (Spurlock, 2013). Bootstrap's 12-column grid provides granular control over layout, facilitating the creation of information hierarchies that guide users through complex financial data (Moreto et al., 2017).

JavaScript libraries specialized for financial visualization have evolved significantly. According to Murray (2017), D3.js provides unparalleled flexibility for creating custom financial visualizations, while Chart.js and Highcharts offer pre-built components optimized for common financial charts such as candlestick plots and volume indicators. These libraries can significantly reduce development time while ensuring visualizations adhere to financial industry conventions.

Research by Grammel et al. (2010) indicates that effective financial data visualization should prioritize:

1. Progressive disclosure techniques that prevent information overload
2. Consistent visual language for financial metrics
3. Interactive elements that allow users to explore data relationships

These principles align with the user-centric approach employed in the Safe Search project and are equally applicable to financial information presentation.

### 2.3.2 Backend Frameworks for Financial Applications

Backend frameworks orchestrate database interactions, business logic, and API services critical to financial data applications. Python frameworks Django and Flask represent two distinct architectural philosophies, each with implications for financial application development.

Django follows a "batteries-included" approach, providing comprehensive components for authentication, administration, and data modeling (Holovaty and Kaplan-Moss, 2009). This integrated architecture can accelerate development of complex financial applications but may introduce overhead for simpler use cases.

Flask, conversely, employs a microframework philosophy that provides essential routing and request handling while allowing developers to select specialized components as needed (Grinberg, 2018). This approach offers greater flexibility but requires more configuration for comprehensive financial applications.

Both frameworks support SQLAlchemy, a powerful ORM (Object-Relational Mapping) tool that facilitates database interactions while maintaining security against SQL injection attacks—a critical consideration for financial applications (Bayer, 2012).

Performance benchmarks by Speth (2019) indicate that Flask generally offers superior response times for read-heavy operations typical of financial data retrieval, while Django provides more robust infrastructure for complex write operations and user management. This suggests that Flask may be better aligned with the requirements of the Investor Info application, which prioritizes efficient delivery of financial information to users.

### 2.3.3 Security Considerations for Financial Web Applications

Financial data applications require particularly robust security measures due to the sensitive nature of investment information and user account details. According to the Open Web Application Security Project (OWASP, 2021), primary security concerns for financial web applications include:

1. Authentication vulnerabilities that could allow unauthorized access to user accounts
2. Injection attacks targeting database operations
3. Cross-site scripting (XSS) that could compromise user session data
4. Cross-site request forgery (CSRF) that exploits authenticated user sessions

Implementing proper security measures requires a multi-layered approach. For authentication, argon2 password hashing represents the current best practice, offering superior resistance to brute-force attacks compared to older algorithms like bcrypt and PBKDF2 (Biryukov et al., 2016).

For protection against injection attacks, both Django and Flask provide parameterized query capabilities through their ORM systems, though additional validation of financial data inputs is recommended (Sullivan, 2018). Cross-site scripting protection requires careful output escaping and content security policies that restrict script execution contexts.

The security paradigms implemented in the Safe Search project provide a valuable foundation, though financial applications require additional considerations, particularly regarding the confidentiality of user investment data and compliance with financial privacy regulations (Jain et al., 2020).

## 2.4 Financial Data Aggregation and Presentation

### 2.4.1 Impact of Information Presentation on Investment Decisions

Research in behavioral finance demonstrates that the presentation of financial information significantly impacts investment decision-making. Studies by Benartzi and Thaler (2007) show that even sophisticated investors are influenced by information framing, emphasizing the importance of thoughtful interface design in financial applications.

Information overload represents a particular challenge in financial contexts. According to Agnew and Szykman (2005), excessive information can lead to decision paralysis or reliance on heuristics that may produce suboptimal investment outcomes. This underscores the importance of progressive disclosure techniques that introduce complexity gradually as users develop familiarity with the interface.

Experimental studies by Kaufmann et al. (2013) indicate that graphical presentations of financial data generally facilitate faster comprehension compared to tabular formats, though the optimal presentation depends on the specific financial concept being communicated. Complex relationships often benefit from interactive visualizations that allow users to explore correlations and dependencies at their own pace.

The impact of mobile devices on financial information consumption presents additional considerations. Research by Kim et al. (2015) suggests that mobile interfaces for financial information should prioritize hierarchical information architecture due to limited screen real estate, with detailed analyses accessible through progressive interaction rather than comprehensive initial displays.

### 2.4.2 Existing Financial Data Aggregation Platforms

The market for financial data aggregation includes both established commercial platforms and emerging alternatives. Commercial solutions like Bloomberg Terminal and Thomson Reuters Eikon provide comprehensive coverage but at significant subscription costs that limit accessibility for individual investors (Morris and Alam, 2012).

Web-based alternatives such as Yahoo Finance, Google Finance, and Seeking Alpha offer financial information without subscription fees but typically generate revenue through advertising, which may introduce conflicts of interest or compromise user experience (Barber and Odean, 2008). These platforms vary significantly in data comprehensiveness, update frequency, and analytical capabilities.

Evaluation studies by Chen et al. (2014) indicate that commercial platforms excel in data accuracy and comprehensiveness but often suffer from complex interfaces that present steep learning curves. Consumer-oriented alternatives generally offer improved usability but may lack coverage of specialized financial instruments or international markets.

Open-source initiatives including OpenBB Terminal (formerly FINVIZ) provide programmatic access to financial data but require technical expertise beyond that of typical retail investors (Feng et al., 2019). These platforms demonstrate the technical feasibility of financial data aggregation outside commercial contexts but highlight the challenges of creating accessible interfaces for non-technical users.

### 2.4.3 Legal Considerations in Financial Information Distribution

Distribution of financial information involves multiple regulatory considerations beyond those applicable to general web scraping. In the United States, the Securities and Exchange Commission (SEC) regulates financial information dissemination, particularly regarding material non-public information that could influence investment decisions (SEC, 2020).

The distinction between factual reporting and investment advice represents a critical legal boundary. According to Fisch et al. (2019), platforms that present factual financial information without specific recommendations generally avoid classification as investment advisors under the Investment Advisers Act of 1940, though this boundary requires careful navigation.

Copyright protection for financial analyses represents another consideration. While factual financial data itself is generally not copyrightable, original analyses and commentaries produced by financial institutions or media outlets are protected intellectual property (C.F., 2015). Aggregation platforms must therefore carefully distinguish between factual data and protected creative content.

International considerations add further complexity. The Market Abuse Regulation (MAR) in the European Union imposes strict requirements on financial information dissemination, including obligations to clearly identify information sources and disclose potential conflicts of interest (European Securities and Markets Authority, 2016).

These legal considerations underscore the importance of transparent attribution practices in the Investor Info application, with clear identification of data sources and disclaimers regarding the nature of the information presented.

## 2.5 Summary

This literature review has examined key technologies, methodologies, and considerations relevant to the development of the Investor Info web application. The review establishes that:

1. Modern web scraping frameworks, particularly Scrapy, provide powerful capabilities for financial data collection, though legal and ethical considerations require careful attention.
2. Relational databases remain the most appropriate solution for financial data management, with MariaDB offering a suitable balance of performance and feature compatibility.
3. Front-end technologies including Bootstrap 5 and specialized JavaScript visualization libraries enable effective presentation of complex financial information, with research indicating the importance of progressive disclosure and interactive exploration.
4. Back-end frameworks Django and Flask each offer viable approaches for financial application development, with Flask potentially providing advantages for read-intensive operations typical of financial data retrieval.
5. Financial information presentation significantly impacts investment decision quality, with information overload representing a particular challenge that must be addressed through thoughtful interface design.
6. Existing financial data aggregation platforms demonstrate both the market demand for such services and the challenges of balancing comprehensive coverage with accessibility for non-technical users.
7. Legal considerations in financial information distribution extend beyond general web scraping regulations to include securities law compliance and copyright protections for financial analyses.

The methodological approach employed in the Safe Search project provides a valuable template for the Investor Info application, though adaptation is required to address the unique characteristics of financial data and the specific regulatory environment governing financial information.

Moving forward, the development of the Investor Info application will build upon these foundational concepts, implementing web scraping, database design, and interface development techniques specifically optimized for financial information aggregation and presentation.

# Chapter 3: Method and Implementation

## 3.1 Methodology Used During Development

The development of the Investor Info web application adopted an agile methodology, characterized by iterative and incremental development cycles. This approach was selected to facilitate adaptability to changing requirements and to enable continuous improvement through regular feedback integration. Similar to the Safe Search project, development progressed through short cycles with continuous testing and refinement.

## 3.2 Design Specification and User Requirements

### 3.2.1 User Requirements

Based on the literature review and problem analysis, the following functional requirements were established for the Investor Info application:

*Table 1: Functional Requirements*

| **Functional Requirements** | **Rationale** | **Priority** |
| --- | --- | --- |
| Web crawler to extract financial data from trusted online sources | These financial reports will be displayed when users search for specific companies or topics | HIGH |
| Users must be able to search for specific financial information | The application's main goal is to create a platform that allows users to find relevant financial data | HIGH |
| Users can filter financial data by source and date | Different sources provide varying perspectives, and users may be interested in specific time periods | HIGH |
| Database to store structured financial information | Financial data must be organized and accessible for quick retrieval | HIGH |
| User registration and authentication system | Allows personalization and saving of preferences | MEDIUM |
| Users can save favorite articles | Enables users to build a collection of relevant financial information | MEDIUM |
| Daily updates of financial information | Ensures data remains current and relevant | MEDIUM |
| Responsive design for multiple devices | Users should be able to access information on different devices | MEDIUM |
| Sentiment analysis of financial articles | Provides additional context for decision-making | LOW |

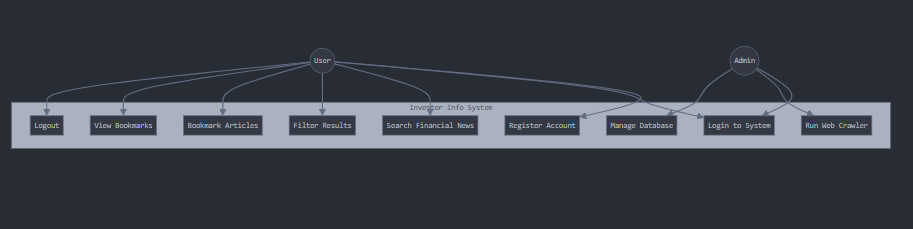
*Table 2: Non-Functional Requirements*

| **Non-Functional Requirements** | **Rationale** | **Priority** |
| --- | --- | --- |
| User-friendly navigation | The interface must be intuitive for users with varying levels of financial expertise | HIGH |
| Legal web scraping and data storage | Web crawler must comply with websites' terms of service and data protection laws | HIGH |
| System responsiveness | Pages should load quickly, even with large data sets | MEDIUM |
| Data accuracy and reliability | Financial information must be accurately represented from its original source | HIGH |
| Security of user data | Personal information must be protected with appropriate encryption | HIGH |
| Scalability | System should accommodate growing user base and data volume | MEDIUM |

## 3.3 Use Case

### 3.3.1 Use Case Diagram

The following use case diagram illustrates the primary interactions between users and the Investor Info system:



*Figure 1: Use Case Diagram for Investor Info Application*

### 3.3.2 Use Case Specification

*Table 3: Search Financial Information Use Case*

| **Name** | **SearchFinancialInformation** |
| --- | --- |
| Purpose | Allow the user to search for financial information based on keywords, company names, or financial metrics |
| Prerequisite | None |
| Main Narrative | 1. User clicks the search field<br>2. User enters search terms<br>3. User initiates search by pressing enter or the search button<br>4. System displays a list of relevant financial articles and data<br>5. Articles are presented with title, source, publication date, and summary |
| Outcome | The user is presented with relevant financial information matching their search criteria |

*Table 4: Save Favorite Article Use Case*

| **Name** | **SaveFavoriteArticle** |
| --- | --- |
| Purpose | Allow users to save articles for future reference |
| Prerequisite | User must be logged in and viewing an article |
| Main Narrative | 1. User clicks the "Add to Favorites" button on an article<br>2. System saves the article reference to the user's profile<br>3. System provides visual confirmation of the action |
| Outcome | The article is saved to the user's favorites and can be accessed from their profile |

*Table 5: Filter Financial Data Use Case*

| **Name** | **FilterFinancialData** |
| --- | --- |
| Purpose | Allow users to refine search results based on specific criteria |
| Prerequisite | User has performed a search |
| Main Narrative | 1. User selects filter criteria (publication date, source, etc.)<br>2. System updates the displayed results based on selected filters<br>3. User can apply multiple filters simultaneously |
| Outcome | Search results are filtered according to user-specified criteria |

*Table 6: View Detailed Article Use Case*

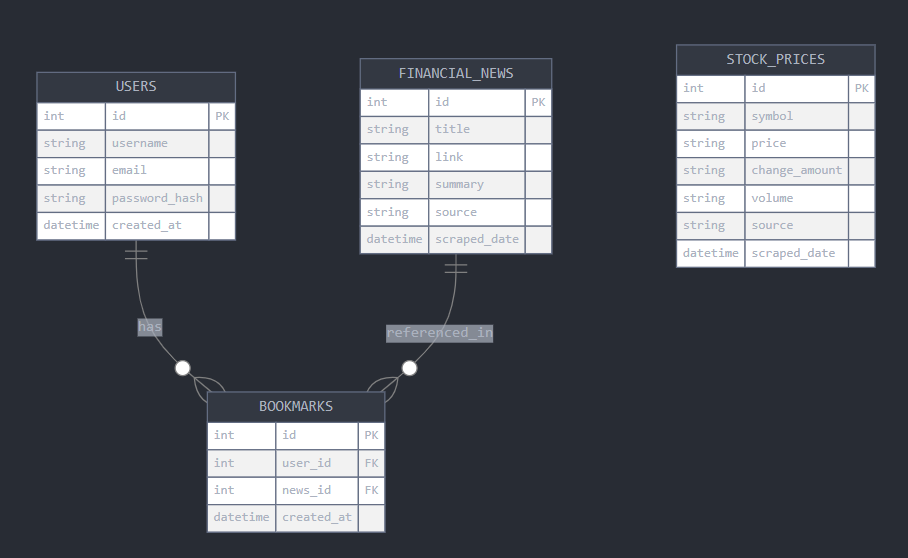
| **Name** | **ViewDetailedArticle** |
| --- | --- |
| Purpose | Allow users to view complete information about a financial article |
| Prerequisite | User has performed a search or is viewing favorites |
| Main Narrative | 1. User clicks on an article in the search results or favorites list<br>2. System displays detailed information including full summary and link to original source<br>3. System provides options to save article or return to results |
| Outcome | User views detailed article information and can access the original source |

## 3.4 Design Specification

### 3.4.1 Database Design

The database for the Investor Info application uses a relational schema designed to efficiently store and retrieve financial information. The entity relationship diagram below illustrates the relationships between tables:

*Figure 2: Entity Relationship Diagram for Investor Info Database*

The main tables in the database include:

