MSCS 621L - Fall 2025 - R. O. Topaloglu

News API
Project 1

Tyler DeLorey
tyler.delorey1@marist.edu

September 28th, 2025

Contents

1 How to Setup Project

1.1 Installing Docker

This project uses Docker in order to set up everything correctly. I would recommend installing Docker Desktop, as that is what I used and it worked out. To learn how to install Docker Desktop, click the link here.

Now you should have installed Docker Desktop and created an account. You can type `docker version` in your CLI of choice to see if it installed correctly. If it did, in the directory of this project, if you aren't logged in, type `docker login` to login to your account.

1.2 Commands to Build Image and Run Containers

In your directory, make sure you have the `proj1_image.tar` file installed, since that is the Docker Image.

To load this Docker Image into your own Docker, use the command:

- `docker load -i proj1_image.tar`

Now that the image is loaded, you can run a container and the program using the following command:

- `docker run --rm -e NEWSAPI_KEY='apiKey' -e FILE='file.txt' -e WORKERS='num' -v news_cache_volume:/app proj1`
  - `--rm` makes sure the container is removed after we run it, since for this project and my implementation, the container doesn't need to run after the program ends.
  - `NEWSAPI_KEY`, `FILE`, and `WORKERS` are all environmental variables used to run the program. These values need to be replaced with certain values.
    - `apiKey` needs to be replaced with your actual API key from NewsAPI. An account can be made for free there.
    - `file.txt` needs to be replaced with the name of the text file. The text file is important, as these contain the requests from the users that the program will be using. In the image already, there are three .txt files that can be used, `Xprompts.txt`, `Yprompts.txt`, and `Zprompts.txt`. These files will be explained in Section 2.2.
    - `num` needs to be replaced with a positive integer. This program partially uses a worker pool pattern, meaning there can be multiple workers doing some work to speed up this program and help with concurrency [Worker Pools]. This number can be manipulated. I recommend having it set to around the same amount of

requests coming in. For example, if there are 10 requests coming in, I would set my workers to around 10, but this number can be played with.

  – `news_cache_volume:/app proj1` is how we store the database results. This Docker Volume will be created automatically if it doesn't already exist. This volume will exist even after the program finishes running, so running another program has the chance of pulling database results if queries are similar.

If you want to delete the volume `news_cache_volume` that stores the database, you can run the following command:

- `docker volume rm news_cache_volume`

Now because files are embedded into the Docker Image, if you wanted to create your own test case, you would need to add those .txt files into your directory (in a specific format, as will be explained in Section 2.2), and then you would need to rebuild the image.

To rebuild the image, make sure you have the Dockerfile. Then, you run the command:

- `docker build -t proj1 .`

Make sure you have the `.` at the end!

If you want to save the image on your computer, run the command:

- `docker save -o proj1_image.tar proj1:latest`

The next section has all of the needed information for the software architecture used in the code and why these decisions were made.

2 Explanations

2.1 Dockerfile

Below is my implementation of the Dockerfile, the file that is used (with the source code) to build the Docker Image for this program.

```dockerfile
# -- BUILD STAGE --
FROM golang:1.25.1-alpine AS build-stage

# Install build dependencies
RUN apk add --no-cache upx

WORKDIR /app

# Download and Cache Go modules
COPY go.mod go.sum ./
RUN go mod download

# Copy the source code
COPY *.go .

# Build static binary with stripped debug info (and disables SQLite extension loading)
RUN CGO_ENABLED=0 GOOS=linux go build -tags "sqlite_omit_load_extension" -ldflags="-s -w" -o proj1

# Compress binary with UPX
RUN upx --best --lzma proj1

# -- RUNTIME STAGE --
FROM scratch

WORKDIR /app

# Copy binary from builder stage
COPY --from=build-stage /app/proj1 .

# Copy CA certificates for HTTPS
COPY --from=build-stage /etc/ssl/certs/ca-certificates.crt /etc/ssl/certs/

# Copy all input files to the image
COPY *.txt .

# Run default command
CMD ["./proj1"]
```

Figure 2.1: Dockerfile

The build stage is where a chunk of the code and the data is. This data is only loaded once, as Go modules are cached to speed up rebuilds. It uses the Go Alpine Image, which is small, lightweight, and efficient for our program. It installs build dependencies, like `upx`, which is a small image for usage in multi-stage Docker builds to compress binary files like Go or Rust [UPX Source17]. We set our working directory to `/app`, and download our program dependencies found in `go.mod`, and the checksum found in `go.sum`. After we get our source code into the Image, it will build the static binary (with some other extra settings to lower the size of the Image), and it will be compressed using `UPX`.

At the runtime stage, we start from an empty Image. This greatly reduces the Image size because the runtime build only contains what we explicitly copy. After copying the compressed static binary from the build stage, we have to worry about HTTP certificates. As explained by [CA Certificates24], we wouldn't be able to make the API call without a certification. Using a specific filepath found in [SSL Certificate Location on Unix/Linux09], we can get ourselves a certificate so we can use HTTP. After we get a certificate, this is where all of the user request test cases are added to the Image. This is why you would need to rebuild the Image if you wanted to add new test cases. Then, we specify the default command that will run when a container starts from this image.

