
# ITCS-5102: Survey of Programming Languages

# ScrapeVerse: Web Scraping for Real-Time Stock Market Data

**By Compile Crew**

Tarun Lagadapati,

Krishna Teja Rayapudi,

Sudhish Cherukuri,

Saivenkat Chinta,

Jashmitha Daggupati

# Introduction – Motivation

The motivation behind ScrapeVerse comes from the growing importance of having quick access to real-time financial data, such as stock prices and percentage changes. Financial decision-making, whether for personal investing or business analytics, requires accurate, up-to-date information. ScrapeVerse automates this process by scraping financial data from various online sources. The project enables users to access timely stock market data, eliminating the need to manually gather this information from multiple platforms.

Go (Golang) was chosen as the programming language for ScrapeVerse due to its simplicity, efficiency, and robust support for concurrent processing. Go's built-in concurrency model, based on goroutines, is perfect for scraping data concurrently from multiple stock tickers. The language's static typing ensures high performance and reliability, while its simple syntax reduces the development time for a project of this scale. Additionally, Go's strong support for web servers with frameworks like Gin was instrumental in building the RESTful API that serves the scraped data.

# Language Specifications:

## Go Paradigm

Go is a statically typed, compiled, procedural programming language with built-in support for concurrent programming. Although it follows a procedural approach, Go allows object-oriented principles through the use of structs and interfaces, providing a high level of flexibility and scalability. The concurrency model is based on lightweight goroutines, which allows for efficient multitasking, making it ideal for web scraping tasks where multiple data points are collected in parallel.

## History

Go was developed by Google engineers Robert Griesemer, Rob Pike, and Ken Thompson in 2007 and was officially released in 2009. It was designed to address challenges faced by developers in large-scale software systems, particularly with languages like C++ and Java, which, while powerful, are often cumbersome to work with. Go's design aims to be simple, efficient, and conducive to high-performance systems, particularly in cloud computing and web services. Go has since grown in popularity for its simplicity, strong concurrency support, and

excellent performance. It is widely used for building web servers, networking tools, and scalable applications, especially in environments where performance is critical.

## Elements of Go

**Reserved Words:** Keywords such as package, import, func, var, const, type, interface, struct, and return is part of Go's syntax.

**Primitive Data Types:** Go supports a range of primitive types, including int, float64, string, bool, byte, and rune.

**Structured Types:** Go allows the creation of custom types using struct, which can be used to define complex data structures.

## Syntax

A Go program consists of the following parts:

- Package declaration
- Import packages
- Functions
- Statements and expressions

For example,

```
package main
import "fmt"
func main() {
   var message string = "Hello, World!"
   fmt.Println(message)
}
```

## Control Abstraction

Loops: Go uses the for loop for all iteration constructs.

```
for i := 0; i < 10; i++ {
   fmt.Println(i)
}
```

Conditionals: Go provides if, else, and switch statements for conditional control.

```
package main

import "fmt"

func main() {
    x := 5
    if x > 0 {
        fmt.Println("x is positive")
    } else {
        fmt.Println("x is non-positive")
    }
    day := 2
    switch day {
    case 1:
        fmt.Println("Monday")
    case 2:
        fmt.Println("Tuesday")
    default:
        fmt.Println("Other day")
    }
}
```

## Abstraction in Go

Functions: Functions are a fundamental part of Go. They are defined using the func keyword and can accept parameters and return values.

```
func add (a int, b int) int {
    return a + b
```

}

Interfaces: Go supports interfaces to provide polymorphism. A type can implement an interface implicitly, meaning that no explicit declaration is needed.

## Evaluation of Go

**Writability:** Go's syntax is simple and minimalist, making it easy to write clear and maintainable code.

**Readability:** Go enforces consistent formatting using tools like gofmt, which ensures that all Go code is formatted uniformly.

**Reliability:** Go's static typing and garbage collection make it a reliable language for building production-level systems.