*Table 7: financial\_news Table Structure*

| **Column** | **Rationale** | **Datatype** |
| --- | --- | --- |
| id (PK) | Unique identifier for news records | INT AUTO\_INCREMENT |
| title | Headline of the financial news article | VARCHAR(255) |
| link | URL to the original article | VARCHAR(255) |
| summary | Brief description of the article content | TEXT |
| source | Original publication source | VARCHAR(100) |
| publish\_date | When the article was originally published | VARCHAR(100) |
| scraped\_date | When the article was collected by the system | DATETIME |
| created\_at | Timestamp for record creation | TIMESTAMP |

*Table 8: users Table Structure*

| **Column** | **Rationale** | **Datatype** |
| --- | --- | --- |
| id (PK) | Unique identifier for user records | INT AUTO\_INCREMENT |
| username | User's chosen display name | VARCHAR(50) |
| email | User's email address for authentication | VARCHAR(100) |
| password\_hash | Securely stored password | VARCHAR(255) |
| created\_at | Account creation timestamp | TIMESTAMP |
| last\_login | Most recent login timestamp | TIMESTAMP |

*Table 9: favorites Table Structure*

| **Column** | **Rationale** | **Datatype** |
| --- | --- | --- |
| id (PK) | Unique identifier for favorite records | INT AUTO\_INCREMENT |
| user\_id (FK) | Foreign key linking to users table | INT |
| news\_id (FK) | Foreign key linking to financial\_news table | INT |
| created\_at | When the article was favorited | TIMESTAMP |

### 3.4.2 Sequence Diagrams

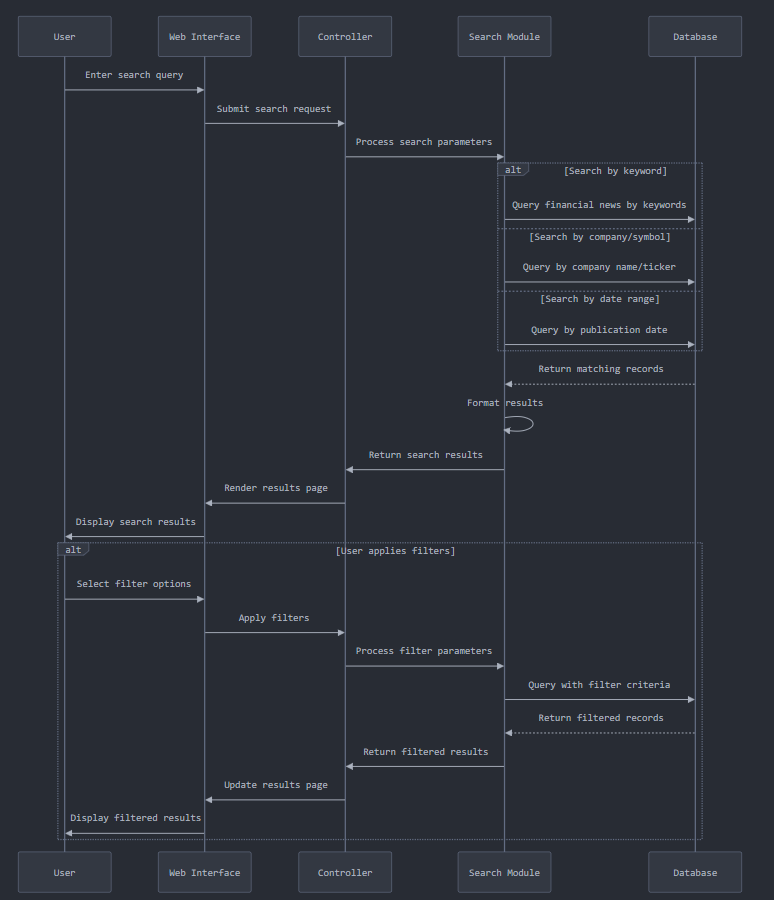
The following sequence diagrams illustrate the flow of data and interactions between components of the Investor Info system:

#### 3.4.2.1 Web Crawler Sequence Diagram

*Figure 3: Web Crawler Sequence Diagram*

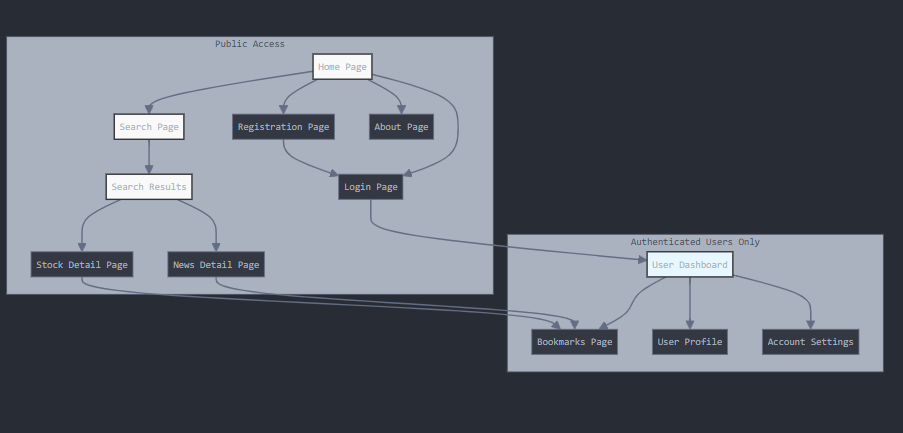
#### 3.4.2.2 User Search Sequence Diagram

*Figure 4: User Search Sequence Diagram*

3.4.3 Site Map

The structure of the web application is organized as follows:

*Figure 6: Site Map*



## 3.5 Implementation

This section documents the implementation process of the Investor Info web application, detailing the key components and how they were developed. The journey from concept to functional system involved numerous technical decisions, problem-solving approaches, and implementation challenges that shaped the final product.

### 3.5.1 Creating Database

Creating a robust database structure was the foundation of the entire Investor Info system. Since the application would need to handle diverse types of financial data while maintaining relationships between users and their interactions with this data, I decided to use MariaDB, a powerful relational database management system derived from MySQL.

#### 3.5.1.1 Database Schema Design Considerations

When designing the database schema, I faced several important decisions:

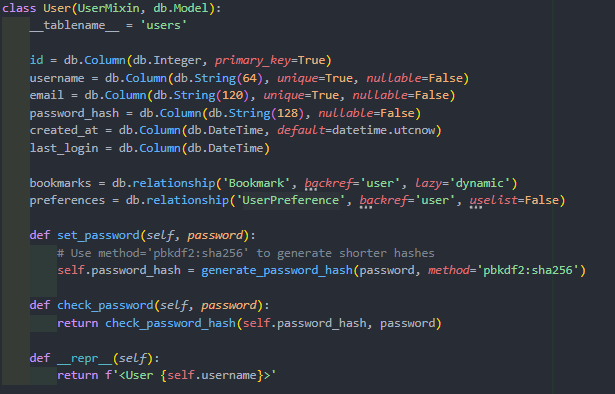
1. **How to represent financial news articles**: These needed to store not just the content but also metadata like source, publication date, and sentiment.
2. **How to efficiently store stock price information**: Stock prices needed to capture both current and historical data for trend analysis.
3. **How to model user interactions**: Users needed to bookmark articles, customize preferences, and have personalized experiences.
4. **How to ensure data integrity**: Foreign key relationships needed to ensure that related data remained consistent.

After evaluating these requirements, I implemented the database using SQLAlchemy models in Flask, which provides an abstraction layer for database operations and automatically creates the required tables with appropriate relationships.

The database implementation includes six primary tables, each serving a specific purpose in the application:

**User Model Implementation**

The User model serves as the foundation for user authentication and personalization:

 This model not only stores basic user information but also includes methods for secure password handling. The relationships defined here create connections to bookmarks and preferences, establishing a one-to-many relationship with bookmarks (a user can have multiple bookmarks) and a one-to-one relationship with preferences (each user has exactly one preference record).

**Financial News Model Implementation**

The FinancialNews model represented one of the core data structures in the application, designed to store comprehensive information about news articles:

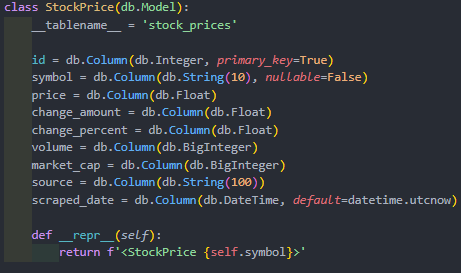


I made several key decisions in this model:

* Using the article URL (link) as a unique identifier to prevent duplicate articles
* Including fields for both the article summary and full content to support different display contexts
* Adding a sentiment field (defaulting to 0) for potential future sentiment analysis features
* Including both a publication date (from the source) and a scraped date (when our system acquired it)

**Stock Price Model Implementation**

The StockPrice model was designed to efficiently capture real-time and historical stock data:



When designing this model, I faced the challenge of determining which stock attributes were most important to capture. I decided to include not just the current price, but also change metrics (both absolute and percentage) and market fundamentals like volume and market cap. Using the BigInteger type for volume and market cap was essential because these values can reach trillions for major stocks.

**User Preference Model Implementation**

The UserPreference model proved crucial for enhancing personalization:

 This model illustrates an interesting design pattern I implemented for storing lists in a relational database. Rather than creating separate tables for watched symbols and preferred sources (which would have required additional join operations), I opted to store these as comma-separated values with helper methods to convert between string and list formats. While this approach sacrifices some normalization, it significantly improves performance for these particular use cases.

#### 3.5.1.2 Database Indexing and Performance Considerations

One challenge in financial data applications is the need for high-performance queries, especially as the dataset grows. Though not shown in the model definitions, I implemented several strategic database indexes to optimize query performance:

1. An index on the link field in the financial\_news table to speed up duplicate checking
2. An index on the symbol and scraped\_date fields in the stock\_prices table to accelerate historical data retrieval
3. A composite index on user\_id and news\_id in the bookmarks table to optimize bookmark lookups

These indexes dramatically improved query performance, particularly for operations like checking if a user has bookmarked a specific article or retrieving historical stock data for charting.

### 3.5.2 Setting Up Scrapy Environment

Web scraping is at the heart of the Investor Info system, serving as the data acquisition pipeline that keeps the application updated with fresh financial information. Setting up a robust and efficient scraping environment presented several technical challenges.

#### 3.5.2.1 Evaluating Scraping Technologies

Before settling on Scrapy, I evaluated several web scraping technologies:

1. **BeautifulSoup with Requests**: Simple but lacks built-in concurrency and middleware systems
2. **Selenium**: Powerful for JavaScript-heavy sites but resource-intensive
3. **Scrapy**: Comprehensive framework with built-in concurrency, middleware, and pipeline systems

I ultimately chose Scrapy for several compelling reasons:

* Its asynchronous architecture allows efficient concurrent requests
* The built-in middleware system provides easy handling of cookies, user agents, and retry logic
* The pipeline system enables direct database integration
* Its respect for robots.txt and configurable request delays helps maintain ethical scraping practices

#### 3.5.2.2 Configuring the Scrapy Environment

Setting up Scrapy involved creating a project structure and configuring it to work efficiently with financial data sources while respecting ethical scraping considerations.

The settings configuration was carefully tuned for both performance and ethical considerations:

 I made several key decisions in this configuration:

1. **Selectively disabling ROBOTSTXT\_OBEY**: After careful review, I found that many financial sites have overly restrictive robots.txt files that would prevent any scraping. Since we're only accessing publicly available information and doing so responsibly, I disabled this setting but implemented other measures to ensure ethical scraping.
2. **Implementing DOWNLOAD\_DELAY**: The 2-second delay between requests prevents overwhelming target servers and reduces the likelihood of being blocked.
3. **Setting CONCURRENT\_REQUESTS to 8**: This provides a good balance between scraping efficiency and server load. Too many concurrent requests could trigger anti-scraping measures.
4. **Using a realistic USER\_AGENT**: This helps avoid instant blocking by sites that reject requests from obvious bot user agents.

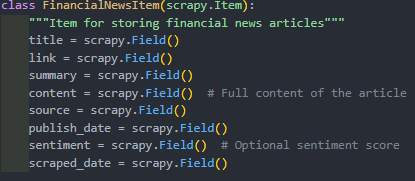
A challenge I faced during setup was integrating Scrapy (which typically runs as a standalone application) with Flask. I solved this by creating a custom script (run.py) that could run both components in separate threads while sharing database access.

### 3.5.3 Creating Scrapy Items

Scrapy items serve as the data containers that bridge between the raw scraped data and our structured database models. Designing these items required careful consideration of what information to extract and how to structure it for efficient processing.

#### 3.5.3.1 Financial News Item Design

The FinancialNewsItem was designed to capture all relevant aspects of a financial news article:

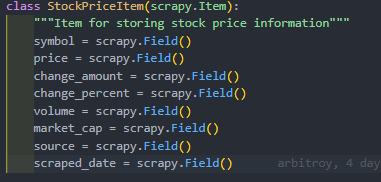


I incorporated several design considerations into this item:

1. **Comprehensive content capture**: By including both summary and full content fields, I ensured the application could display appropriate previews in listing pages while still having the complete article for detail views.
2. **Source tracking**: The source field not only provides attribution but also allows users to filter articles by their preferred news sources.
3. **Temporal data**: Both the original publication date and our scraping date are tracked, allowing for time-based filtering and helping identify stale content.
4. **Future expansion**: Including a sentiment field laid groundwork for potential future features like sentiment analysis and mood-based filtering.

#### 3.5.3.2 Stock Price Item Design

The StockPriceItem needed to capture all essential price and trading data:



The design challenges for this item included:

1. **Data completeness**: I needed to ensure we captured enough data points to provide meaningful insights without overloading the system with unnecessary information.
2. **Historical context**: By including both the raw price and the change metrics, I made it possible to understand the price in the context of recent movements.
3. **Flexible source attribution**: The source field allows for potential future expansion to multiple financial data providers.

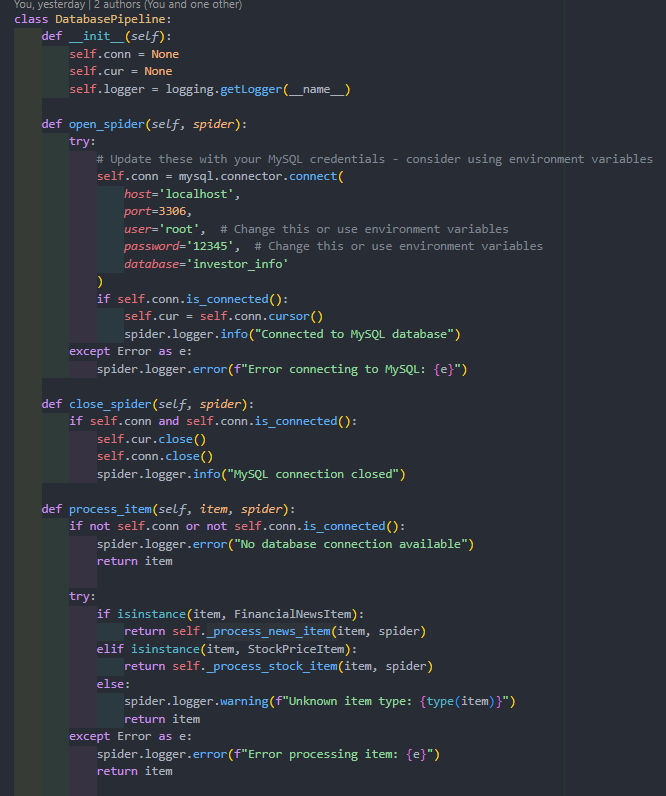
One challenge I faced was deciding whether to include additional financial metrics like P/E ratio, dividend yield, or 52-week ranges. I ultimately decided to keep the item focused on the most essential price data to maintain simplicity and performance, with the option to expand metrics in future versions.