It was a difficult decision to make the user rebuild the Image everytime they wanted to add new input files. However, a lot of the workarounds were OS-dependent, meaning the commands differ between Windows and Linux. So, I decided to have the user add the files to the Image itself so it could be used in the containers.

The Docker Image size is only around 6.8 MB, mostly because the runtime uses scratch, but also because we are using a multi-stage build and requiring containing only the compiled binary, certificates, and input files, things we definitely need.

Most of the information I used to create my Dockerfile, for Golang specifically, was found here: [Golang Build Images Guide24].

2.2 Software Architecture with Input Files

The three input files for this program are `Xprompts.txt`, `Yprompts.txt`, and `Zprompts.txt`. Each serve some purpose to test this program. Of course, you can add your own test cases, but these are the ones I found helpful to test my program.

`Xprompts.txt`

Below displays the test case presented in `Xprompts.txt`

```
music|3|9
bitcoin|2|20
hello world|5|2
america|2|5
golang|3|15
meaning of life|7|12
breaking news|1|8
sports|4|10
weather|2|7
technology|6|25
```

Figure 2.2: Xprompts.txt

This is the most basic test case, but it helps us look at the format that the files need to be in. Each line represents a users request, and each line has three fields, separated by |.

- The first field is the `query`, or the search the user wants to make.
- The second field is the `days`, filtering these search results by date. For example, if the days says 2, that means only include articles that are from today or yesterday.
- The third field is the `limit`. This also filters the results by only displaying `limit` search results (or less if the query returns a less amount of articles).

`Yprompts.txt`
Below displays the test case presented in `Yprompts.txt`

```
daft punk|10|5
daft punk|5|1
daft punk|15|1
beatles|7|3
beatles|5|2
beatles|10|1
pink floyd|3|10
pink floyd|5|2
pink floyd|1|1
queen|0|5
queen|2|3
queen|10|1
radiohead|7|7
radiohead|3|2
radiohead|10|1
nirvana|5|5
nirvana|5|1
nirvana|7|3
```

Figure 2.3: Yprompts.txt

This test case is important because it tests out the cache, which is the special aspect that I added to this project (this program also has a test case on Line 10 if the date is an invalid number). Usually, writing to the database can be slow. This means that if we have multiple matching queries, it will take some time to wait for the write to happen before reading from it and getting the results from the database. I created the cache because reads and writes are easier and faster. Let's take a look at this test case to see what I mean.

Because these requests run concurrently (which I will explain when looking at the code), there is no guarantee that it will run in the order that is showed in the file. However, assuming it does, lets take a look at the requests. The first request is daft punk, only showing articles from the past 10 days, and only showing a limit of 5 articles. This will pull from the API, save to the cache, and eventually will be saved to the database. The second request has the same query, but since it has a smaller date, it will pull from the cache and will not do an API call. The third request also has the same query, but it has an older date, meaning it would need to pull from the API, then it will save to the cache and database.

The cache is reset after the program ends, but if this exact program is run for a second time, all results will be fetched from the database, making it way faster than doing API calls.

This is something important to note. There is no way to limit the amount of articles you get back from NewsAPI (pageSize and pageLimit don't do actual limit the results, just its display). So, since I'm getting older data I'm not using anyways, I will store it in the database. Even though it will take up more space in the volume, it will save time because you wouldn't need to do another API call if the limit gets higher, but date and query stay the same.

`Zprompts.txt`

Below displays the test case presented in `Zprompts.txt`

```
golang|5|5
golang|3|5
golang|7|5
golang|5|3
golang|5|10
python|2|5
python|5|5
python|5|10
python|7|3
python|3|3
docker|5|5
docker|2|5
docker|5|2
docker|7|7
docker|5|10
```

```
java|3|5
java|5|5
java|7|5
java|5|2
java|10|10
```

Figure 2.4: Zprompts.txt

This just tests the cache more thoroughly by having more of the same queries, while changing the date and limit.

API calls are the slowest thing in this program. Waiting for database writes is also pretty slow, so that's why I use the cache instead. When running the program another time, the cache gets reset and results are pulled from the database.

2.3 Imports, Global Variables, and Structures

Now let's take a look at the code that is used to run this program. Below are the imports this program requires:

```
1  package main
2
3  import (
4    "bufio"
5    "database/sql"
6    "encoding/json"
7    "fmt"
8    "net/http"
9    "net/url"
10   "os"
11   "strconv"
12   "strings"
13   "sync"
14   "time"
15
16   _ "modernc.org/sqlite"
17 )
```

Figure 2.5: Imports (proj1.go)

I'm not going to go over all of these imports, but one that is very import is the last line shown on Line 16. I'm using sqlite to run the database. SQLite is an in-process implementation of a self-contained, serverless, zero-configuration, transactional SQL database engine [modernc.org/sqlite Package25]. This import affects our `go.mod` and `go.sum` files, because it has dependencies and checksums.