## Strengths and Weaknesses

**Strengths:** Simple syntax, strong concurrency support, high performance, excellent library ecosystem.

**Weaknesses:** Lack of generics (though this is now addressed in Go 1.18), verbosity in error handling, limited functional programming support.

## Overview of the Program

The provided Go program is a web-based stock data scraping and analysis tool. It uses the following components and features:

- Web Framework (gin-gonic): Implements a RESTful API and serves HTML pages for a web interface.
- Web Scraping (colly): Fetches stock price and change data from Yahoo Finance.
- File Handling (os, encoding/csv): Writes scraped stock data to a CSV file and allows users to download it.
- Sorting and Searching: Implements stock data sorting (by price) and searching (by symbol).
- Top Performers: Identifies the top gainer and loser based on percentage change.
- Concurrency Handling: Uses a flag (hasScrapedStock) to prevent duplicate scraping requests.
- Error Handling: Manages errors for scraping, file operations, and user queries.

## Language Features Highlighted

1. **Concurrency and State Management**

The program uses the hasScrapedStock flag to track the scraping status. Go's lightweight goroutines and channels are not explicitly used here, but Go is well-suited for managing concurrent web scraping tasks if extended.

Ease of Use: Go's built-in concurrency model makes state and task management simple and predictable.

2. **Third-Party Package Integration**

The program relies on gin-gonic for web routing and colly for web scraping.

Ease of Use: Go's modular architecture and go mod dependency management simplify the inclusion and use of third-party libraries.

3. **Error Handling**

Implements error handling for file operations, scraping, and malformed user inputs using Go's idiomatic error type.

Ease of Use: Error propagation using if err != nil is explicit but verbose, which can increase code clarity at the expense of verbosity.

4. **Data Structures**

Defines a Stock struct with tags for JSON serialization.

Uses slices for dynamic arrays and maps for efficient stock symbol lookups.

Ease of Use: Go's struct system is simple and supports direct JSON encoding/decoding, making it easy to interact with APIs and web services.

5. **HTTP Handling**

The gin-gonic package enables clean routing and middleware integration.

Example: r.GET("/api/stocks", getStockData)

Ease of Use: The gin package abstracts much of the boilerplate associated with HTTP servers in Go.

6. **File Operations**

Uses os and encoding/csv to create, write to, and serve a CSV file.

Ease of Use: Go's standard library is robust for file I/O operations, offering cross-platform compatibility and simplicity.

7. **Web Scraping**

Implements web scraping using colly, with handlers for specific HTML elements.

Example:

```
c.OnHTML("fin-streamer[data-field='regularMarketPrice']", func(e *colly.HTMLElement) {
    price, _ := strconv.ParseFloat(e.Attr("data-value"), 64)
})
```

Ease of Use: The colly library integrates smoothly with Go's type system and error handling.

8. **String Manipulation and Parsing**

Uses strings for text operations and strconv for converting strings to numbers.

Example:

```
changeStr := strings.Trim(stock.Change, " ()%")
changePercentage, err := strconv.ParseFloat(changeStr, 64)
```

Ease of Use: String handling is straightforward but lacks some advanced utilities found in scripting languages.

## Easy to Implement

- Standard Library: The robust and well-documented standard library made tasks like file handling, HTTP routing, and JSON encoding straightforward.
- Static Typing: Go's strong typing system helped catch errors early during compilation, ensuring fewer runtime bugs.
- Third-Party Ecosystem: Libraries like gin and colly abstracted much of the complexity

in web development and scraping.