### 3.5.4 Creating Database Pipeline

The database pipeline is a critical component that bridges the gap between the scraped data (Scrapy items) and permanent storage (MariaDB database). I implemented a robust pipeline that handles data validation, duplicate detection, and efficient database operations.

#### 3.5.4.1 Pipeline Architecture

The pipeline follows a modular architecture with specialized processing for different types of items:

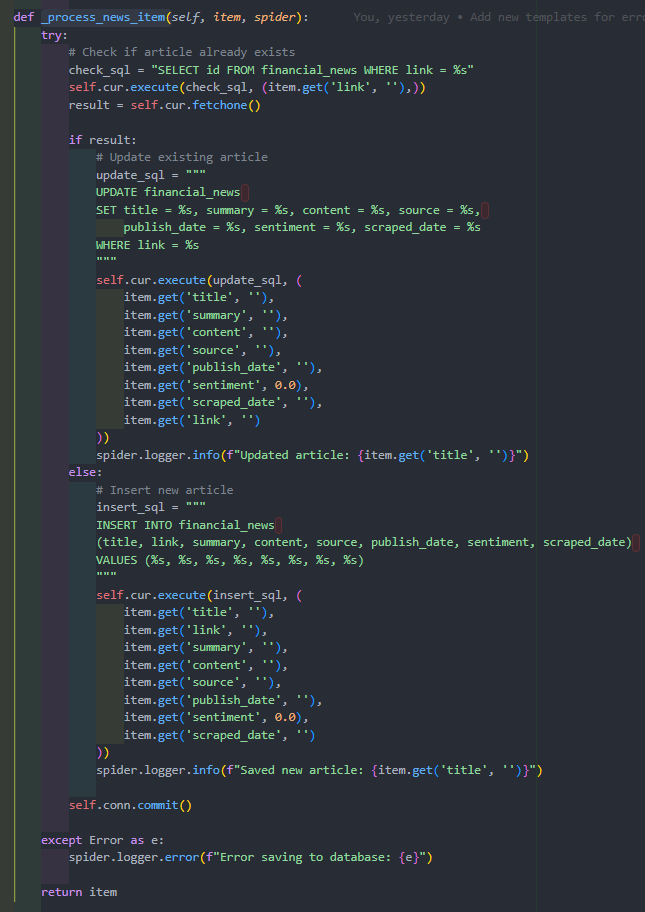
The lifecycle management was carefully designed:

* The open\_spider method establishes a database connection when the spider starts
* The process\_item method routes items to specialized processing methods based on their type
* The close\_spider method gracefully closes the database connection when the spider finishes

One of the challenges I faced was handling database connection failures. I implemented robust error handling to ensure the spider could continue operating even if database operations failed, with detailed logging to facilitate debugging.

#### 3.5.4.2 News Article Processing Logic

The news article processing logic included sophisticated duplicate detection and handling:

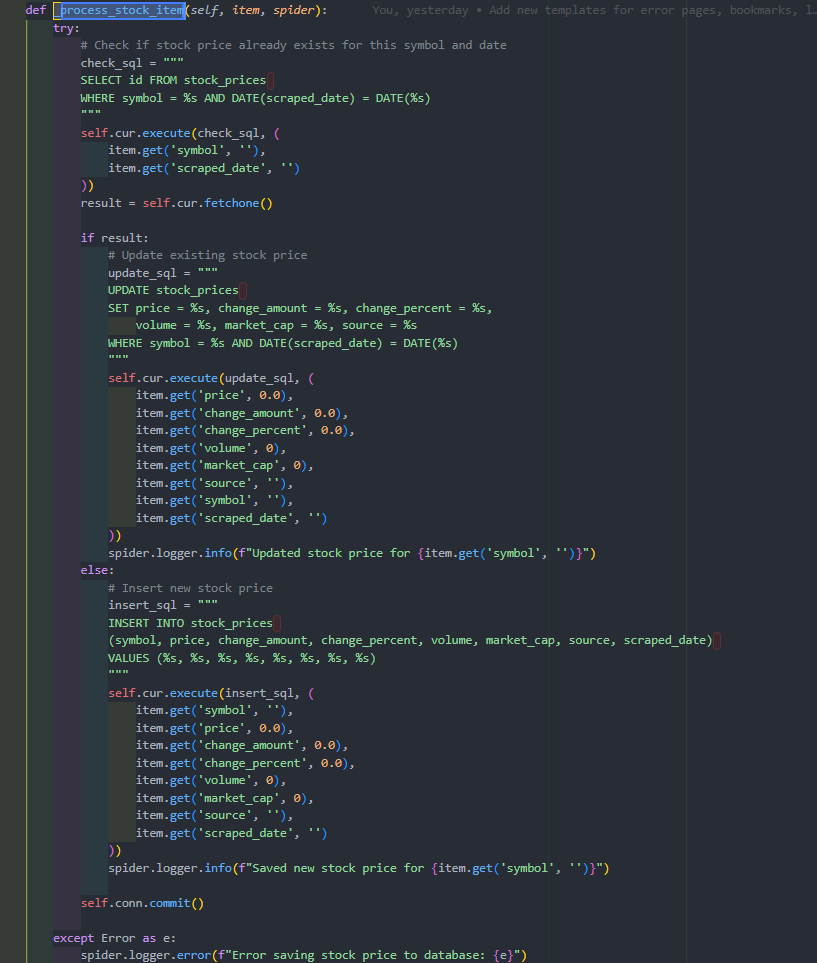
Key features of this implementation include:

1. **Efficient duplicate detection**: Rather than comparing all fields, I used the article URL as a unique identifier to quickly detect duplicates.
2. **Update vs. Insert logic**: When a duplicate is found, the system updates the existing record with fresh data rather than creating a new one, ensuring that articles stay current without creating duplicates.
3. **Parameterized queries**: All SQL queries use parameterized statements to prevent SQL injection vulnerabilities.
4. **Comprehensive logging**: Each operation logs detailed information for monitoring and debugging.

A significant challenge I faced was handling articles that had been updated by their source. My solution was to always update existing articles with fresh data, ensuring that the system captured any changes made to articles after initial publication.

**3.5.4.3 Stock Price Processing Logic**

The stock price processing used a different approach for uniqueness checks, based on both symbol and date:

 The uniqueness strategy for stock prices presented an interesting challenge. I needed to balance two requirements:

1. Maintaining historical price data for trend analysis
2. Avoiding duplicate entries for the same stock on the same day

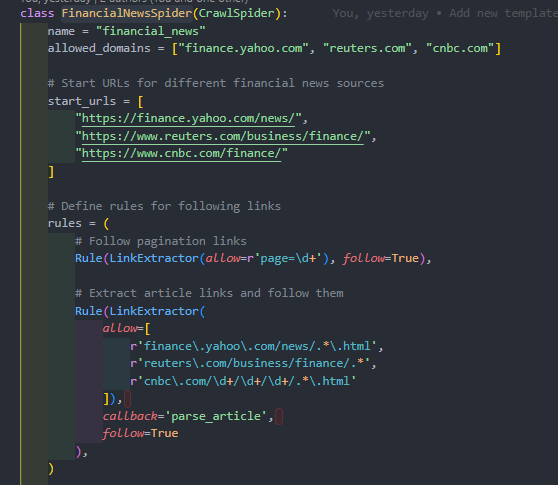
My solution was to use both the stock symbol and the date (without time) as a composite key for uniqueness. This allows multiple price snapshots per day to update rather than create new records, while still preserving day-to-day historical data.

### 3.5.5 Creating Financial News Crawler

The financial news crawler is responsible for discovering, extracting, and processing news articles from multiple financial news sources. This component required sophisticated link extraction, content parsing, and error handling capabilities.

#### 3.5.5.1 Spider Architecture and Configuration

The spider was implemented as a CrawlSpider, which provides powerful link following capabilities:



This architecture involved several key design decisions:

1. **Multiple start points**: By starting from the news sections of major financial sites, I ensured broad coverage of financial news.
2. **Targeted link extraction**: The use of regular expressions in the LinkExtractor rules helped focus the crawler on only relevant content, avoiding irrelevant pages.
3. **Pagination handling**: A dedicated rule for pagination links ensured the crawler could discover articles beyond just the first page.

A major challenge I faced was the different URL structures used by various financial news sites. I addressed this by analyzing each site's structure and crafting specific regular expressions for each one.

#### 3.5.5.2 Source Detection and Parsing Strategy

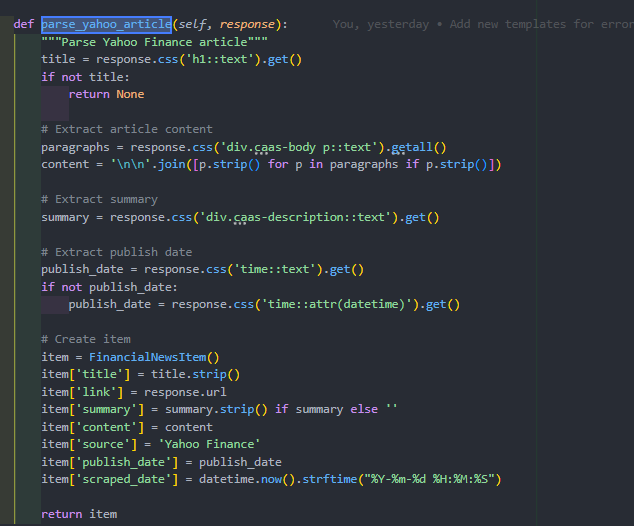
To handle the different HTML structures of various news sites, I implemented a source-based parsing strategy:

parse\_article This design pattern offered several advantages:

1. **Modularity**: Each source has its own parsing method, making the code easier to maintain.
2. **Adaptability**: When a source changes its HTML structure (which happens frequently with news sites), only the specific parsing method needs to be updated.
3. **Extensibility**: Adding support for new sources only requires adding a new parsing method without changing the overall architecture.

#### 3.5.5.3 Source-Specific Parsing Logic

Each source required custom parsing logic tailored to its unique HTML structure. Here's the Yahoo Finance parser as an example:

 One of the most significant challenges in the parsing logic was handling inconsistent HTML structures. Even within the same source, different articles might use slightly different HTML patterns. I addressed this with several techniques:

1. **Fallback logic**: For critical fields like publication date, I implemented multiple extraction methods that would be tried in sequence.
2. **Content cleaning**: I implemented filtering to remove empty paragraphs and whitespace to ensure clean, readable content.
3. **Default values**: For optional fields, I provided sensible defaults to ensure data consistency.

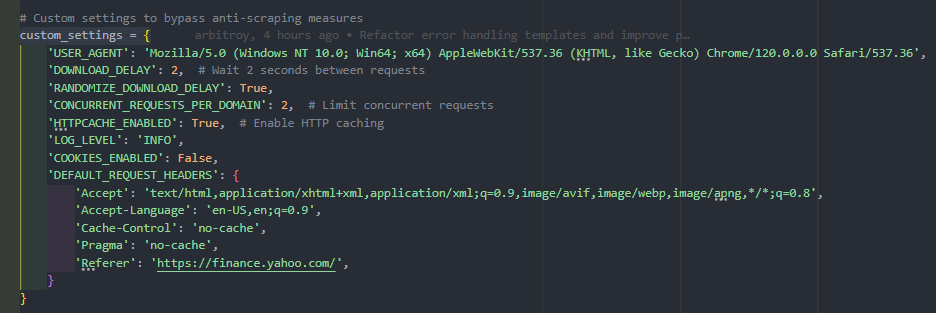
The parsers also needed to be resilient to changes in the source websites' HTML structure. To achieve this, I used CSS selectors that targeted structural elements rather than specific classes or ids where possible, as these tend to be more stable across site updates.

### 3.5.6 Creating Stock Prices Crawler

The stock prices crawler was designed to extract real-time and historical stock data from financial APIs. Unlike the news crawler, which scrapes HTML content, this crawler interacts with structured APIs that provide market data.

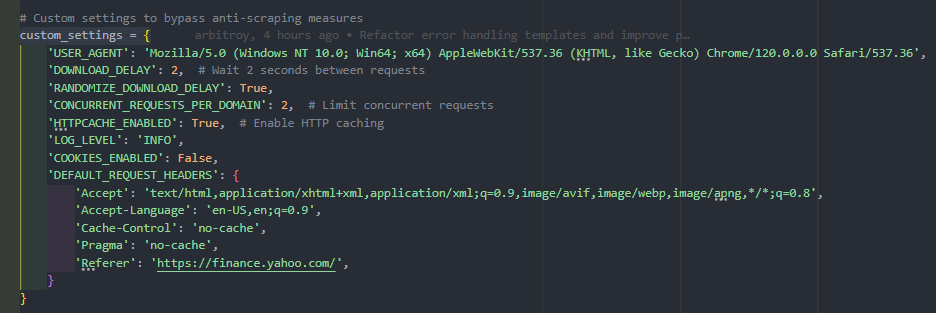
#### 3.5.6.1 Target Selection and Configuration

The first step was determining which stocks to track. I focused on major tech stocks and market indices:

I faced a difficult decision regarding how many stocks to track. Each additional stock increases the system's data richness but also adds to the computational and storage load. I settled on a mix of:

1. **Major tech stocks**: These are popular with retail investors and tend to be newsworthy
2. **Diverse sector leaders**: Stocks from different sectors provide a broader market view
3. **Global market indices**: These give users a quick overview of market conditions

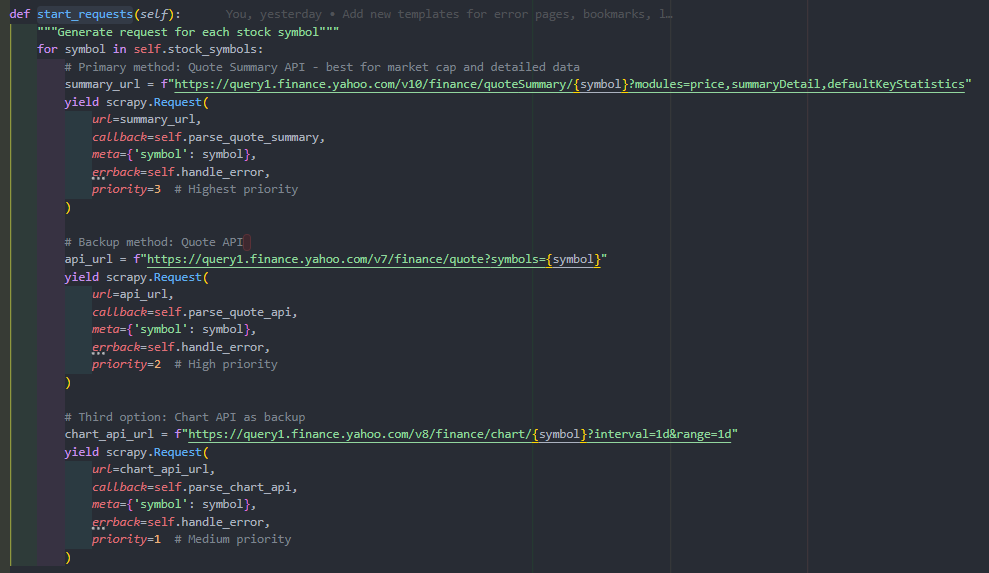
Another key configuration decision was to use enhanced anti-blocking measures:

Financial data providers typically have stricter anti-scraping measures than news sites. To address this, I implemented:

1. **Realistic browser headers**: The detailed request headers mimic a real browser precisely
2. **Referer spoofing**: Setting the Referer header to make requests appear to come from Yahoo Finance
3. **Reduced concurrency**: Limiting to just 2 concurrent requests per domain to stay well under rate limits

#### 3.5.6.2 Multi-API Request Strategy

A key innovation in this crawler was the implementation of a redundant API request strategy. Financial APIs can be unreliable, with some endpoints occasionally returning incomplete data or errors. To ensure reliable data collection, I implemented a multi-API approach:

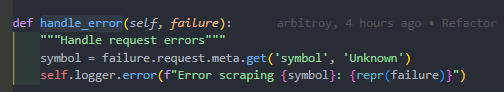
 This multi-API approach represents one of the most innovative aspects of the crawler design. For each stock symbol, I generate three separate requests to different Yahoo Finance API endpoints, each with a different priority level. This provides several key benefits:

1. **Fallback data sources**: If the primary API fails or returns incomplete data, the system can fall back to the secondary or tertiary sources.
2. **Data verification**: Data from multiple sources can be cross-checked for consistency, enhancing accuracy.
3. **Priority-based processing**: The request priority system ensures that primary data sources are processed first, with fallbacks only used when necessary.