The other imports are also important, but we will get to the use for all of them later.

Below are some global variables that very important and used throughout the program.

```go
19 // Global variables
20 var (
21   // Reference to database (news_cache.db)
22   db *sql.DB
23 
24   // Channel for writing results to DB safely
25   // Holds the request as well as its corresponding response
26   writeChan chan reqNresp
27 
28   // Mutex used to check cache to see if query has been asked
     before
29   cacheMu sync.RWMutex
30   cache   = make(map[string]*reqNresp)
31 
32   // All workers with the same query (and correct parameters) use
     the same mutex.
33   queryMutexesMu sync.Mutex
34   queryMutexes   = make(map[string]*RequestMutex)
35 )
```

Figure 2.6: Global Variables (proj1.go)

- `db` (Line 22), is a pointer to the database. This is important, because without this, we wouldn't have a database
- `writeChan` (Line 26), is a channel. Channels are very important when using Goroutines, as it passes data from one goroutine to another [Channels]. The write channel stores the writes that need to be done to the database after a request has been processed.
- `cacheMu` (Line 29), is a Mutex. A Mutex is another way to safely access data across multiple goroutines [Mutexes]. It is used here to lock reads and writes for the `cache` (Line 30). The cache maps a query to a request and the corresponding response (structures will be explained later in this section). The Mutex is used with the cache to make sure safe reads and safe writes occur. Because this program uses concurrency, dangerous things could occur without the Mutex.
- `queryMutexesMu` (Line 33), is also a Mutex. This Mutex works with `queryMutexes` (Line 34), what is a map that maps a query to a Mutex. This is something very important to understand. Because requests happen concurrently, if two identical requests happen at the same time, they will both use the API call. However, the `queryMutexes` map makes

sure to lock any other requests with the same name until that original request has been processed. That means that the first request will use the API, and the second will use the cache.

Below are the self-made structures used in this program:

```go
// Structure for blocking off certain requests if similar requests are being processed
type RequestMutex struct {
  Request SearchRequest
  Mutex   *sync.Mutex
}

// A structure based off of the user request
type SearchRequest struct {
  Query string
  Days  string
  Limit string
}

// Structure for the source of each Article
type Source struct {
  ID   string `json:"id"`
  Name string `json:"name"`
}

// Structure for each article that the API returns
type Article struct {
  Source      Source `json:"source"`
  Author      string `json:"author"`
  Title       string `json:"title"`
  Description string `json:"description"`
  URL         string `json:"url"`
  URLToImage  string `json:"urlToImage"`
  PublishedAt string `json:"publishedAt"`
  Content     string `json:"content"`
}

// The initial response response from the API contains status, totalResults, and the articles
// Use of JSON tags to map JSON fields to Go fields
type NewsAPIResponse struct {
  Status       string    `json:"status"`
  TotalResults int       `json:"totalResults"`
  Articles     []Article `json:"articles"`
  Message      string    `json:"message"`
}

// Structure used to write results to DB safely, storing request and corresponding result
type reqNresp struct {
  req  SearchRequest
  resp NewsAPIResponse
}
```

Figure 2.7: Structures (proj1.go)

- `RequestMutex` (Line 38) is a structure that stores the request as well as the corresponding Mutex. This is used to block off similar requests being processed
- `SearchRequest` (Line 44) is used to store the fields of a request. It has the query, the number of days, and the limit. These are all strings because they may be passed into a URL.
- `reqNresp` (Line 78) is used for writing results into the database and getting results from the cache. It stores the request with its corresponding result.
- These next couple of structures are based specifically on the response JSON object that NewsAPI returns.
  - `Source` (Line 51) gets the Source of each article in JSON format.
  - `Article` (Line 57) gets information on each Article from the response in JSON format.
  - `NewsAPIResponse` (Line 70) gets the full response from NewsAPI in JSON format.
  - `NewsAPIResponse` is required to get the results, while `Article` and `Source` are used for organization and clean code.

The documentation for the NewsAPI response can be found here [NewsAPI Documentation25].

2.4 Helper Functions

Here I will explain the basis for all of the functions that help make this program work. Some of these functions are long, so I will only be explaining parts that I find worthy of an explanation, since I already explained most of my logic behind the software architecture as a whole.