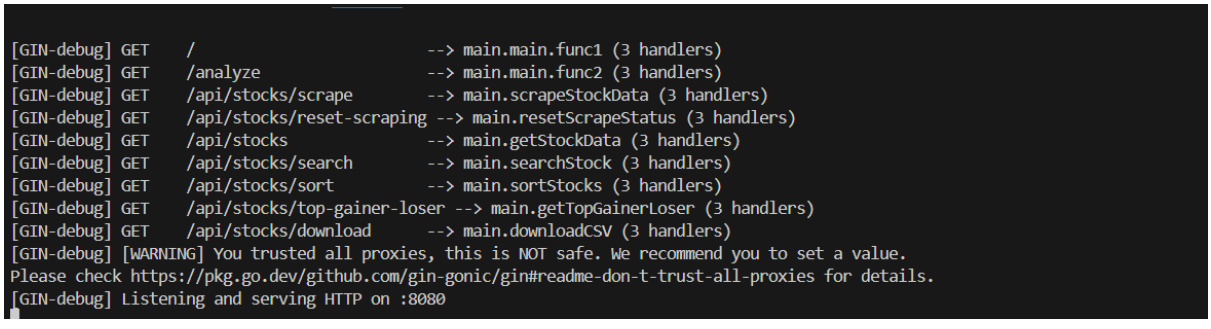
- Error Handling: Explicit error handling provided clarity and control over possible failure points.

### Challenging to Implement

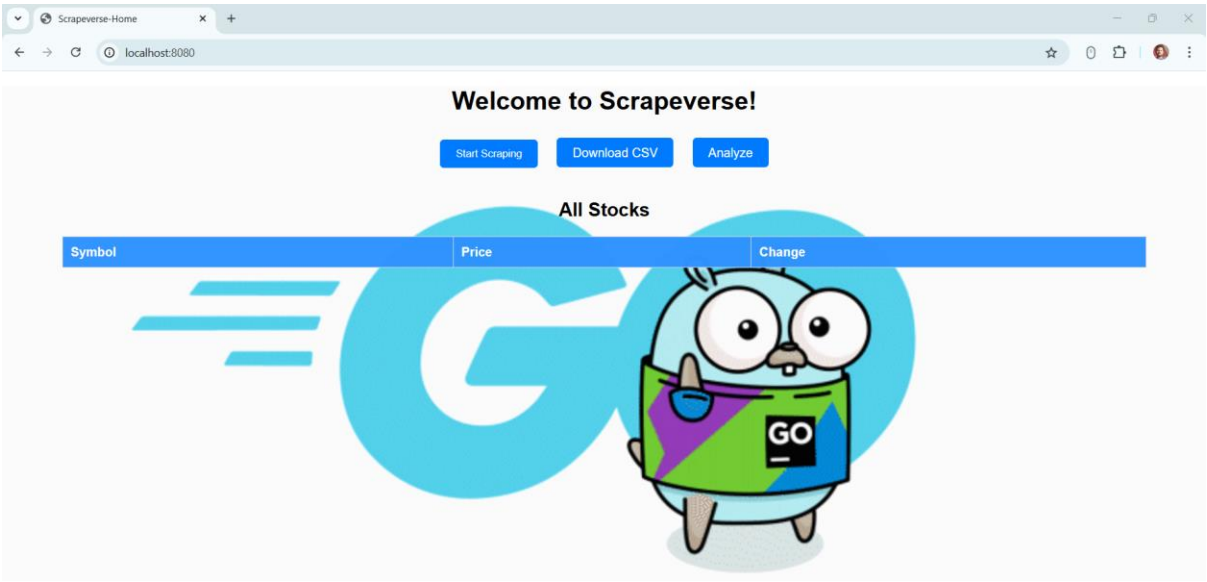
- Verbosity: Go's minimalistic design leads to verbose error handling and repetitive constructs, which can slow down development.
- Limited Language Features:
  - ✓ No native support for generics (prior to Go 1.18) made certain operations on slices/maps more cumbersome.
  - ✓ Lack of syntactic sugar (e.g., Python-style comprehensions) for common patterns increases boilerplate.
- Concurrency Complexity: If the program were to use goroutines for scraping multiple URLs concurrently, managing synchronization could be challenging without proper design.

## Sample Run of the Program

This debug log shows routes and handlers set up in a Gin web framework in Go, mapping endpoints (e.g., /api/stocks/scrape) to their respective functions (e.g., main.scrapeStockData). Each route processes HTTP requests with middleware and a handler function. The warning suggests securing the app by configuring trusted proxies using gin.SetTrustedProxies. The server listens on port 8080, ready to handle requests. For testing, use tools like Postman or curl to ensure routes and responses work as expected.