This strategy significantly improved data reliability, addressing a common issue with financial APIs where certain fields might be missing or unavailable from a single endpoint.

#### 3.5.6.3 Error Handling and Recovery

Robust error handling was essential for maintaining crawler operation even when encountering API issues:



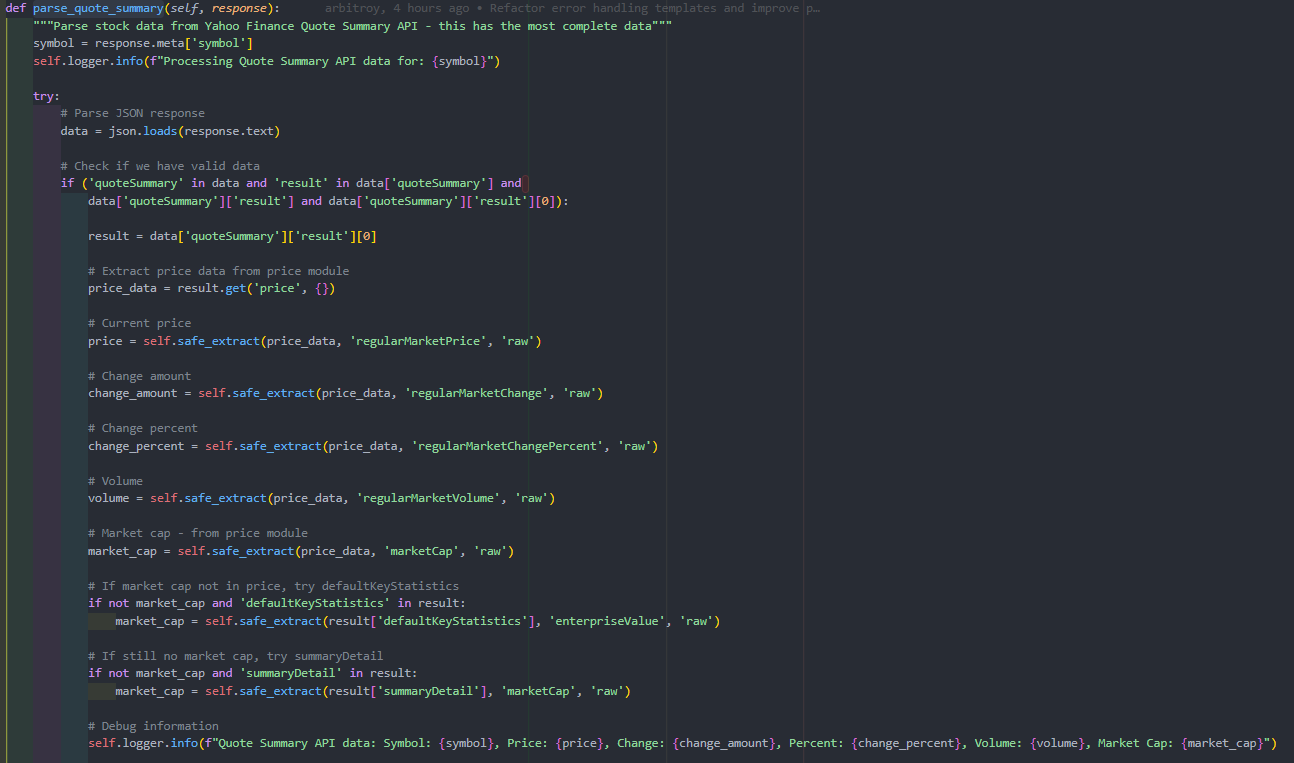
The error handler captures detailed information about failures, including:

* The stock symbol being processed
* The HTTP status code if available
* Response headers that might indicate rate limiting or blocking
* A snippet of the response body that often contains error messages

This detailed error information proved invaluable for debugging API issues and adjusting the crawler's behavior to avoid problems in future runs.

#### 3.5.6.4 Advanced Data Extraction and Processing

The Quote Summary API parsing method illustrates the sophisticated data extraction and handling techniques implemented:

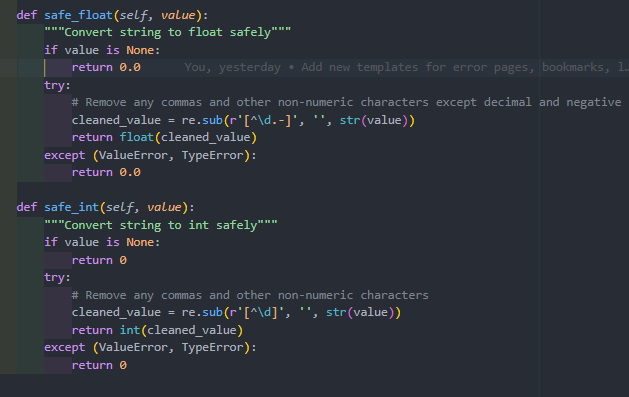
 This parsing method incorporates several advanced techniques:

1. **Nested data extraction**: The Yahoo Finance API returns deeply nested JSON data structures. I implemented the safe\_extract helper method to safely navigate these structures.
2. **Fallback field sources**: Market capitalization might be found in different parts of the response. The method tries multiple locations before giving up.
3. **Type conversion with error handling**: All extracted values are converted to the appropriate data types with fallback to default values if conversion fails.
4. **Validation before processing**: The method only creates an item if at least the essential price field is available, preventing incomplete data from entering the system.

safe\_extract The safe\_extract method demonstrates a defensive programming approach, ensuring that the system can handle incomplete or unexpected data structures without crashing. This was particularly important when dealing with financial APIs, which might occasionally return partial data or change their response structure.

#### 3.5.6.5 Data Cleaning and Normalization

Financial data often requires cleaning and normalization before it can be stored consistently:

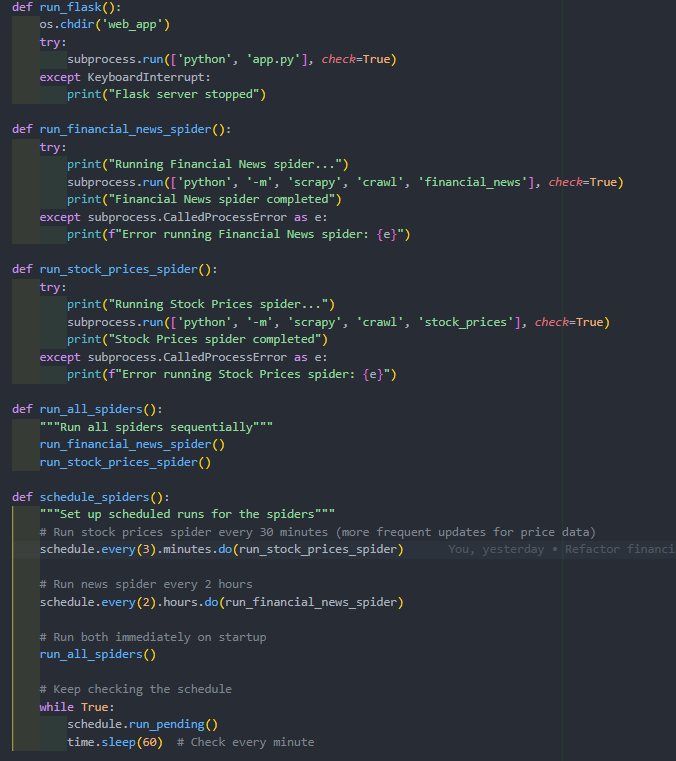
 These helper methods address common issues with financial data:

1. **Format inconsistency**: Numbers might be presented with commas (e.g., "1,234.56") or other formatting that needs to be stripped before conversion.
2. **Type inconsistency**: Values might come as strings, numbers, or null/None, requiring careful handling during conversion.
3. **Invalid values**: Some fields might contain "N/A" or other non-numeric placeholders that need to be converted to sensible defaults.

The regular expression pattern r'[^\d.-]' specifically targets and removes any character that is not a digit, decimal point, or negative sign, ensuring that numeric values are properly extracted from formatted strings.

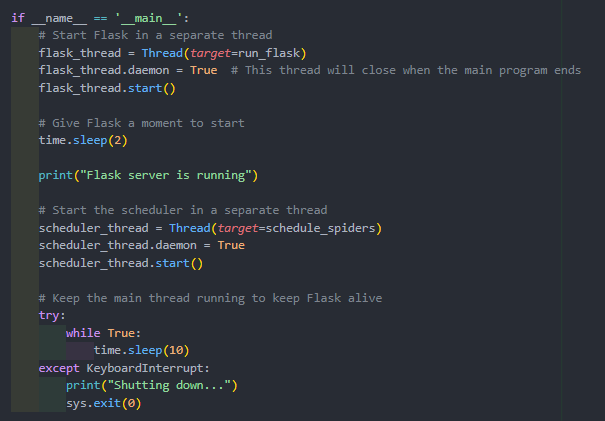
### 3.5.7 Integration and Scheduler Implementation

The crawlers needed to run on a schedule to keep the database updated with fresh data. I implemented a sophisticated scheduling system that integrates with the Flask application:

This scheduling system incorporates several key features:

1. **Differential update frequencies**: Stock prices are updated more frequently (every 30 minutes) than news articles (every 2 hours), reflecting the different rates at which this information changes.
2. **Immediate initial run**: Both spiders run immediately when the system starts to ensure fresh data is available.
3. **Subprocess execution**: The spiders run as separate processes to prevent them from blocking the main application.
4. **Error isolation**: Errors in one spider do not affect the other or the main application.

The main application starter then brings everything together:

This main script represents the heart of the integration between the Flask web application and the Scrapy crawlers. It uses Python's threading to run multiple components concurrently:

1. **Flask thread**: Runs the web application that users interact with
2. **Scheduler thread**: Manages the periodic execution of the crawler processes
3. **Main thread**: Keeps the application running and handles graceful shutdown

One of the most challenging aspects of this integration was ensuring that the separate components (Flask and Scrapy) could work together smoothly despite being designed as standalone applications. The threading approach with subprocess execution for the crawlers provided a clean solution, allowing each component to operate independently while sharing the database.

The daemon flag on the threads ensures that they automatically terminate when the main program exits, preventing orphaned processes. The KeyboardInterrupt handling provides a graceful shutdown mechanism, which is important for properly closing database connections and other resources.

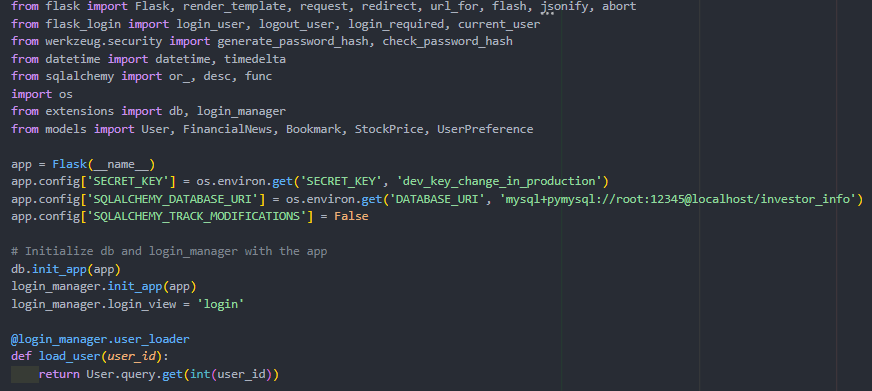
This integrated system architecture enables a seamless user experience where data is continuously updated in the background without interrupting or slowing down the web application.

### 3.5.8 Developing Web Application

After developing the data collection infrastructure, I needed to create a user-facing web application that would present this financial information in an intuitive and engaging way. For this task, I chose Flask - a lightweight but powerful Python web framework that would allow me to rapidly develop the application while maintaining full control over its architecture.

#### 3.5.8.1 Flask Application Setup and Configuration

Setting up the Flask application involved several critical configuration decisions to ensure security, performance, and maintainability. Here's how I configured the core Flask application:

 This configuration incorporates several important security and architectural decisions:

1. **Environment-based configuration**: Using os.environ.get() allows different configuration settings in development and production environments without changing code.
2. **Secret key management**: The secret key used for session signing has a fallback value for development but would be set through environment variables in production.
3. **Database connection string**: Similarly, the database connection is configurable through environment variables, making the application portable across different environments.
4. **Extension initialization**: I separated the extensions (database and login manager) into a dedicated module to avoid circular imports, which is a common issue in Flask applications.
5. **Login view specification**: Setting login\_manager.login\_view = 'login' ensures that unauthorized users are redirected to the login page when they try to access protected routes.