`check()`

```go
// Panic if there was an error
func check(e error) {
  if e != nil {
    panic(e)
  }
}
```

Figure 2.8: check() (proj1.go)

This program will return an error if something went wrong. This function is called a lot throughout the program, especially in cases where if something goes wrong, it will mess up the entire program. Errors for single requests will not end the program, it will just return an invalid request and skip it.

parseLine()

```go
// Parses each line of the file into a Request
func parseLine(text string, lineNum int) (SearchRequest, bool) {

	// Split each line and make sure input is valid
	parameters := strings.Split(text, "|")

	// Requests must be three parameters
	if len(parameters) != 3 {
		fmt.Printf("Only three parameters allowed per line (query, days, and limit, separated by '|'). Line %d has %d parameters.\n", lineNum, len(parameters))
		return SearchRequest{}, false
	}

	// The search term is the first value (index 0)
	// The number of days since published is the second value (index 1)
	// The amount of articles displayed (limit) is the third value (index 2)

	// Trim the leading and trailing spaces of each string
	query := strings.TrimSpace(parameters[0])
	daysStr := strings.TrimSpace(parameters[1])
	limit := strings.TrimSpace(parameters[2])

	// Days must be a number
	days, err := strconv.Atoi(daysStr)
	if err != nil || days <= 0 {
		fmt.Printf("The number of days must be a positive number! On Line %d, it is currently '%s'.\n", lineNum, parameters[1])
		return SearchRequest{}, false
	}

	// Convert the day number to an actual date (Ex: if days was 1, date would be today, if it was 2, date would be yesterday, etc...)
	date := time.Now().AddDate(0, 0, -(days - 1)).Format("2006-01-02")

	// Limit must be a number (but still will be put into the request as a string since it is put into a URL for API calls)
	limitVal, err := strconv.Atoi(limit)
	if err != nil || limitVal <= 0 {
		fmt.Printf("The limit must be a positive number! On Line %d, it is currently '%s'\n.", lineNum, parameters[2])
		return SearchRequest{}, false
	}

	// If request made it here, that means it is valid
	// Create the request and return success
	return SearchRequest{Query: query, Days: date, Limit: limit}, true
}
```

Figure 2.9: parseLine() (proj1.go)

This function will validate the input on each line in the file. The formal parameters include the entire line in the file, as well as the line number (since this program gets called for each line in a loop). The string gets split using the | separator, and validates the input, meaning it checks if there are three parameters, and if the days and limits are valid numbers. If the request is valid, it will create the SearchRequest structure with those parameters, and it will be successful.

`createDatabase()`

```go
133 // Creates the database using sqlite
134 func createDatabase() {
135   var err error
136 
137   // Open the database
138   db, err = sql.Open("sqlite", "./news_cache.db")
139   check(err)
140 
141   // Limit database connections to a single open and idle connection
142   db.SetMaxOpenConns(1)
143   db.SetMaxIdleConns(1)
144 
145   // Create the table (if this is the first time the program is run)
146   _, err = db.Exec(`
147     CREATE TABLE IF NOT EXISTS articles (
148       query TEXT NOT NULL,
149       days TEXT NOT NULL,
150       data TEXT NOT NULL,
151       PRIMARY KEY (query, days)
152     )
153   `)
154   check(err)
155 
156   // Allows concurrent reading and writing (has limited effect due to open/idle connection limit)
157   _, err = db.Exec("PRAGMA journal_mode=WAL;")
158   check(err)
159 }
```

Figure 2.10: createDatabase() (proj1.go)

This function creates the database if it doesn't already exist. I used a lot of different references to create my database, like [Database Access Tutorial], which teaches the basics of how to create a database. This is a similar article, but it specifically talks about SQLite, which is used to create my database [Using SQLite with Go25]. The database is used by opening news_cache.db, which is the volume creating in the `docker run` command explained earlier (Line 138). Something very important was to limit the database connections to avoid concurrency issues (Lines 142-143). Now, according to research, this doesn't 100% stop these issues from occurring [SQLite Concurrent Writing Performance16], but for this case, I think it was good enough since now there seems to be no errors regarding database locking. Then, the database creates a table (if it wasn't already created), that stores the query, the days (from the request), and the data (Lines 146-153). Then, to speed up database reads and writes by a little bit, I allowed concurrent reading and writing (Line 157), but it gets slowed down a bit before the open/idle connection limit. According to sources, this still has small performance benefits [SQLite Concurrent Writing Performance16] [Write-Ahead Logging (WAL)].

`loadFromDatabase()`

```
161 // Load current query from the Database, and return true if was found
162 func loadFromDatabase(req SearchRequest) (*NewsAPIResponse, bool) {
163 
164   // Query the table to check if database results can be used instead of using API
165   row := db.QueryRow(`
166     SELECT data FROM articles
167     WHERE query = ? AND days <= ?`,
168     req.Query, req.Days)
169 
170   // Store result from the query
171   var data string
172 
173   // If there were no results in the query, return to process request using API
174   err := row.Scan(&data)
175   if err != nil {
176     return nil, false
177   }
178 
179   // Store the JSON response
180   var response NewsAPIResponse
181 
182   // Attempt to unmarshal the JSON string from the database into the response struct.
183   err = json.Unmarshal([]byte(data), &response)
184   check(err)
185 
186   // If everything succeeds, return the response and true.
187   return &response, true
188 
189 }
```

Figure 2.11: loadFromDatabase() (proj1.go)

This function tries to load data from the database. Before processing a request, it checks if that request was in the database, and it will use the database data instead. Nothing really special going on here, except for formatting the JSON string from the database result to fit the `NewsAPIResponse` structure (Line 183). It encodes the regular string to a JSON string [JSON]. If a result came back from the database, it would return that response (Line 187).