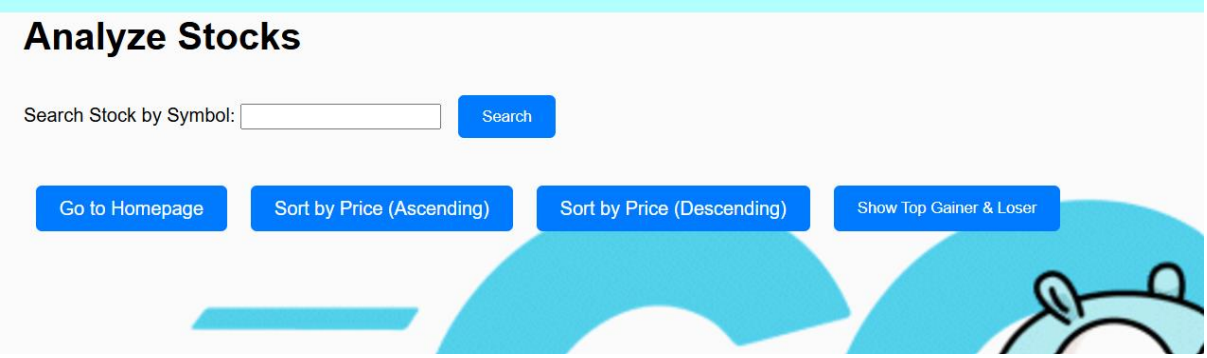

When the server starts on localhost:8080, the homepage initially shows no data because the scraping process hasn't run yet. Clicking "Start Scraper" triggers the /api/stocks/scrape endpoint, which fetches data, processes it, and updates the server. Once complete, the homepage reloads or fetches the updated data from /api/stocks to display it.



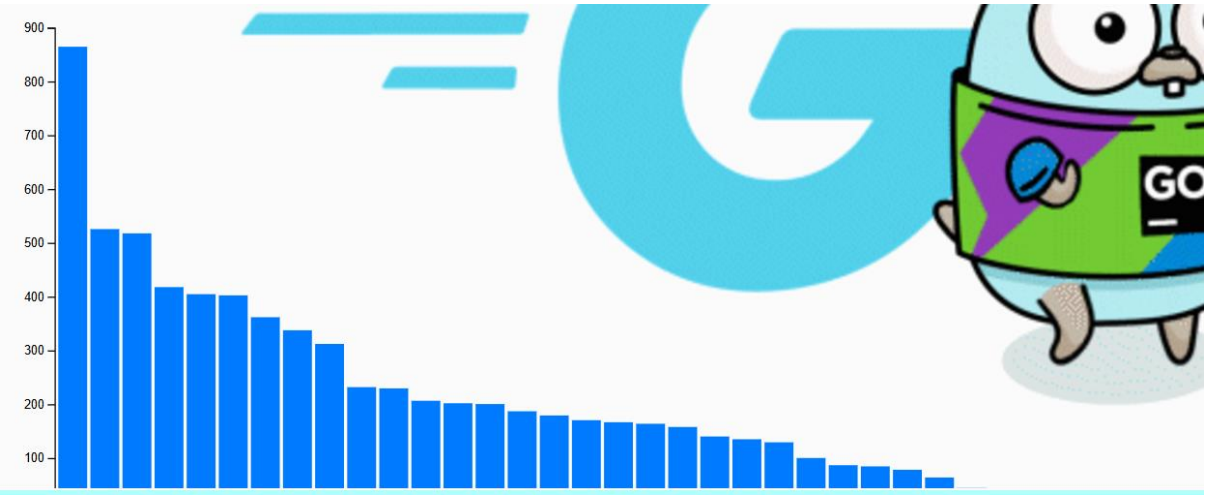

When the server starts on localhost:8080, the homepage initially displays no data. Clicking "Start Scraper" triggers the /api/stocks/scrape endpoint, which retrieves stock data from Yahoo Finance and updates the database. Once scraping is complete, the homepage reloads or fetches the data from /api/stocks to display it. The terminal logs the data retrieval process during scraping. If you click "Start Scraper" again, the system won't re-run the scraper immediately. Instead, it returns the message: "Scrape request received. HasScrapedStock: true". This ensures the scraper doesn't duplicate work unnecessarily.

