

Task 1.1:

```
type Book struct {
    Title      string `json:"title"`
    Price      string `json:"price"`
    Rating     string `json:"rating"`
    Availability string `json:"availability"`
    ImageURL   string `json:"image_url"`
}
```

This part defines a Book struct, which holds the essential details of each book: title, price, rating, availability, and image URL.

It also makes use of JSON tags to simplify the process of serializing the data into JSON format.

```

func scrapeBooks(url string) ([]Book, error) {
    resp, err := http.Get(url)
    if err != nil {
        return nil, fmt.Errorf("failed to fetch page: %w", err)
    }
    defer resp.Body.Close()

    if resp.StatusCode != http.StatusOK {
        return nil, fmt.Errorf("bad status code: %d", resp.StatusCode)
    }

    body, err := io.ReadAll(resp.Body)
    if err != nil {
        return nil, fmt.Errorf("failed to read body: %w", err)
    }

    doc, err := html.Parse(bytes.NewReader(body))
    if err != nil {
        return nil, fmt.Errorf("failed to parse HTML: %w", err)
    }

    var books []Book

    // Find all <article class="product_pod">
    var walk func(*html.Node)
    walk = func(n *html.Node) {
        if n.Type == html.ElementNode && n.Data == "article" {
            for _, attr := range n.Attr {
                if attr.Key == "class" && strings.Contains(attr.Val, "product_pod") {
                    book := extractBookData(n)
                    books = append(books, book)
                    break
                }
            }
            for c := n.Firstchild; c != nil; c = c.NextSibling {
                walk(c)
            }
        }
        walk(doc)
    }

    return books, nil
}

```

This function is responsible for gathering the book data. Here's how it works:

- It sends an HTTP GET request to the target website URL.
- Once the page is fetched, it uses the `golang.org/x/net/html` package to parse the HTML content.

- The function then walks through the DOM tree, looking specifically for all `<article class="product_pod">` elements. Each of these represents a single book on the page.
- Finally, it returns a slice of Book objects, each containing the extracted information for one book.

```
func extractBookData(n *html.Node) Book {
    book := Book{}

    var walk func(*html.Node)
    walk = func(n *html.Node) {
        if n.Type == html.ElementNode {
            switch n.Data {
                case "h3":
                    for c := n.FirstChild; c != nil; c = c.NextSibling {
                        if c.Type == html.ElementNode && c.Data == "a" {
                            for _, attr := range c.Attr {
                                if attr.Key == "title" {
                                    book.Title = strings.TrimSpace(attr.Val)
                                }
                            }
                        }
                }
                case "p":
                    for _, attr := range n.Attr {
                        // price
                        if attr.Key == "class" && strings.Contains(attr.Val, "price_color") {
                            if n.FirstChild != nil {
                                book.Price = strings.TrimSpace(n.FirstChild.Data)
                            }
                        }
                        // rating
                        if attr.Key == "class" && strings.Contains(attr.Val, "star-rating") {
                            classes := strings.Split(attr.Val, " ")
                            if len(classes) > 1 {
                                book.Rating = strings.TrimSpace(classes[1])
                            }
                        }
                    }
                    // availability
                    if attr.Key == "class" && strings.Contains(attr.Val, "availability") {
                        book.Availability = strings.TrimSpace(extractText(n))
                    }
                }
                case "img":
                    for _, attr := range n.Attr {
                        if attr.Key == "src" {
                            book.ImageURL = "http://books.toscrape.com/" + strings.TrimPrefix(attr.Val, "../")
                        }
                    }
            }
            for c := n.FirstChild; c != nil; c = c.NextSibling {
                walk(c)
            }
        }
    walk(n)

    return book
}
```

For each book article, this function pulls out the following details:

- Title: Extracted from the title attribute inside the <h3><a> tag.
- Price: Located within the <p class="price_color"> element.
- Rating: Found in the <p class="star-rating"> class (e.g., "Three", "Five", etc.).
- Availability: Taken from the <p class="availability"> element.
- Image URL: Constructed by extracting the src attribute from the tag, and then prepending the base URL to form the complete link.

```
// Extract text content (trimmed)
func extractText(n *html.Node) string {
    if n.Type == html.TextNode {
        return strings.TrimSpace(n.Data)
    }
    var text string
    for c := n.FirstChild; c != nil; c = c.NextSibling {
        text += extractText(c)
    }
    return strings.TrimSpace(text)
}
```