`saveToDatabase()`

```go
// Save the response data to the database
func saveToDatabase(req SearchRequest, resp NewsAPIResponse) {

  // Convert the NewsAPIResponse struct to a JSON string for
    storage
  data, _ := json.Marshal(resp)

  // Adds a new row to the database with the given API data
  _, err := db.Exec(`
    INSERT OR REPLACE INTO articles (query, days, data)
    VALUES (?, ?, ?)`,
    req.Query, req.Days, string(data),
  )
  check(err)
}
```

Figure 2.12: saveToDatabase() (proj1.go)

This function saves the data from an API request to the database. This function gets called based off of data in the writeChan (explained later), so only API requests get saved to the database. Cache requests don't need to be saved since those results should've been saved by an earlier API request.

`processRequest()`

```go
func processRequest(request SearchRequest, apiKey string) {

  // Get query
  query := request.Query

  // Check the in-memory cache to see if request was asked
    previously
  cacheMu.RLock()
  mem, inCache := cache[query]
  cacheMu.RUnlock()

  // If it was asked (and current request has all results the
    cached request had)
  // Print the response based off of the map
  if inCache {
    cacheDate, _ := time.Parse("2006-01-02", mem.req.Days)
    requestDate, _ := time.Parse("2006-01-02", request.Days)

    if !cacheDate.After(requestDate) {
      printResponse(request, mem.resp, "CACHE")
      return
    }
  }
229   // IF NOT IN THE DATABASE OR THE CACHE, DO AN API CALL
230   // Makes sure spaces are handled if they are in the request
231   q := url.QueryEscape(request.Query)
232 
233   // Create the URL using fields from the request and the API Key
234   url := "https://newsapi.org/v2/everything?q=" + q + "&from=" +
        request.Days + "&sortBy=popularity&apiKey=" + apiKey
235 
236   // Make a HTTP GET request to this URL, returning an HTTP
        response
237   resp, err := http.Get(url)
238   check(err)
239 
240   // Uses HTTP response body to create a JSON Decoder
241   // Parses the JSON to fill the response structure
242   var response NewsAPIResponse
243   err = json.NewDecoder(resp.Body).Decode(&response)
244   check(err)
245 
246   // Closes once response is decoded
247   resp.Body.Close()
248 
249   // If GET request had an error, print the error message
250   if response.Status == "error" {
251     panic(response.Message)
252   }
253 
254   // Save the data to the database via the write channel
255   writeChan <- reqNresp{req: request, resp: response}
256 
257   // Save to in-memory cache if it has more data than previous
        cached query, or this is the first instance of that query
258   cacheMu.Lock()
259   cache[query] = &reqNresp{req: request, resp: response}
260   cacheMu.Unlock()
261 
262   // Print the response
263   printResponse(request, response, "API")
264 }
```

Figure 2.13: processRequest() (proj1.go)

This is a very important function, where all requests are processed (unless it was found in the database). First of all, before doing an API call, it checks if the query is in the cache (Lines 213-215). If it was in the cache, check if we can use the results in the cache. We cannot use results if the current request date is older than the date in the cache, because then the cache wouldn't have all the data we need. However, if the request date is newer, we can get the cache result and use that for our response (Lines 219-227).

If it wasn't in the database or the cache, it will do an API call. It will create the URL using the query and the correctly formatted date, as well as the user-supplied API key (Line 234). It will get a response (Line 237) and decode that into JSON format using the `NewsAPIResponse` structure (Lines 242-244). If there was an error with handling the request, this probably has to do with your API key, so end the program (Lines 250-252). If there wasn't, both the request and the response get put into the `reqNresp` structure, which then gets passed into the writeChan (Line 255). Channels are what we use to communicate between different goroutines [Channels]. Like mentioned previously, this channel is used to save the API request and result into the database. It also will save the result in the cache for the duration of this program runtime (Line 259). Then, the response will print.