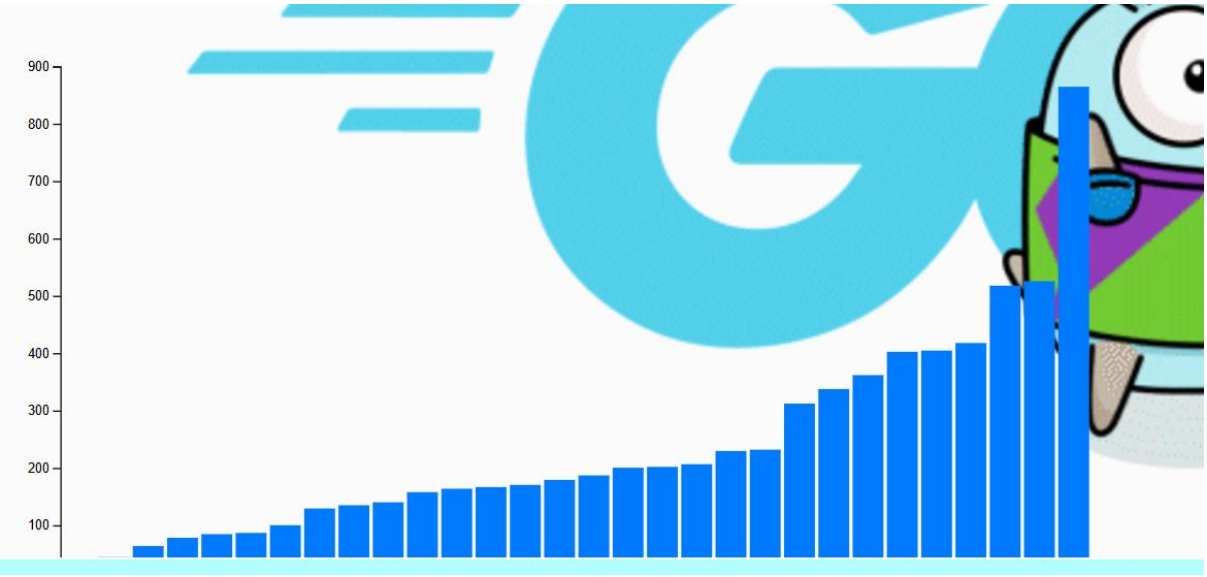
```
Visiting: https://finance.yahoo.com/quote/BABA
Visiting: https://finance.yahoo.com/quote/NFLX
Visiting: https://finance.yahoo.com/quote/AMT
Visiting: https://finance.yahoo.com/quote/MDLZ
Visiting: https://finance.yahoo.com/quote/MRK
Visiting: https://finance.yahoo.com/quote/NKE
Stock data successfully written to CSV!
[GIN] 2024/11/25 - 18:57:13 | 200 |   33.2175652s |             ::1 | GET      "/api/stocks/scrape"
[GIN] 2024/11/25 - 18:57:13 | 200 |            0s |             ::1 | GET      "/api/stocks"
Scrape request received. HasScrapedStock: true
[GIN] 2024/11/25 - 18:58:46 | 400 |       610.5µs |             ::1 | GET      "/api/stocks/scrape"
[GIN] 2024/11/25 - 18:58:46 | 200 |            0s |             ::1 | GET      "/api/stocks"
```

The analysis page includes a search bar for finding specific stocks and options for sorting data in ascending or descending order. It also features a filter to display the top gainers or losers in the stock market. These tools help users analyze stock performance efficiently.