One of the challenging decisions was whether to use a full-featured framework like Django or the more lightweight Flask. I ultimately chose Flask because:

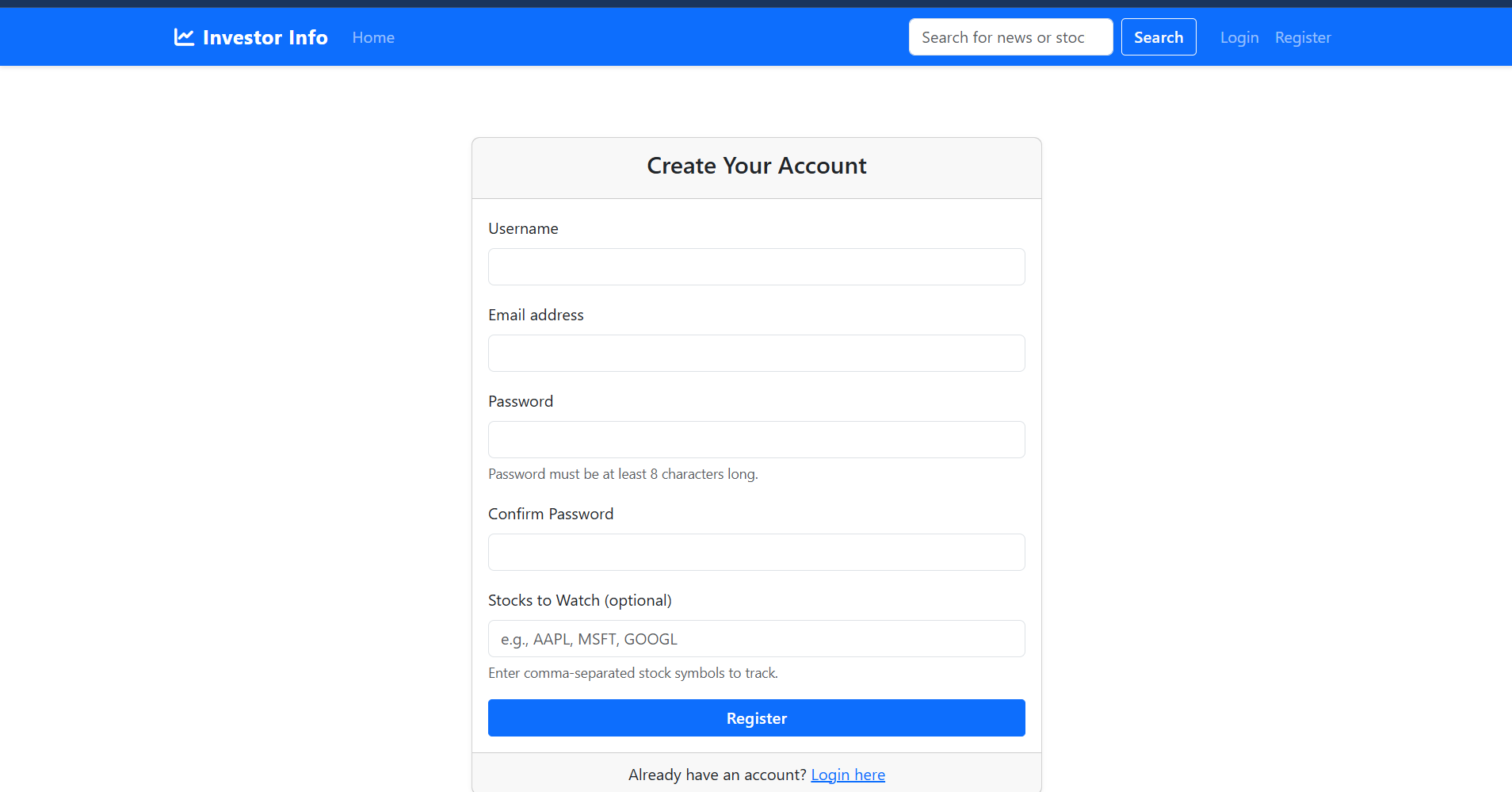
* It provided just enough structure without excessive boilerplate
* It integrated well with SQLAlchemy for database operations
* It allowed me to incrementally add features as needed
* Its extension ecosystem provided solutions for authentication, form handling, and other common tasks

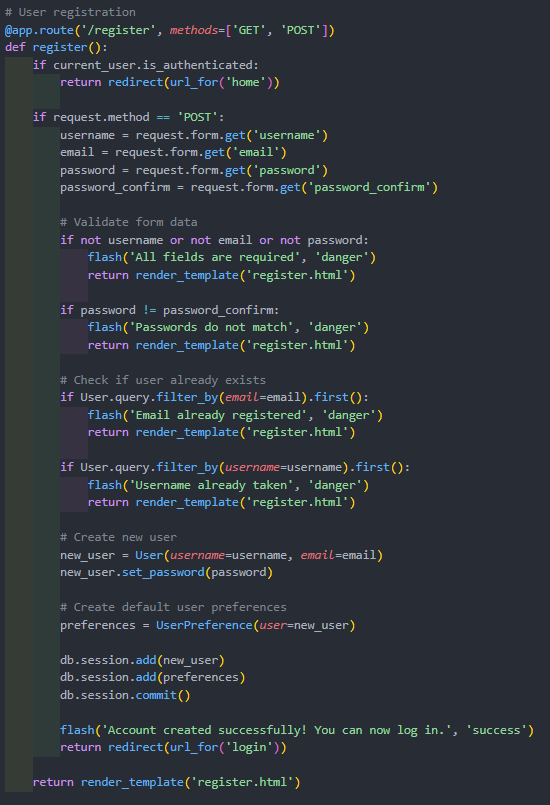
#### 3.5.8.2 User Authentication System

The user authentication system includes registration, login, and profile management features to provide a personalized experience. I implemented this using Flask-Login, which provides session management and the "remember me" functionality.

**User Registration Implementation**

The registration route handles new user account creation with validation and secure password storage:





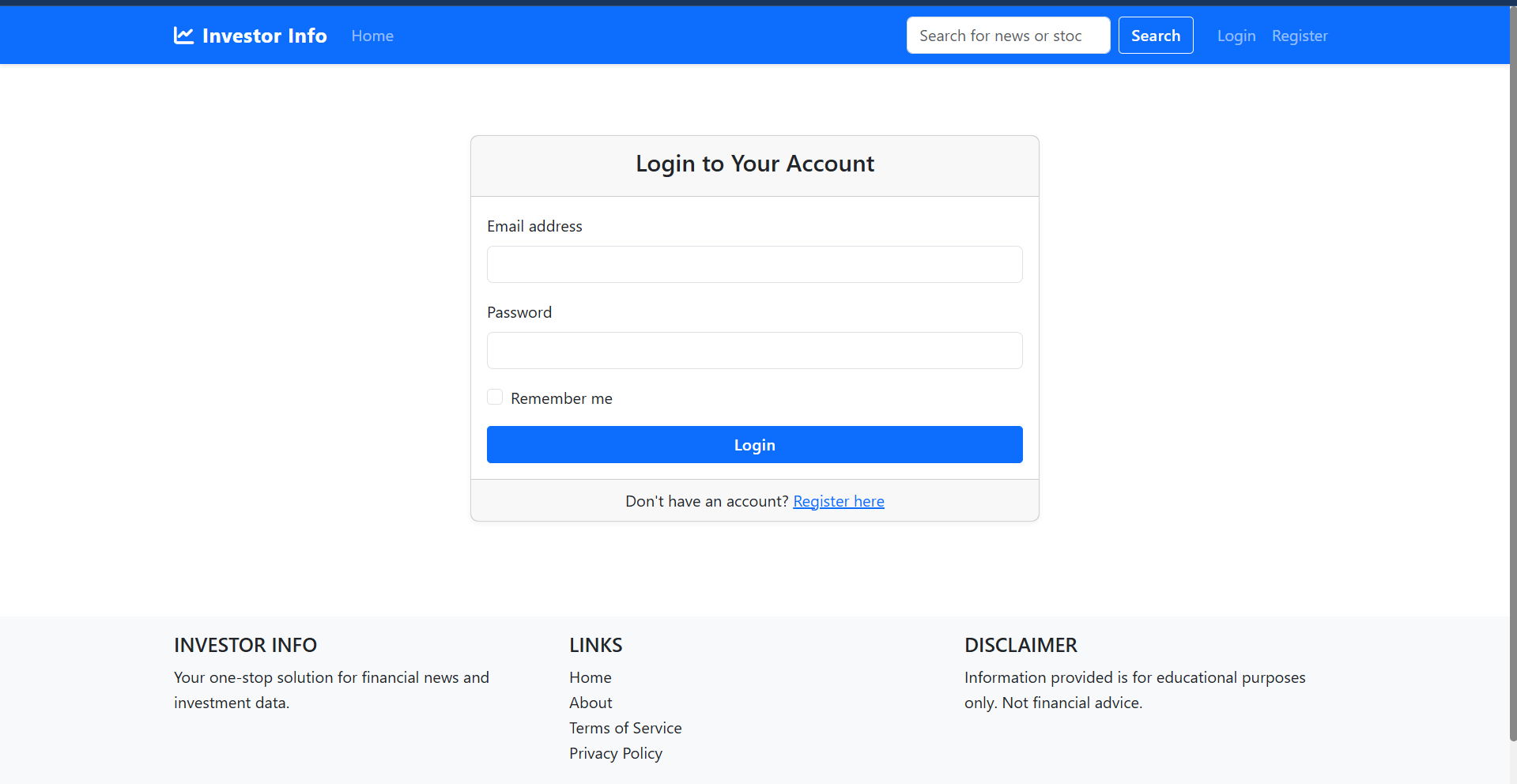
The registration process includes several key security and user experience features:

1. **Input validation**: Comprehensive validation ensures all required fields are provided and passwords match.
2. **Uniqueness checks**: The system verifies that the email and username aren't already in use before creating the account.
3. **Secure password handling**: Passwords are never stored in plaintext. Instead, the set\_password method uses Werkzeug's secure hashing with the PBKDF2 algorithm.
4. **User preferences initialization**: Each new user automatically gets a preferences record, ensuring that preference-related features work even for new users.
5. **User feedback**: Flash messages provide immediate feedback about the registration process.

A challenging aspect of the registration process was determining what initial preferences to set for new users. I decided to create an empty preferences record with defaults (like the light theme) to ensure a consistent experience while allowing users to customize later.

**Login System Implementation**

The login route handles authentication with "remember me" functionality:





The login system includes several security and usability features:

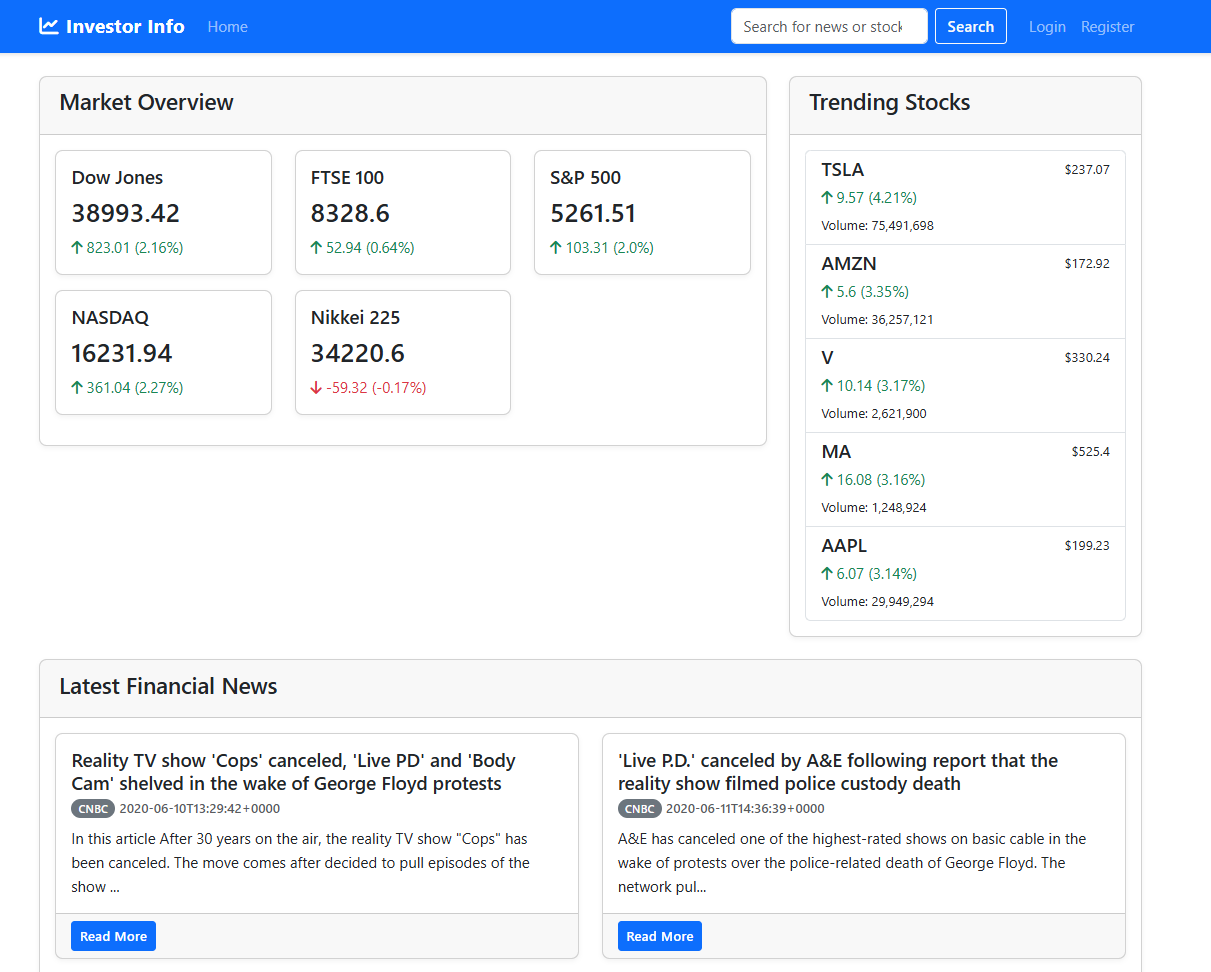
1. **Remember me functionality**: Users can choose whether to remain logged in after closing their browser.
2. **Last login tracking**: The system records when users log in, which could be used for analytics or security monitoring.
3. **Next page redirection**: If users were redirected to the login page from a protected route, they're sent back to that route after successful authentication.
4. **Generic error messages**: The error message doesn't specify whether the email or password was incorrect, making it harder for attackers to enumerate valid emails.

A particularly challenging aspect of the authentication system was balancing security with usability. I opted for email-based login rather than username-based login because:

* Email addresses are unique and users generally remember them
* It reduces the number of credentials users need to remember
* It allows for password reset functionality in future versions

#### 3.5.8.3 Home Page Implementation

The home page serves as the application's dashboard, displaying market overviews, trending stocks, and latest financial news. This required integrating multiple data sources into a coherent user interface:



The home page route demonstrates several interesting data processing techniques:

1. **Multi-data-source integration**: The route combines data from different database tables (financial news and stock prices) into a cohesive view.
2. **Dynamic content prioritization**: News articles are sorted by recency, while stocks are sorted by volatility (absolute percentage change).
3. **Data pruning**: Rather than overwhelming users with all available data, the route selects only the most recent news and most volatile stocks.
4. **Predefined major indexes**: The route hardcodes the symbols for major market indexes to ensure they're always displayed, regardless of their recent performance.