`printResponse()`

```go
266 // Prints the response from the request
267 func printResponse(req SearchRequest, resp NewsAPIResponse, location string) {
268 
269   // Uses a string Builder to make sure all input prints out together at once
270   // This avoids concurrency issues
271   var sb strings.Builder
272 
273   // Parse requested limit
274   reqLimit, _ := strconv.Atoi(req.Limit)
275   articleLength := len(resp.Articles)
276 
277   // Display that request was processed
278   fmt.Fprintf(&sb, "\n--- USING: %s, RESULTS FOR QUERY: %s (Days=%s, Limit=%d) ---\n", location, req.Query, req.Days, reqLimit)
279 
280   // Keeps track of the minimum date in Time format
281   minDate, _ := time.Parse("2006-01-02", req.Days)
282 
283   // Keeps track of how many requests were printed
284   printed := 0
285 
286   // Print results
287   // For each of the top results, print information
288   for i := 0; i < articleLength && printed < reqLimit; i++ {
289     currentArticle := resp.Articles[i]
290 
291     // Don't show results older than this request if coming from CACHE
292     // Parse publishedAt key in RFC3339 format (if using cache and has a smaller day limit)
293     published, _ := time.Parse(time.RFC3339, currentArticle.PublishedAt)
294 
295     // Keep only the year, month, day (time will be 00:00:00 UTC)
296     publishedDate := time.Date(published.Year(), published.Month(), published.Day(), 0, 0, 0, 0, time.UTC)
```

```go
    // Skip articles older than requested date
    if publishedDate.Before(minDate) {
      continue
    }

    fmt.Fprintf(&sb, "ENTRY %d: %s\n", printed+1, currentArticle.
    Title)
    fmt.Fprintf(&sb, "PUBLISH DATE: %s\n", currentArticle.
    PublishedAt)
    fmt.Fprintf(&sb, "DESCRIPTION: %s\n", currentArticle.
    Description)
    fmt.Fprintf(&sb, "URL: %s\n", currentArticle.URL)
    fmt.Fprintln(&sb)

    printed++
  }

  // Print message if results were empty
  if printed == 0 {
    fmt.Fprintln(&sb, "\nNo articles matched the request...")
  }

  // Print the final built String
  fmt.Print(sb.String())
}
```

Figure 2.14: printResponse() (proj1.go)

This is the function that handles the printing. The entire output gets passed into a builder string variable named `sb`. This variable is used to print out all the output at once [String Builder25]. Because of how I set up the database and cache, there could be some results in the `newsAPIResponse` that is older than the request date. This was helpful for referring to the database or cache instead of the API, but it does hurt a little here. Now we have to make sure that we are skipping articles older than the requested date. Once all of the allowed articles are added to the string builder (due to the `limit`), it will print the results in the CLI.
`getQueryMutex()`

```go
// Gets the mutex for this query (so similar queries will need to
    wait until results are uploaded into cache)
func getQueryMutex(req SearchRequest) *sync.Mutex {
  queryMutexesMu.Lock()
  defer queryMutexesMu.Unlock()

  reqMutex, exists := queryMutexes[req.Query]

  // If query didn't exist, create a new Mutex in the map
  if !exists {
    mu := &sync.Mutex{}
    queryMutexes[req.Query] = &RequestMutex{req, mu}
    return mu
  }

  // Get the original cached request
  cachedReq := reqMutex.Request

  // If new request needs more data than that was cached (date is
    older), create a new Mutex
  if req.Days < cachedReq.Days {
    mu := &sync.Mutex{}
    queryMutexes[req.Query] = &RequestMutex{req, mu}
    return mu
  }

  // Otherwise, reuse existing mutex
  return reqMutex.Mutex
}
```

Figure 2.15: getQueryMutex() (proj1.go)

This function is used for requests to wait if a current request that is processing has the same name query as the new request. Basically, if any request that is currently processing has the same name as an incoming request, that incoming request would need to wait for that processing request to finish. This uses Mutexes, as described earlier, to wait for requests [Mutexes]. If the query is found in the `queryMutexes` map, and the date of this request is newer or equal to the date of the processing request, then it will reuse that existing mutex. This function will create new Mutexes if it is a new query, or if that query now is associated with an older date.

2.5 Main Program

Here I will be breaking down each part of main that I find worthy to explain, since a lot has already been explained in this Section.

```go
func main() {
  // Keep track of how long it takes to run this program
  start := time.Now()

  // Creates database and articles table (if it does not exist
    already)
  createDatabase()

  // Gets API key from environmental variables on CLI
  key := os.Getenv("NEWSAPI_KEY")

  // Gets the file path for the user prompt
  filePath := os.Getenv("FILE")

  // Gets the number of workers working in the worker pool
  workers := os.Getenv("WORKERS")
```

Figure 2.16: main() ENV VARS (proj1.go)

The main function starts off with keeping track of how long the program runs for, creating the database and table (if it hasn't been created already), and getting the env variables for the API KEY, the file path, and the number of workers for the worker pool. In our implementation, since we use Docker, those variables will be supplied on the command line when doing the `docker run` command, as explained in Section 1.

```
365  // Makes sure user supplied their API Key
366  if key == "" {
367    fmt.Println("Please supply API Key to run the program. \nUsing Docker: \n " +
368      "docker run --rm -e NEWSAPI_KEY='apiKey' -e FILE='file.txt' -e WORKERS='num' -v news_cache_volume:/app proj1")
369    return
370  }
371
372  // Remove quotes from CLI input (if it exists)
373  key = strings.Trim(key, "'\"")
374  filePath = strings.Trim(filePath, "'\"")
375  workers = strings.Trim(workers, "'\"")
376
377  // Default number of worker if input wasn't valid
378  DEFAULT_NUM_WORKERS := 10
379
380  // Makes sure number of workers input is valid
381  numWorkers, err := strconv.Atoi(workers)
382  if err != nil || numWorkers <= 0 {
383    fmt.Printf("Number of workers needs to be an integer! It is currently %s. Defaulting to %d Workers.\n", workers, DEFAULT_NUM_WORKERS)
384    numWorkers = DEFAULT_NUM_WORKERS
385  }
```