`extractText()`: This function recursively extracts text content from HTML nodes.

```
// Save books to JSON file
func saveBooksToJson(books []Book, filename string) error {
    data, err := json.MarshalIndent(books, "", " ")
    if err != nil {
        return fmt.Errorf("failed to marshal JSON: %w", err)
    }
    return os.WriteFile(filename, data, 0644)
}
```

Converts price strings (e.g., "£51.77") to float64 for calculations

```
func priceToFloat(price string) float64 {
    p := strings.TrimPrefix(price, "£")
    f, _ := strconv.ParseFloat(p, 64)
    return f
}
```

Marshals the book data to JSON format and saves to file

```
func main() {
    url := "http://books.toscrape.com/catalogue/page-1.html"
    fmt.Printf("Scraping books from: %s\n", url)

    books, err := scrapeBooks(url)
    if err != nil {
        fmt.Printf("Error: %v\n", err)
        return
    }

    fmt.Printf("Found %d books\n\n", len(books))

    // Print first book as sample
    if len(books) > 0 {
        fmt.Println("Book 1:")
        fmt.Printf("    Title: %s\n", books[0].Title)
        fmt.Printf("    Price: %s\n", books[0].Price)
        fmt.Printf("    Rating: %s\n", books[0].Rating)
        fmt.Printf("    Availability: %s\n\n", books[0].Availability)
    }
}
```

- Scrapes books from page 1
- Prints the first book as a sample

```

// calculate average price
var total float64
for _, b := range books {
    total += priceToFloat(b.Price)
}
avg := total / float64(len(books))

fmt.Printf("Summary:\n")
fmt.Printf(" Total books: %d\n", len(books))
fmt.Printf(" Average price: £%.2f\n\n", avg)

// Save to JSON
if err := saveBooksToJson(books, "books.json"); err != nil {
    fmt.Printf("Failed to save JSON: %v\n", err)
    return
}

fmt.Printf("Saved %d books to books.json\n", len(books))
}

```

- Calculates and displays the average book price
- Saves all book data to books.json

Output:

```
Task1.1 > {} books.json > ...
1  [
2  {
3      "title": "A Light in the Attic",
4      "price": "£51.77",
5      "rating": "Three",
6      "availability": "In stock",
7      "image_url": "http://books.toscrape.com/media/cache/2c/da/2cdad67c44b002e7ead0cc35e"
8  },
9  {
10     "title": "Tipping the Velvet",
11     "price": "£53.74",
12     "rating": "One",
13     "availability": "In stock",
14     "image_url": "http://books.toscrape.com/media/cache/26/0c/260c6ae16bce31c8f8c95dad0"
15 },
16  {
17      "title": "Soumission",
18      "price": "£50.10",
19      "rating": "One",
20      "availability": "In stock",
21      "image_url": "http://books.toscrape.com/media/cache/3e/ef/3eff99c9d9adef34639f5106e"
22 },
23  {
24      "title": "Sharp Objects",
25      "price": "£47.82",
26      "rating": "Four",
27      "availability": "In stock",
28      "image_url": "http://books.toscrape.com/media/cache/32/51/3251cf3a3412f53f339e42cac"
29 }
```

Task 2.1:

```

type TMDBSearchResponse struct {
    Page      int `json:"page"`
    Results []struct {
        ID          int     `json:"id"`
        Title       string  `json:"title"`
        Overview    string  `json:"overview"`
        ReleaseDate string  `json:"release_date"`
        VoteAverage float64 `json:"vote_average"`
        GenreIDs   []int   `json:"genre_ids"`
        PosterPath  string  `json:"poster_path"`
    } `json:"results"`
    TotalResults int    `json:"total_results"`
}

type Movie struct {
    ID          int     `json:"id"`
    Title       string  `json:"title"`
    Overview    string  `json:"overview"`
    ReleaseDate string  `json:"release_date"`
    Rating      float64 `json:"rating"`
    Genres      []string `json:"genres"`
    PosterURL   string  `json:"poster_url"`
}

type TMDBClient struct {
    APIKey      string
    baseURL     string
    HTTPClient  *http.Client
    genreMap    map[int]string
}

```

- TMDBSearchResponse: Represents the API response structure
- Movie: Custom structure to store processed movie information
- TMDBClient: Client wrapper with API key, HTTP client, and genre mapping

```

func NewTMDBClient(apiKey string) *TMDBClient {
    return &TMDBClient{
        APIKey: apiKey,
        BaseURL: TMDBBaseURL,
        HTTPClient: &http.Client{
            Timeout: 15 * time.Second,
        },
        GenreMap: make(map[int]string),
    }
}