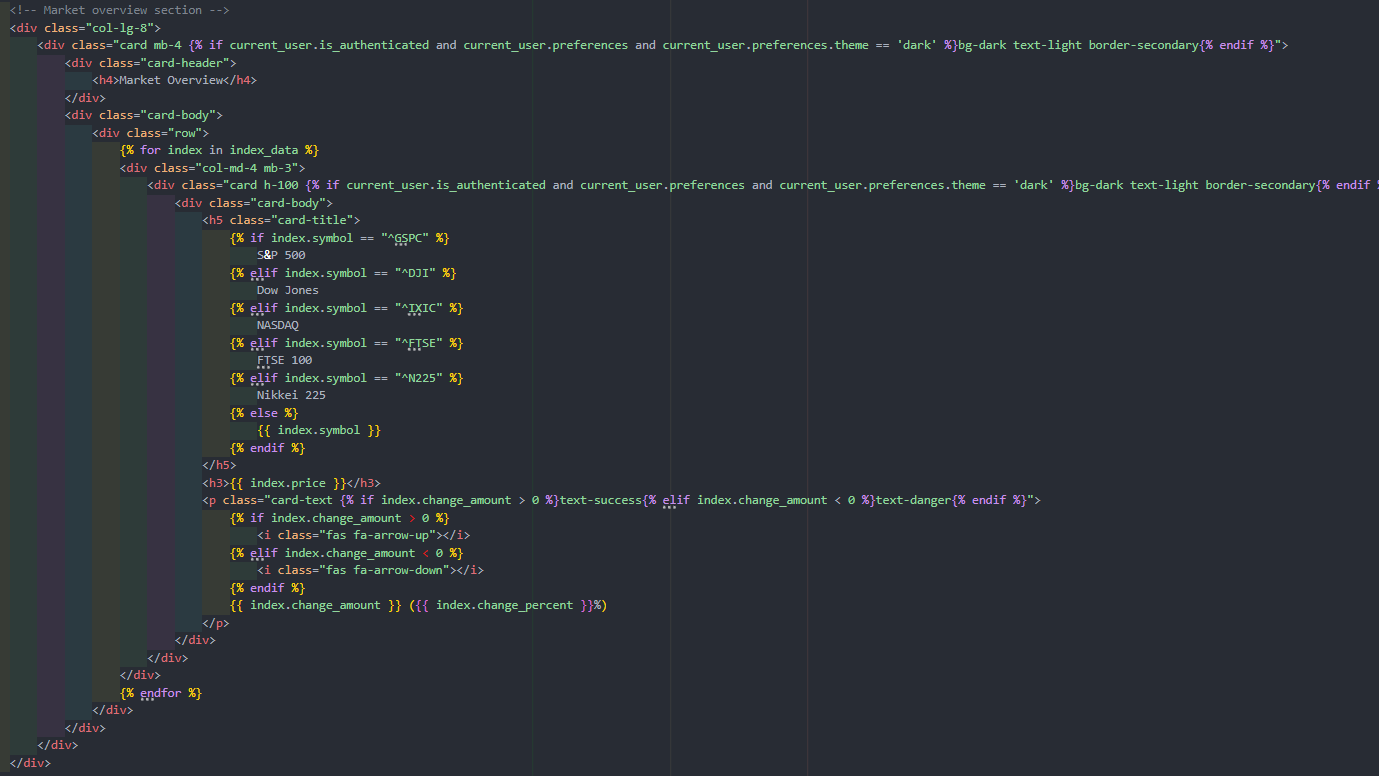
A considerable design challenge for the home page was determining what information should be displayed by default. I decided to highlight:

* **Major market indexes**: To give users a quick overview of global market conditions
* **Trending stocks**: To highlight potentially newsworthy price movements
* **Recent financial news**: To provide context for market movements

This approach provides a balance of quantitative data (prices and percentages) and qualitative information (news articles) that helps users understand not just what is happening in the markets, but potentially why it's happening.

**Home Page Template Design**

The home page template renders the data in an intuitive dashboard layout:

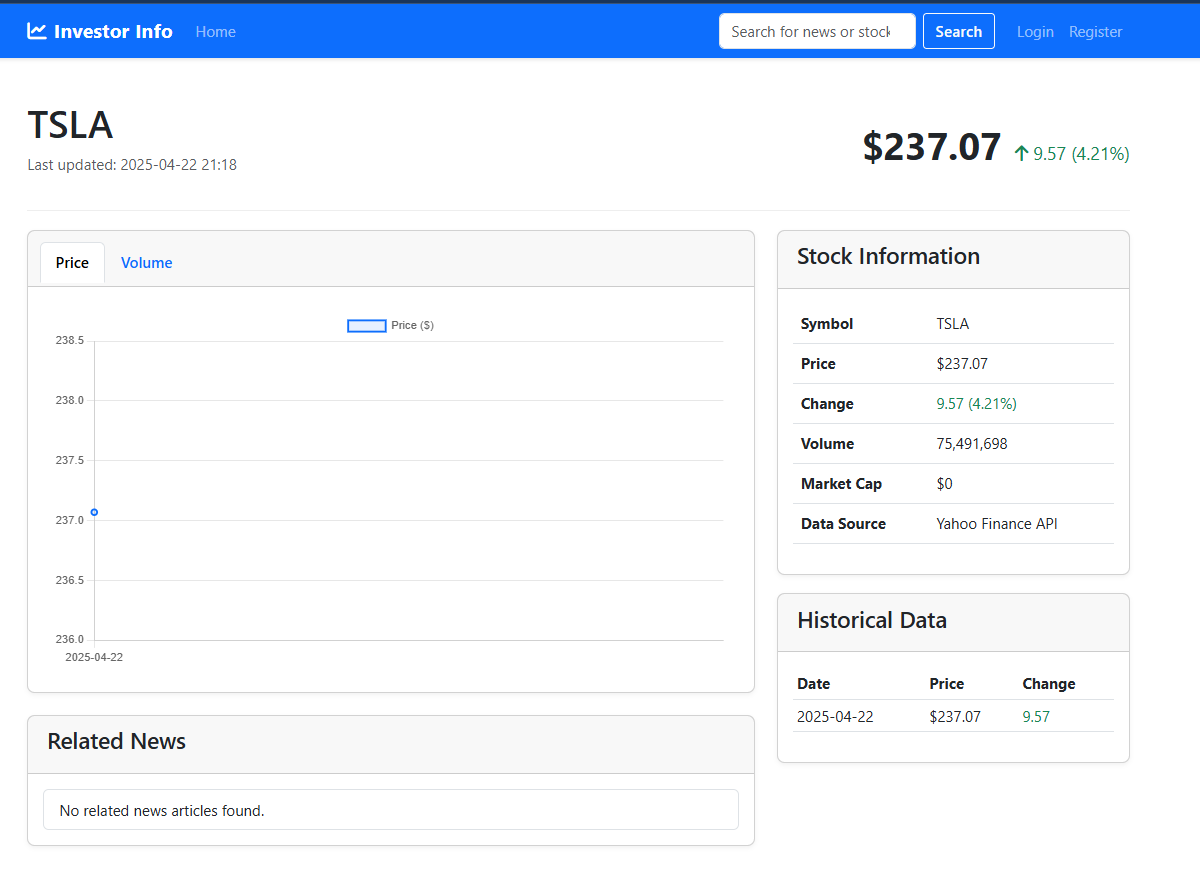
The template includes several user experience enhancements:

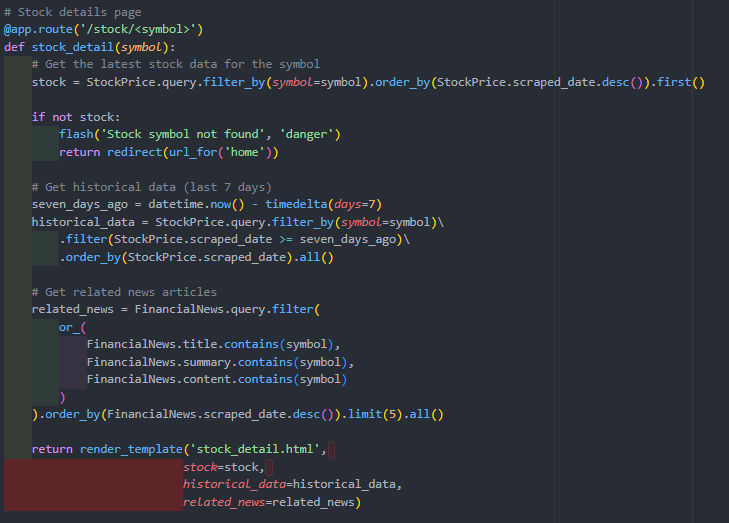
1. **Human-readable index names**: Technical symbols like "^GSPC" are translated into familiar names like "S&P 500".
2. **Visual indicators**: Positive changes are highlighted in green with an up arrow, while negative changes are in red with a down arrow.
3. **Responsive layout**: The Bootstrap grid system ensures the layout works well on both desktop and mobile devices.
4. **Theme-aware styling**: Conditional classes adjust the styling based on the user's theme preference.

This approach to visualization helps users quickly interpret the data without needing to parse raw numbers and symbols.

#### 3.5.8.4 Stock Detail Page Implementation

The stock detail page provides comprehensive information about a specific stock, including current price, historical data, and related news articles:



This route demonstrates several advanced data retrieval techniques:

1. **Latest record retrieval**: The query for the stock data uses order\_by(StockPrice.scraped\_date.desc()).first() to get the most recent record.
2. **Date-range filtering**: The historical data query uses the timedelta class to fetch only the last 7 days of data.
3. **Full-text search**: The related news query uses SQLAlchemy's or\_ operator to search for the stock symbol across multiple text fields.
4. **Error handling**: If the stock symbol isn't found, the user is redirected to the home page with an informative flash message.

A significant challenge in implementing this page was deciding how to identify "related news" for a stock. I considered several approaches:

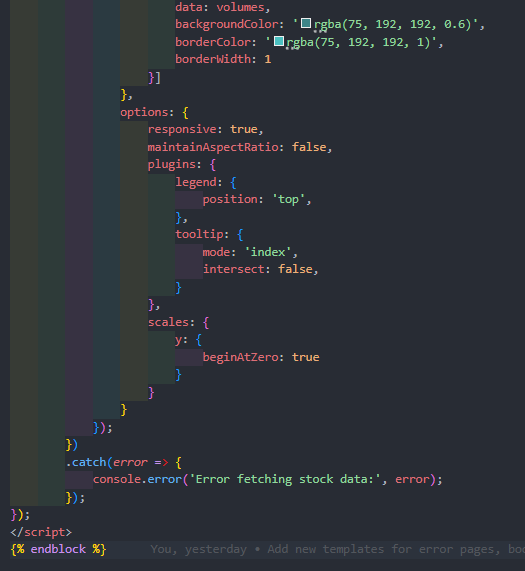
1. **Symbol matching**: Search for exact symbol matches in article text (e.g., "AAPL")
2. **Company name matching**: Search for company name mentions (e.g., "Apple")
3. **Industry matching**: Group articles by industry sector

I ultimately chose the symbol matching approach for its precision and simplicity, searching across the title, summary, and full content to capture all relevant mentions. This ensures that the related news is truly about the specific stock rather than tangentially related companies or industries.

**Stock Detail Template with Chart.js Integration**

The stock detail page includes interactive charts using Chart.js:



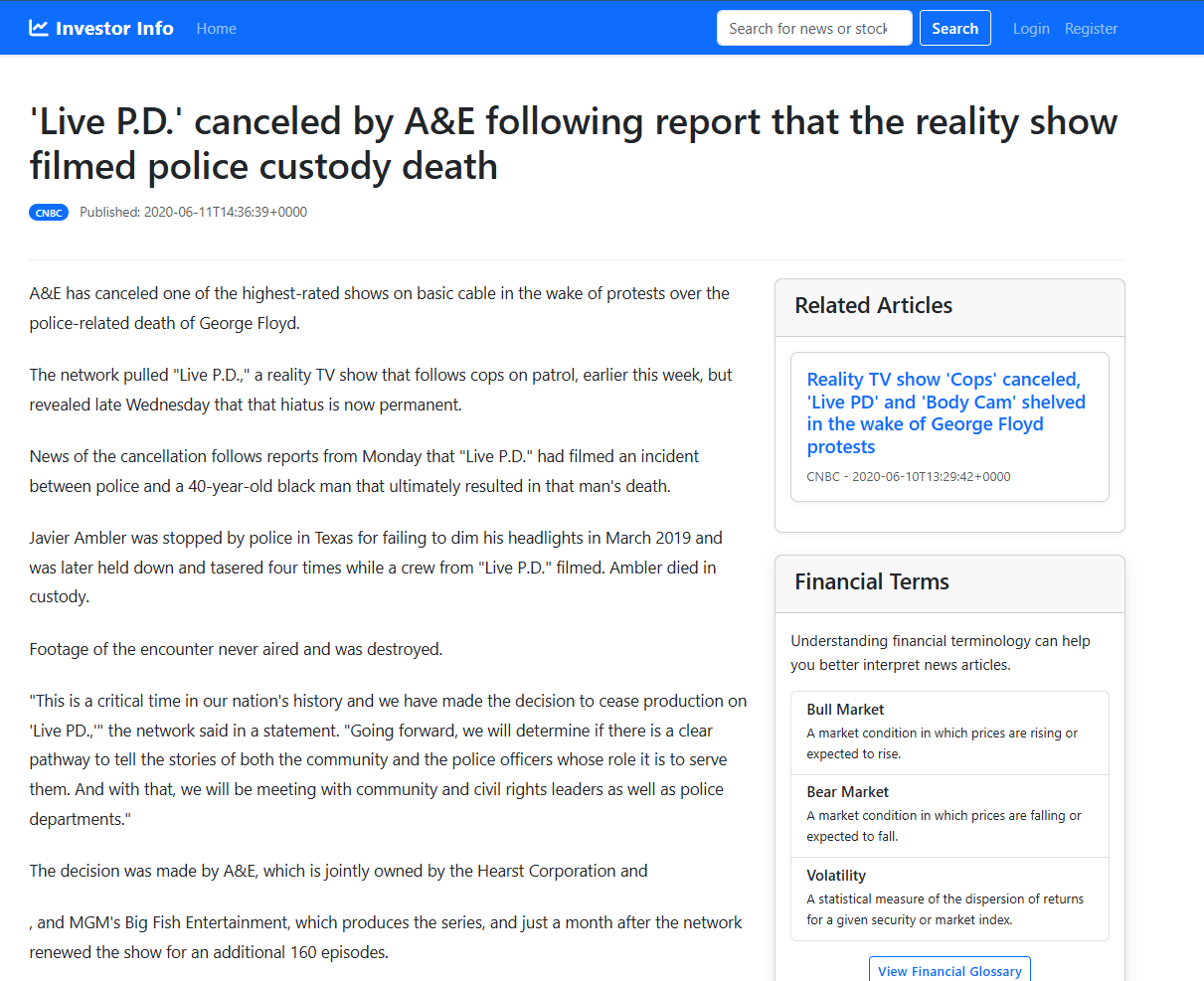
This JavaScript code demonstrates several advanced frontend techniques:

1. **API-driven data loading**: Rather than embedding all the historical data in the initial HTML, the code fetches it asynchronously through a dedicated API endpoint.
2. **Data transformation**: The raw API data is transformed into the format required by Chart.js.
3. **Multiple visualizations**: The same dataset is visualized in two different ways (line chart for price, bar chart for volume).
4. **Responsive design**: Both charts are configured to be responsive and maintain their aspect ratio.
5. **Interactive features**: The charts include tooltips that display precise values when hovering over data points.

This approach to data visualization provides a more engaging user experience than static tables or text descriptions, allowing users to quickly identify trends and patterns in the stock's performance.

#### 3.5.8.5 News Detail Page Implementation

The news detail page displays a complete news article along with related articles and sharing options:



This route demonstrates several interesting features:

1. **404 handling**: The get\_or\_404 method automatically returns a 404 response if the article isn't found.
2. **Bookmark status detection**: The route checks whether the current user has bookmarked the article.
3. **Related article identification**: The route implements a simple content similarity algorithm based on word overlap in titles.
4. **Pagination prevention**: The filter explicitly excludes the current article to prevent it from appearing in the related articles list.