Figure 2.17: main() Validating Data (proj1.go)

This section of the function will validate all of the environmental variables. If the API key field is empty, it will throw an error and tell the user how to use the program (Lines 366-370). The program will remove all quotation marks that may be with the files, just to make sure that the results don't contain the quotes used in the CLI (Lines 373-375). Then, the number of workers in the worker pool is validated. It needs to be a positive integer. If it isn't, it doesn't end the program, but just sets it to the default number, which is 10 (Lines 381-385). The file path gets validated later in the program, and we will see it when we are about to read the file.

```go
387 // Channel used to write safety into the database
388 writeChan = make(chan reqNresp)
389
390 // Waitgroup that waits for all entries to be added to the database
391 var writeWG sync.WaitGroup
392
393 // Goroutine that makes sure all writes happen in the database
394 for range numWorkers {
395   writeWG.Go(func() {
396     for w := range writeChan {
397       saveToDatabase(w.req, w.resp)
398     }
399   })
400 }
401
402 // Create a channel of requests
403 requestsChan := make(chan SearchRequest)
404
405 // Waitgroup that waits for all results to be processed before program ends
406 var resultsWG sync.WaitGroup
407
408 // Goroutine that collects data from the request channel
409 // Worker pool created for parallel API Requests
410 for range numWorkers {
411   resultsWG.Go(func() {
412     // Will wait until data gets put into the requests channel
413     for req := range requestsChan {
414
415       // Checks if result is already in the database
416       results, inDB := loadFromDatabase(req)
417       if inDB {
418         printResponse(req, *results, "DATABASE")
419       } else {
420         // Only requests with the same query (and a smaller or equal date and limit) will be locked
421         mu := getQueryMutex(req)
422
423         mu.Lock()
424         processRequest(req, key)
425         mu.Unlock()
426       }
427     }
428   })
429 }
```

Figure 2.18: main() Channels and Waitgroups (proj1.go)

After the input has been validated, the write channel is defined (Line 388). We saw this channel earlier when saving results from the API call. Then, a waitgroup is created. A waitgroup is yet another way to wait for multiple goroutines to finish [WaitGroups]. In this case, it is used for waiting for all database writes to occur before ending the program, which we will see later. A worker pool is started (Line 394), and a goroutine is also started with the waitgroup. A goroutine is a lightweight thread of execution that runs parallel with the main thread [Goroutines]. This is how concurrency works in my program. This causes the writes to the database to happen as fast as possible without causing the database to lock (Lines 394-400). Whenever any data gets passed into `writeChan`, this goroutine will cause it to write to the database.

After this goroutine starts, a channel of requests get made (Line 403). When valid requests are made, they will be added to this channel. Then, it will process these requests in the goroutine created by another waitgroup (Line 406). Again, the reason why we are using a waitgroup is so we make sure all requests are processed before ending the program. Another worker pool is created (Line 410), and a goroutine is started (Line 411). For each request, it will look it up in the database. If it is there, print the response from the database, which processes the request (Lines 416-419). If it is not in the database, get the Mutex for this query and process this request. If the getQueryMutex returns a Mutex that is already locked, it will need to wait before processing the request (Lines 419-425).

```
431  // Make sure file path for user input is correct
432  file, err := os.Open(filePath)
433  check(err)
434
435  // Close the file once the program is complete
436  defer file.Close()
437
438  // A waitgroup used to wait for all the goroutines launched to
       finish when reading the lines from the file
439  var fileWG sync.WaitGroup
440
441  // Create scanner to read file
442  scanner := bufio.NewScanner(file)
443
444  // Store line number of request
445  lineNumber := 0
446
447  // Reads file line by line concurrently (using goroutines and
       waitgroups)
448  for scanner.Scan() {
449    // Get text on current line
450    text := scanner.Text()
451
452    // Make a copy of the line number after its incrementation for
       better error messages
453    lineNumber++
454    currentLine := lineNumber
456     // Each of these goroutines work concurrently
457     fileWG.Go(func() {
458
459       // Validate the current request
460       req, success := parseLine(text, currentLine)
461
462       // If it is valid, send to requests channel for further
    processing
463       if success {
464         requestsChan <- req
465       }
466     })
467   }
```

Figure 2.19: main() File Reading (proj1.go)

This is where file handling occurs. Reading is a basic task that is needed for this program [Reading Files]. The file gets opened and validated (Line 432-433), and it will close when the program finishes (Line 436). Another waitgroup is created to make sure all lines of the file have been processed before the program ends (Line 439). Then, a scanner is created. This Scanner is used to read the file line by line [Line Filters]. The reads do have to happen sequentially (lines 448-454), unfortunately, which slows the program down a little, but it is essential for good error messages. Even though these reads happen sequentially, the parsing happens concurrently in a goroutine (Line 457). It concurrently parses and validates each request (Line 460) and if it was successful, add the request to the channel to then get processed (Line 463-465).

```
469   // Checks if there was an error reading the file
470   check(scanner.Err())
471
472   // Waits for all lines to be read
473   fileWG.Wait()
474
475   // If there were no errors, close the request channel
476   close(requestsChan)
477
478   // Waits for all requests to be processed
479   resultsWG.Wait()
480
481   close(writeChan)
482
483   // Waits for all writes to be processed in the database
484   writeWG.Wait()
485
486   // Once all lines of the file are read and the results are
    processed, the program can end
487   fmt.Printf("\nProgram took %s to run.\n", time.Since(start))
488 }
```