The analysis page provides interactive features for data visualization. When sorting in ascending order, the data is displayed in a pictorial representation with values organized from smallest to largest. Similarly, selecting descending order reverses the order, showing the largest values first in the same visual style. The search bar allows users to find specific stocks by entering keywords, and the results are displayed dynamically based on the query. This ensures quick and accurate stock analysis. Sorting and searching work seamlessly together, enabling users to refine their view. The visuals make it easier to understand trends and patterns in the data.



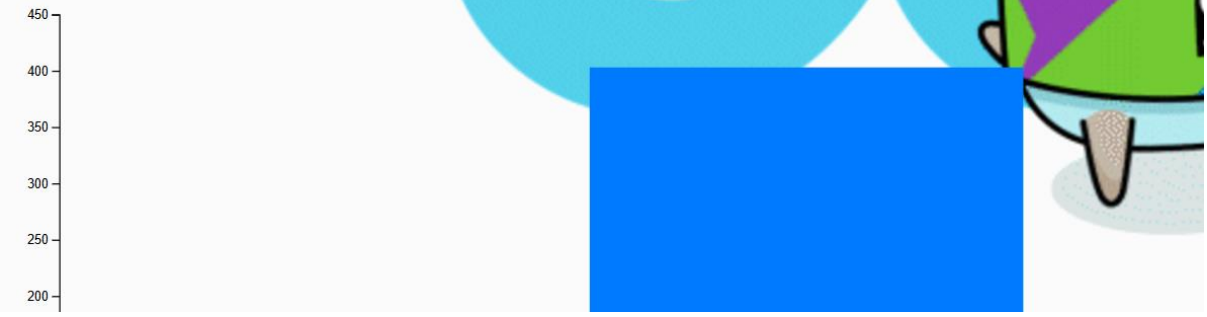




## Top Gainer

Symbol: APLD, Price: 10.68, Change: (+8.43%)

## Top Loser

Symbol: MSTR, Price: 403.45, Change: (-4.37%)



# Problem Definition

The main problem addressed by ScrapeVerse is the manual and time-consuming process of collecting real-time stock market data from multiple online sources. Stock prices, percentage changes, and other key financial metrics are typically scattered across various websites or platforms. This necessitates constant manual monitoring, which is not only labor-intensive but also prone to errors. For investors, analysts, and traders, having up-to-date and accurate stock data is crucial to making informed financial decisions. ScrapeVerse automates the process of scraping real-time stock data from predefined websites, aggregates the information, and organizes it in an easily accessible format for immediate use. This provides a more efficient and reliable solution to stay updated with the stock market.

## Use Cases

**Scraping Stock Data:** The program scrapes real-time data, such as prices and percentage changes, from yahoo finance websites.

**Viewing Stock Information:** Users can view the current price and percentage change of a stock by querying the API.

**Searching for Stocks:** The application allows users to search for stocks by their ticker symbols.

**Sorting Stocks:** Users can sort stocks based on their price change.

**Displaying Top Gainers and Losers:** The application displays the top gainer and top loser for the day based on percentage change.

**Downloading Data:** Users can download the scraped data as a CSV file for further analysis.

# Proposed Method

## Intuition

The key intuition behind ScrapeVerse is the ability to scrape data from financial website concurrently using Go's goroutines. This allows the application to gather stock prices and changes quickly and efficiently. Colly, the web scraping library, handles the intricacies of extracting data from HTML pages, while Go's concurrency model ensures that multiple scraping tasks can be executed simultaneously.

## Algorithms

**Scraping Algorithm:** The algorithm starts by visiting the stock's page on the predefined website using Colly. It then locates the relevant HTML elements (e.g., price and percentage change) using CSS selectors and extracts the data.

**Data Handling:** The data is stored in a structured format (CSV) for easy retrieval and analysis.

**Sorting and Filtering:** The application can sort and filter stocks based on their price.

**API Method:** A RESTful API is exposed using Gin, allowing users to query and retrieve the scraped stock data in JSON format.

# Experiments

The provided Go programs illustrate several key features of the Go programming language and demonstrate its capability to handle diverse programming constructs, including error handling, data structures, arithmetic operations, and string manipulation. Here's a concise overview of the features and insights:

# Concepts Demonstrated:

**Error Handling Using panic and recover:**

The handlePanic function showcases Go's mechanism for handling unexpected runtime errors.