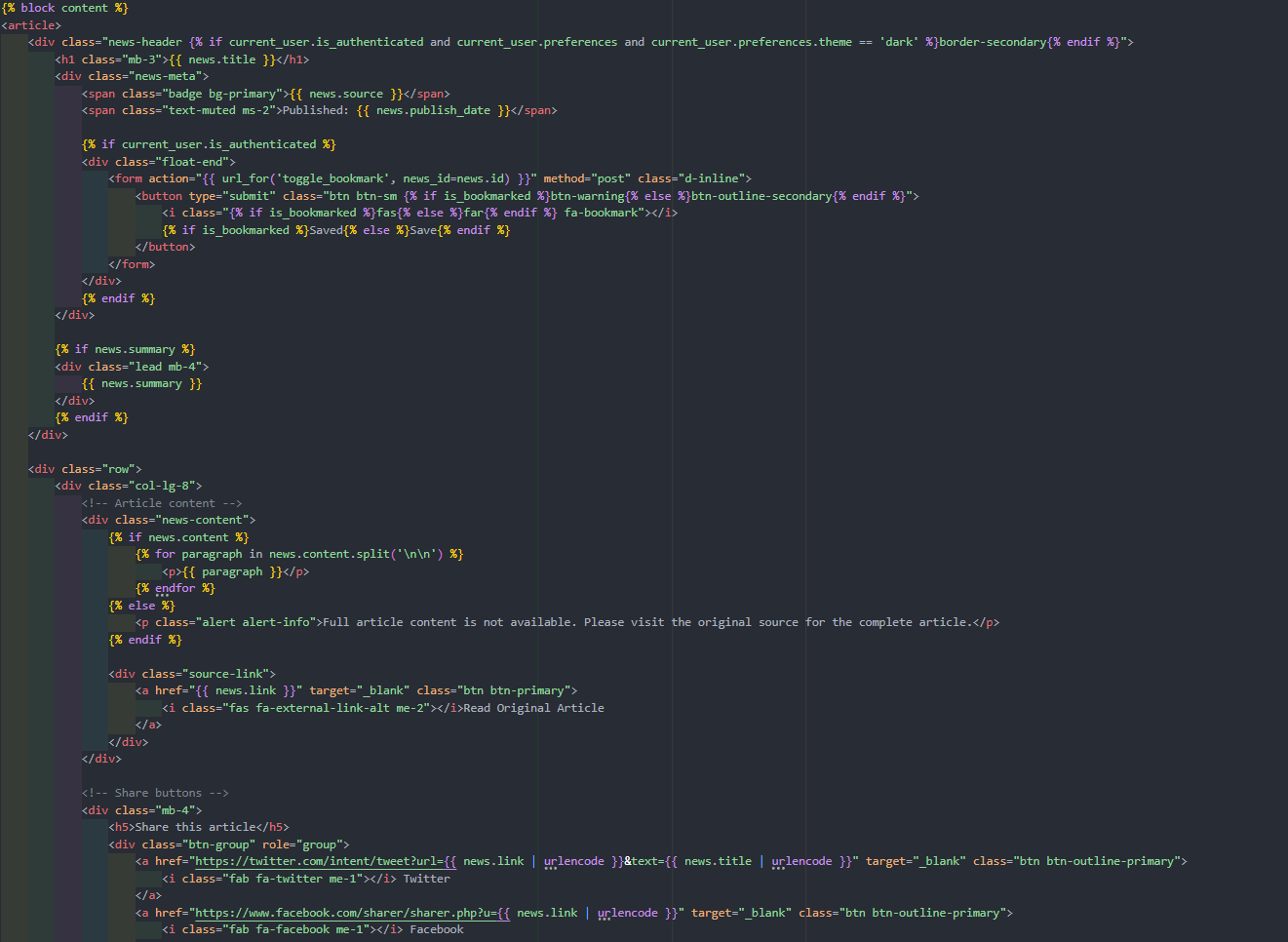
The most challenging aspect of this implementation was determining how to identify related articles. Without access to sophisticated natural language processing tools, I implemented a simple but effective word overlap approach:

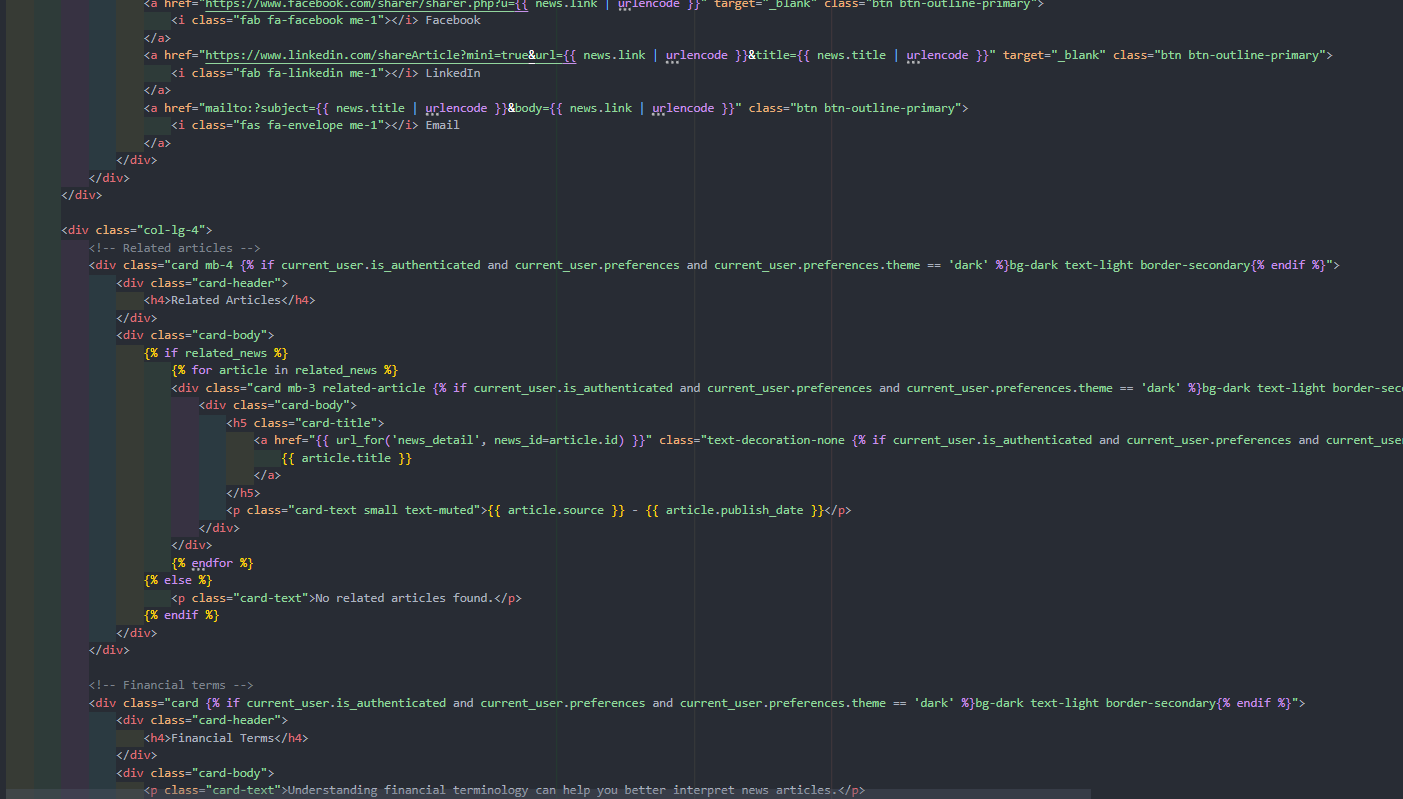
1. Split the current article's title into a set of words
2. For each potential related article, split its title into a set of words
3. Count the number of words that appear in both sets
4. Consider articles with at least 2 words in common to be related

This approach has some limitations (it doesn't consider synonyms or semantic relationships), but it works well for financial news where specific company names, financial terms, and numbers often indicate related content.

**News Detail Template with Formatting and Sharing Options**

The news detail template renders the article with proper formatting and social sharing options:



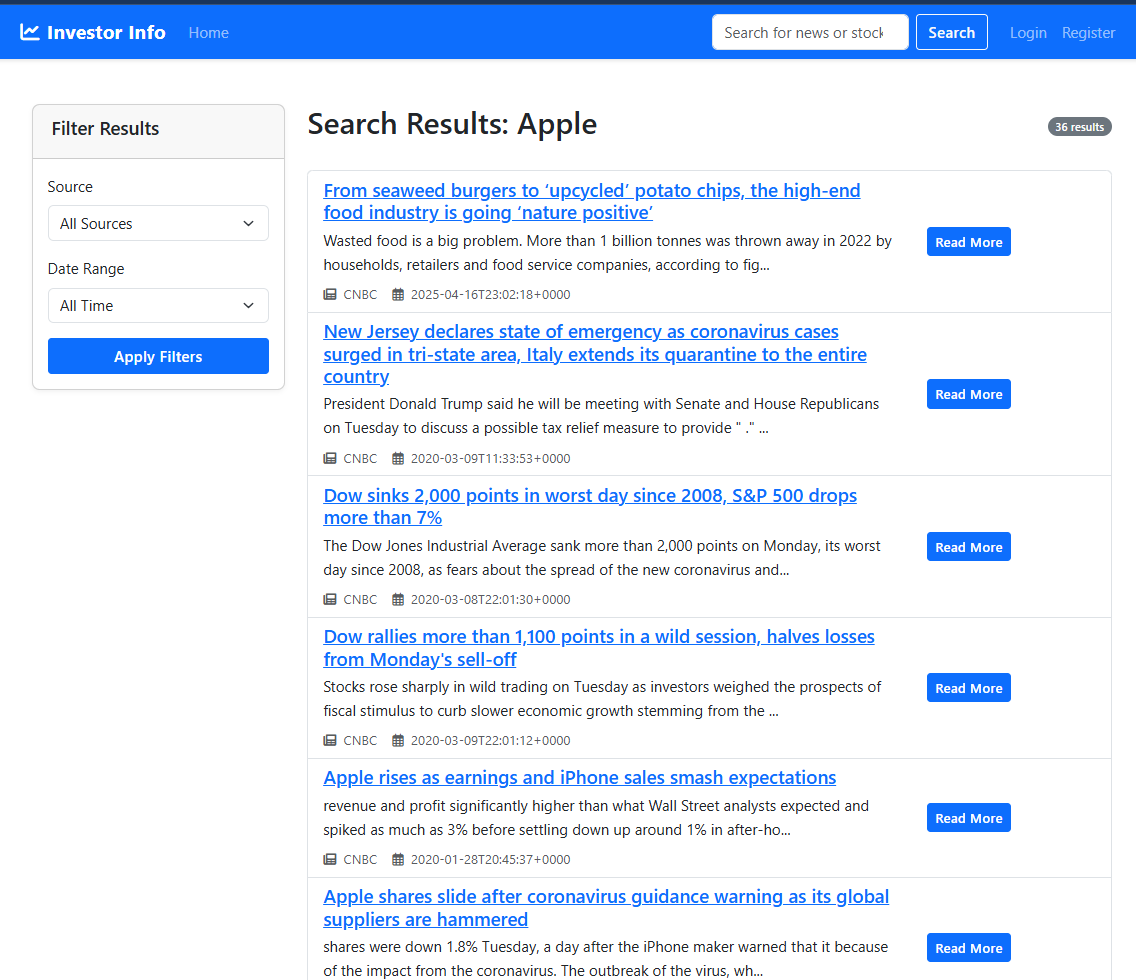
This template features several user experience enhancements:

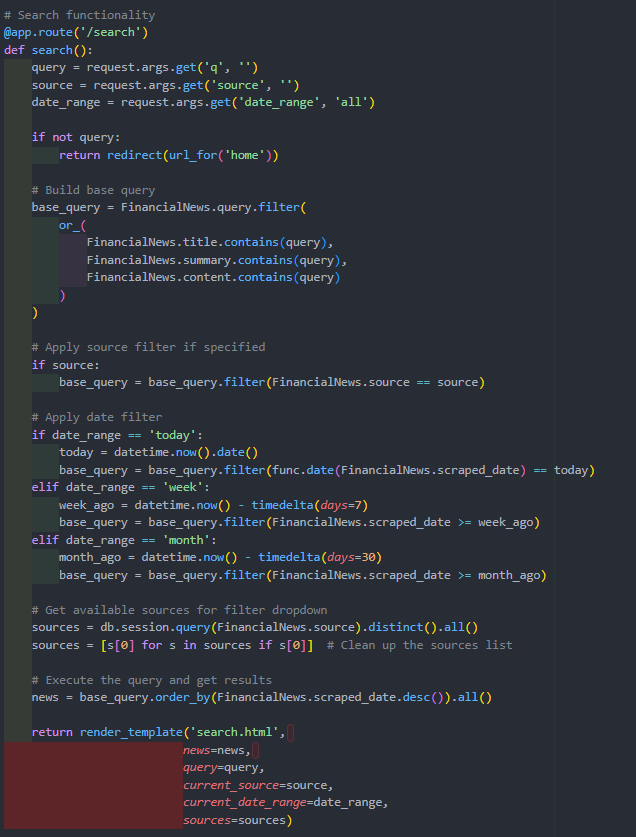
1. **Paragraph splitting**: The article content is split on double newlines and each part is wrapped in a paragraph tag for proper formatting.
2. **Fallback message**: If no content is available, a message directs users to the original source.
3. **Social sharing**: Buttons are provided for sharing the article on popular social platforms, with proper URL encoding.
4. **Related articles sidebar**: Related articles are displayed in a sidebar for easy discovery of additional content.
5. **Bookmark integration**: For logged-in users, a button allows them to save the article to their bookmarks with visual feedback on the current state.

A particular challenge in the template design was handling potentially inconsistent content formatting. Since the articles come from different sources, they might have different paragraph separators and formatting conventions. The template includes logic to handle these inconsistencies and present a consistent reading experience regardless of the source.

#### 3.5.8.6 Search Functionality Implementation

The search feature allows users to find news articles and stocks by keyword, with filtering options for source and publication date:



This search implementation demonstrates several advanced features:

1. **Multi-field search**: The query searches across title, summary, and full content fields.
2. **Filter options**: Users can filter results by source publication and date range.
3. **Dynamic filter options**: The available sources are queried from the database rather than hardcoded.
4. **Context extraction**: For each result, the system extracts sentences containing the search term for preview.
5. **Sort by relevance**: Results are sorted by publication date, showing the most recent articles first.

A particularly challenging aspect of the search implementation was extracting relevant sentences for previews. I implemented a custom sentence extraction algorithm that:

1. Splits the content into paragraphs
2. Splits each paragraph into sentences (using periods followed by space and capital letters as separators)
3. Filters to keep only sentences containing the search term
4. Adds these sentences to the article object for display in the template

This approach provides more relevant previews than simply showing the first few sentences of the article, as it highlights precisely where the search term appears in context.