Figure 2.20: main() Ending Program (proj1.go)

The program can't just end, as it needs to wait for all processes in the waitgroups and channels to finish. The program waits for all lines to be read, then it closes the request channel (Lines 473-476). It waits for all requests to be processed, then it closes the write channel (Lines 479-481). It waits for all writes to be processed in the database, which is important (Line 484). Then, finally, it prints out the total time it took to run the program (Line 487).

3 Conclusion

3.1 My Thoughts

I am proud to say that I didn't use any AI to generate any of the code in this program. I wanted to do this legit, and if I had any questions, I would Google it (as shown in all of the references I have). I spent the first couple of weeks studying how goroutines, channel, worker pools, and waitgroups worked before actually building up my program.

Docker did take a little more time to try to understand, as I've dealt with it before but haven't actually had to create Images, so the process with the Dockerfile was a little difficult for me. However, there is a lot of documentation on how to create the file, especially when dealing with Golang, so after understanding how it works, and after going to classes that reviewed the Dockerfile, I found a way to make it work. The hardest part was allowing the runtime build be built from scratch, because I had to copy over the certification so HTTP could be used in the first place.

Overall this was a major learning experience for me, especially when dealing with cloud computing software principles. I learned how to write a program that runs concurrently and fast. It was an amazing, but long, experience.

3.2 Future Work

Here is some things I would change in the future about my code for this project:

- Only writing to the database at one time after all of the requests have been processed, since it will read from the cache anyways so there is no point in rushing. Could be faster.
- Instead of calling the `panic` function whenever there is an error, handle it a different way. It is a helpful debugging tool, but all of that information being spewed out like that may not be the best idea.
- Organize the code just a little bit better, as it can be confusing without explicit documentation.

Below are my references for this Project.

References

[CA Certificates24]	Docker Documentation. *CA Certificates*. 2024. URL: https://docs.docker.com/engine/network/ca-certs.
[Channels]	Mark McGranaghan. *Channels*. URL: https://gobyexample.com/channels.
[Database Access Tutorial]	Go Documentation. *Database Access Tutorial*. URL: https://go.dev/doc/tutorial/database-access.
[Golang Build Images Guide24]	Docker Documentation. *Golang Build Images Guide*. 2024. URL: https://docs.docker.com/guides/golang/build-images.
[Goroutines]	Mark McGranaghan. *Goroutines*. URL: https://gobyexample.com/goroutines.
[JSON]	Mark McGranaghan. *JSON*. URL: https://gobyexample.com/json.
[Line Filters]	Mark McGranaghan. *Line Filters*. URL: https://gobyexample.com/line-filters.
[modernc.org/sqlite Package25]	Modernc Documentation. *modernc.org/sqlite Package*. 2025. URL: https://pkg.go.dev/modernc.org/sqlite#section-readme.
[Mutexes]	Mark McGranaghan. *Mutexes*. URL: https://gobyexample.com/mutexes.
[NewsAPI Documentation25]	NewsAPI. *NewsAPI Documentation*. 2025. URL: https://newsapi.org/docs/endpoints/everything.
[Reading Files]	Mark McGranaghan. *Reading Files*. URL: https://gobyexample.com/reading-files.
[SQLite Concurrent Writing Performance16]	StackOverflow. *SQLite Concurrent Writing Performance*. 2016. URL: https://stackoverflow.com/questions/35804884/sqlite-concurrent-writing-performance.

[SSL Certificate Location on Unix/Linux09]	ServerFault. *SSL Certificate Location on Unix/Linux*. 2009. URL: https://serverfault.com/questions/62496/ssl-certificate-location-on-unix-linux.
[String Builder25]	Go Documentation. *String Builder*. 2025. URL: https://pkg.go.dev/strings#Builder.
[UPX Source17]	gruebel. *UPX Source*. 2017. URL: https://github.com/gruebel/docker-upx.
[Using SQLite with Go25]	Twilio Blog. *Using SQLite with Go*. 2025. URL: https://www.twilio.com/en-us/blog/developers/community/use-sqlite-go.
[WaitGroups]	Mark McGranaghan. *WaitGroups*. URL: https://gobyexample.com/waitgroups.
[Worker Pools]	Mark McGranaghan. *Worker Pools*. URL: https://gobyexample.com/worker-pools.
[Write-Ahead Logging (WAL)]	SQLite Documentation. *Write-Ahead Logging (WAL)*. URL: https://sqlite.org/wal.html.