Examples:

```
defer handlePanic()

if num == 0 {

   panic("Division by zero error!")

}
```

**Working with Structs:**

The Person struct demonstrates defining and using custom data structures.

Example:

```
type Person struct {
```

```
    name   string
    job    string
    salary int
}
```

**Control Structures (if-else, loops):**

Go's basic control structures are used for conditional logic and iteration.

Example:

```
if person1.salary > person2.salary {
    fmt.Printf("%s earns more.\n", person1.name)
}
```

**Dynamic Arrays and Slices:**

Demonstrates the use of slices for generating Fibonacci numbers.

Example:

```
var fib []int
fib = append(fib, 0, 1)
```

**Mathematical Operations**:

Use of the math package for advanced calculations like square root and power.

Example:

```
max := math.Max(float64(a), float64(b))
sqrt := math.Sqrt(float64(a))
```

**String Manipulations:**

The strings package is used to transform and manipulate text data.

Example:

```
uppercase := strings.ToUpper(str1)
```

**Boolean Operations**:

Demonstrates logical operations like AND and OR.

Example:

fmt.Printf("AND of %t and %t: %t\n", bool1, bool2, bool1 && bool2)

**How These Programs Help in Understanding Go:**

- Error Resilience: By using defer, panic, and recover, programmers learn Go's robust error-handling paradigm.
- Flexibility of Data Types: From structs to slices, the examples show Go's simplicity in working with various data types.
- Performance-Oriented Syntax: The programs highlight Go’s efficiency and straightforwardness in writing performant code.
- Readability: Go enforces clean, concise code, which is evident in the examples provided.
- Package Utilization: Go encourages modular design and reusable packages like fmt, math, and strings.

**Description of the Test Bed**

The test bed comprises a series of Go programs designed to explore and evaluate various language features and programming constructs in Go. Each program is crafted to test specific aspects of Go, such as error handling, data structures, arithmetic operations, string manipulations, and control structures. The experiments utilize standard Go libraries (fmt, math, strings) and focus on idiomatic usage.

**General Questions Across All Experiments**

- How does Go enforce type safety and avoid common errors like null dereferencing or type mismatches?
- How readable and maintainable is Go code for small to medium-scale problems?
- How does Go’s standard library enhance productivity in common programming tasks?
- What features of Go make it a good candidate for performance-critical applications?
- How does Go's error-handling mechanism compare to exception-based handling in other languages like Java or Python?

**Insights from These Experiments**

The programs collectively test Go's language features, standard library, and idiomatic usage. By addressing these questions, the experiments reveal how Go achieves a balance between simplicity, performance, and developer productivity.

## Conclusion:

The experiment implements a robust stock analysis application using the Go programming language, leveraging the Gin web framework and Colly web scraping library. It demonstrates various advanced programming concepts and practices, including web scraping, RESTful API development, CSV file handling, and real-time data processing. The modular design ensures clean code organization, with each function focused on specific tasks such as scraping, searching, sorting, and file management.

The application efficiently fetches stock data from Yahoo Finance for predefined stock symbols, processes it, and stores the results in memory and CSV format. It supports interactive functionalities like searching for specific stocks, sorting by price, identifying top gainers and losers, and downloading the data. The use of Gin enhances the responsiveness and scalability of the server, making it capable of handling multiple concurrent requests.

Error handling is carefully incorporated, particularly in the scraping process, ensuring the application does not crash under unexpected circumstances. The resetScrapeStatus API route provides users with the flexibility to refresh stock data for accurate analysis. The application demonstrates how Go's concurrency, efficient libraries, and simple syntax enable the development of performant and reliable web-based tools.

In summary, this project is a testament to Go’s utility in building data-centric, high-performance applications. It highlights best practices for handling APIs, file systems, and external libraries while showcasing the power of web scraping and real-time data analysis.

**GitHub:** https://github.com/Scheruk1701/Scrapeverse