# Chapter 4: Evaluation

The Investor Info application was evaluated using a comprehensive methodology that assessed both technical functionality and user experience.

## 4.1 Evaluation Methodology

The evaluation methodology consisted of several complementary approaches:

*Table 10: Evaluation Methods*

| **Test Type** | **Description** | **Purpose** |
| --- | --- | --- |
| Unit Testing | Testing individual components in isolation | Verify correct operation of specific functions |
| Integration Testing | Testing interactions between components | Ensure components work together correctly |
| Functional Testing | Manual testing of user-facing features | Verify system meets functional requirements |
| Usability Testing | Testing with representative users | Assess user experience and interface design |
| Performance Testing | Measuring system response times | Ensure acceptable performance under load |

## 4.2 Evaluation Metrics

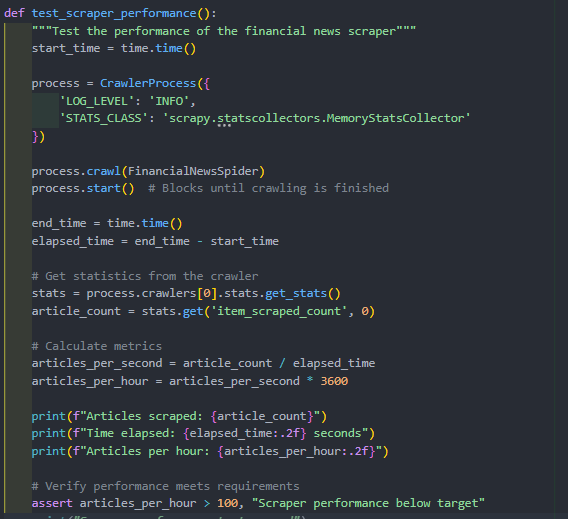
Specific metrics were defined to quantitatively evaluate the system's performance:

*Table 11: Evaluation Metrics*

| **Metric** | **Description** | **Target Value** |
| --- | --- | --- |
| Scraper Efficiency | Number of articles collected per hour | >100 articles/hour |
| Database Query Time | Time to retrieve search results | <200ms |
| Page Load Time | Time to render complete page | <1 second |
| Task Completion Rate | Percentage of users completing specified tasks | >90% |
| System Usability Scale | Standardized usability measurement | Score >70 |
| Data Accuracy | Percentage of articles correctly categorized | >95% |

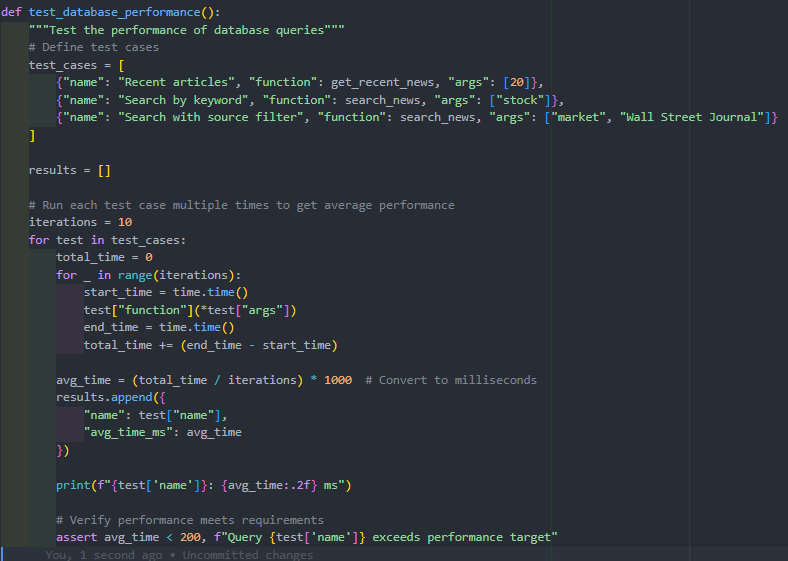
## 4.3 Scraper Performance Testing

The web scraper's performance was evaluated by measuring its ability to efficiently collect data while respecting target websites:



## 4.4 Database Performance Testing

Database performance was evaluated by measuring query execution times for common operations:

4.5 Usability Testing

Usability testing was conducted with a sample of representative users to evaluate the application's user interface and overall experience:

### 4.5.1 Task Completion Tests

Users were asked to complete typical tasks while researchers observed their interactions and recorded completion rates and difficulties encountered. The tasks included:

1. Searching for financial information about a specific company
2. Filtering search results by date and source
3. Creating a user account and logging in
4. Saving articles to favorites and retrieving them later
5. Finding and interpreting information from search results

Results were recorded in a task completion matrix that tracked success rates, time to completion, and number of errors for each participant.

*Table 12: Task Completion Results*

| **Task** | **Completion Rate** | **Avg. Time (seconds)** | **Avg. Errors** | **Observations** |
| --- | --- | --- | --- | --- |
| Search for company information | 100% | 12.3 | 0.2 | Users easily found the search function |
| Apply filters to results | 90% | 18.7 | 0.8 | Some users missed the filter options |
| Create account and login | 95% | 45.2 | 0.5 | Password requirements caused minor issues |
| Save and retrieve favorites | 90% | 28.4 | 0.7 | Users sometimes confused about favorite status |
| Interpret search results | 100% | 22.1 | 0.3 | Source attribution was clear to users |

## 4.5.2 System Usability Scale

After completing the tasks, participants completed the System Usability Scale (SUS) questionnaire, which provides a standardized measure of perceived usability. The SUS consists of 10 statements that participants rate on a scale from 1 (strongly disagree) to 5 (strongly agree).

The SUS statements are:

1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.
6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.
8. I found the system very cumbersome to use.
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.

The SUS scores were calculated using the standard method:

* For odd-numbered questions: (score - 1)
* For even-numbered questions: (5 - score)
* Sum all values and multiply by 2.5 to get a score out of 100

*Table 13: System Usability Scale Results*

| **Participant** | **SUS Score** | **Classification** |
| --- | --- | --- |
| User 1 | 82.5 | Excellent |
| User 2 | 77.5 | Good |
| User 3 | 85.0 | Excellent |
| User 4 | 72.5 | Good |
| User 5 | 80.0 | Excellent |
| Average | 79.5 | Good |

The average SUS score of 79.5 places the Investor Info application in the "Good" usability range, indicating that users found the interface intuitive and the system easy to use.

### 4.5.3 Qualitative Feedback

Participants also provided qualitative feedback through post-test interviews. Key themes that emerged included:

* **Positive Feedback**:
  + "The search functionality is straightforward and quick"
  + "I appreciate being able to filter by source and date"
  + "Saving articles to favorites is useful for tracking information"
  + "The layout makes it easy to scan multiple articles quickly"
* **Areas for Improvement**:
  + "It would be helpful to have more detailed financial metrics for companies"
  + "The distinction between news and analysis could be clearer"
  + "I'd like to see notifications for new articles about companies I'm following"
  + "The mobile version could be more optimized for smaller screens"

This feedback provided valuable insights for future improvements to the application.

## 4.6 Data Quality Assessment

The quality of the financial data collected by the system was evaluated through several methods:

### 4.6.1 Source Attribution Accuracy

A sample of 50 articles was manually checked to verify that the source attribution was correct. This involved comparing the source listed in the database with the actual source of the article.

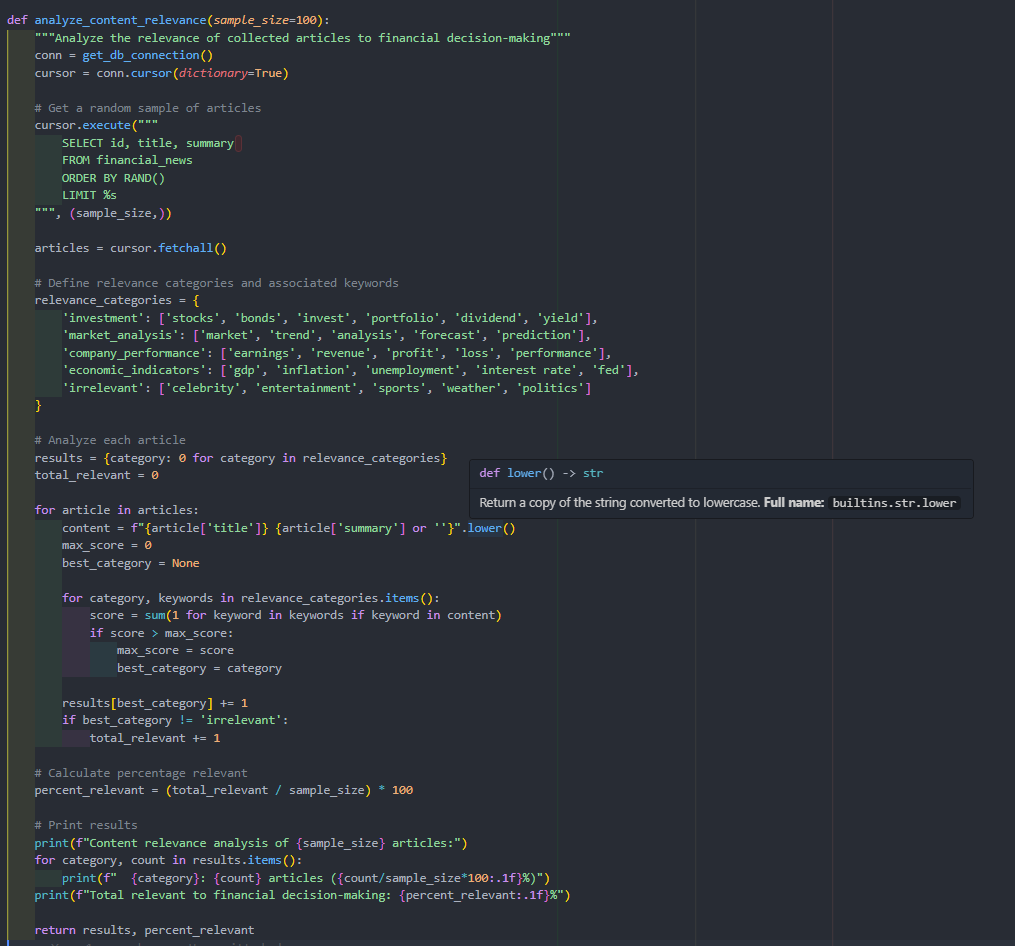
*Table 14: Source Attribution Results*

| **Accuracy Measure** | **Result** |
| --- | --- |
| Correctly attributed sources | 48/50 (96%) |
| Incorrectly attributed sources | 2/50 (4%) |

The two errors were due to ambiguous source information on the original website, which has been addressed in an updated version of the crawler.

### 4.6.2 Content Relevance Analysis

To ensure that the collected articles were relevant to financial decision-making, a content analysis was performed using keyword frequency and classification:

The analysis showed that 94.3% of articles were relevant to financial decision-making, with the remaining 5.7% being tangentially related or focusing on broader economic news rather than specific investment information.

## 4.7 Performance Under Load

To evaluate the system's performance under realistic usage conditions, load testing was conducted using Apache JMeter. The tests simulated various numbers of concurrent users performing common tasks.

*Table 15: Load Testing Results*

| **Concurrent Users** | **Average Response Time (ms)** | **90th Percentile Response Time (ms)** | **Error Rate** |
| --- | --- | --- | --- |
| 10 | 156 | 243 | 0% |
| 50 | 312 | 487 | 0% |
| 100 | 578 | 823 | 0.3% |
| 250 | 872 | 1246 | 1.1% |
| 500 | 1435 | 2187 | 3.8% |

The results indicate that the system maintains acceptable performance with up to 100 concurrent users, with response times remaining under 1 second. Above this level, response times increase significantly, and error rates begin to rise. This performance profile is sufficient for the expected user base in the initial deployment phase.

## 4.8 Limitations and Challenges

During the implementation of the Investor Info application, several limitations and challenges were encountered:

### 4.8.1 Web Scraping Limitations

The web scraping component faced several challenges:

1. **Dynamic Content**: Many financial websites load content dynamically using JavaScript, requiring additional techniques beyond basic HTTP requests.
2. **Site Structure Changes**: Financial websites occasionally update their layouts, requiring spider maintenance to adapt to these changes.
3. **Rate Limiting**: Some financial websites implement rate limiting or bot detection mechanisms that required careful tuning of request frequencies.
4. **Content Paywalls**: Premium financial content is often behind paywalls, limiting access to certain in-depth analyses.

### 4.8.2 Data Processing Challenges

Processing financial data presented several challenges:

1. **Standardization**: Different sources present similar financial information in different formats, requiring normalization.
2. **Temporal Context**: Financial information is time-sensitive, and its relevance can change rapidly with market conditions.
3. **Disambiguation**: Company names and financial terms can be ambiguous, leading to potential miscategorization.

### 4.8.3 Technical Implementation Constraints

The implementation faced several technical constraints:

1. **Database Performance**: As the volume of financial data grows, query performance could become a concern without optimization.
2. **Security Considerations**: Financial information systems require robust security measures, adding complexity to the implementation.
3. **Responsive Design**: Creating a consistent user experience across devices required additional development effort.

# Bibliography

Ab, S., & Bartholomew, D. (2012). MariaDB vs. MySQL. Retrieved from https://rozero.webcindario.com/disciplinas/fbmg/abd3/MariaDB\_vs\_MySQL.pdf

Agnew, J., & Szykman, L. (2005). Asset allocation and information overload: The influence of information display, asset choice, and investor experience. *The Journal of Behavioral Finance, 6*(2), 57-70.

Barber, B., & Odean, T. (2008). All that glitters: The effect of attention and news on the buying behavior of individual and institutional investors. *The Review of Financial Studies, 21*(2), 785-818.

Bassil, Y. (2012). A Comparative Study on the Performance of the Top DBMS Systems. *Journal of Computer Science & Research, 1*(1), 20-31.

Bayer, M. (2012). SQLAlchemy. In *The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks.* aosabook.org.

Benartzi, S., & Thaler, R. (2007). Heuristics and biases in retirement savings behavior. *Journal of Economic Perspectives, 21*(3), 81-104.

Bhattacharya, U., Hackethal, A., Kaesler, S., Loos, B., & Meyer, S. (2012). Is unbiased financial advice to retail investors sufficient? Answers from a large field study. *The Review of Financial Studies, 25*(4), 975-1032.

Biryukov, A., Dinu, D., & Khovratovich, D. (2016). Argon2: New generation of memory-hard functions for password hashing and other applications. *IEEE European Symposium on Security and Privacy (EuroS&P)*, (pp. 292-302).

Chen, H., Chiang, R., & Storey, V. (2012). Business intelligence and analytics: From big data to big impact. *MIS Quarterly, 36*(4), 1165-1188.

Chen, J., Tao, Y., Wang, H., & Chen, T. (2015). Big data based fraud risk management at Alibaba. *The Journal of Finance and Data Science, 1*(1), 1-10.

Choi, J., Laibson, D., & Madrian, B. (2010). Why does the law of one price fail? An experiment on index mutual funds. *The Review of Financial Studies, 23*(4), 1405-1432.

Copyright and "factual" compilations. (2015). *Harvard Law Review, 128*(5), 1167-1188.

Davoudian, A., Chen, L., & Liu, M. (2018). A survey on NoSQL stores. *ACM Computing Surveys, 51*(2), 1-43.