```

NewTMDBClient(): Constructor that initializes the client with a 15-second timeout

```

func (c *TMDBClient) loadGenres() error {
    endpoint := fmt.Sprintf("%s/genre/movie/list?api_key=%s", c.BaseURL, c.APIKey)
    resp, err := c.HTTPClient.Get(endpoint)
    if err != nil {
        return fmt.Errorf("failed to fetch genres: %w", err)
    }
    defer resp.Body.Close()

    if resp.StatusCode != http.StatusOK {
        body, _ := io.ReadAll(resp.Body)
        return fmt.Errorf("TMDB error: %s", body)
    }

    var data struct {
        Genres []struct {
            ID   int `json:"id"`
            Name string `json:"name"`
        } `json:"genres"`
    }

    if err := json.NewDecoder(resp.Body).Decode(&data); err != nil {
        return fmt.Errorf("failed to decode genres: %w", err)
    }

    for _, g := range data.Genres {
        c.GenreMap[g.ID] = g.Name
    }

    fmt.Printf("Loaded %d genres\n\n", len(c.GenreMap))
    return nil
}

```

loadGenres(): Fetches the genre list from TMDB and creates a mapping from genre IDs to names

```
func (c *TMDBClient) searchMovies(query string) ([]Movie, error) {
    escaped := url.QueryEscape(query)
    endpoint := fmt.Sprintf("%s/search/movie?api_key=%s&query=%s", c.BaseURL, c.APIKey, escaped)

    resp, err := c.HTTPClient.Get(endpoint)
    if err != nil {
        return nil, fmt.Errorf("failed to search movies: %w", err)
    }
    defer resp.Body.Close()

    if resp.StatusCode != http.StatusOK {
        body, _ := io.ReadAll(resp.Body)
        return nil, fmt.Errorf("TMDB error: %s", body)
    }

    var tmdbResp TMDBSearchResponse
    if err := json.NewDecoder(resp.Body).Decode(&tmdbResp); err != nil {
        return nil, fmt.Errorf("failed to parse search response: %w", err)
    }

    var movies []Movie
    for _, r := range tmdbResp.Results {
        m := Movie{
            ID:          r.ID,
            Title:       r.Title,
            Overview:   r.Overview,
            ReleaseDate: r.ReleaseDate,
            Rating:     r.VoteAverage,
            PosterURL:  "https://image.tmdb.org/t/p/w500" + r.PosterPath,
        }
        for _, gid := range r.GenreIDs {
            if name, ok := c.GenreMap[gid]; ok {
                m.Genres = append(m.Genres, name)
            }
        }
        movies = append(movies, m)
    }
    return movies, nil
}

• searchMovies():
```

- URL-encodes the search query
- Makes API request to the search endpoint
- Converts TMDB response to custom Movie format
- Maps genre IDs to genre names using the pre-loaded genre map

```

func (c *TMDBClient) getMovieDetails(id int) (*Movie, error) {
    endpoint := fmt.Sprintf("%s/movie/%d?api_key=%s", c.BaseURL, id, c.APIKey)
    resp, err := c.HTTPClient.Get(endpoint)
    if err != nil {
        return nil, fmt.Errorf("failed to fetch movie details: %w", err)
    }
    defer resp.Body.Close()

    if resp.StatusCode != http.StatusOK {
        body, _ := io.ReadAll(resp.Body)
        return nil, fmt.Errorf("TMDB error: %s", body)
    }

    var data struct {
        ID         int      `json:"id"`
        Title      string   `json:"title"`
        Overview   string   `json:"overview"`
        ReleaseDate string   `json:"release_date"`
        VoteAverage float64  `json:"vote_average"`
        Genres     []struct {
            ID   int      `json:"id"`
            Name string   `json:"name"`
        } `json:"genres"`
        PosterPath string   `json:"poster_path"`
    }

    if err := json.NewDecoder(resp.Body).Decode(&data); err != nil {
        return nil, fmt.Errorf("failed to decode details: %w", err)
    }

    var genreNames []string
    for _, g := range data.Genres {
        genreNames = append(genreNames, g.Name)
    }

    movie := &Movie{
        ID:         data.ID,
        Title:      data.Title,
        Overview:   data.Overview,
        ReleaseDate: data.ReleaseDate,
        Rating:     data.VoteAverage,
        Genres:     genreNames,
        PosterURL:  "https://image.tmdb.org/t/p/w500" + data.PosterPath,
    }
    return movie, nil
}

```

getMovieDetails(): Fetches detailed information for a specific movie by ID

```

func saveMoviesToJson(movies []Movie, filename string) error {
    data, err := json.MarshalIndent(movies, "", "  ")
    if err != nil {
        return fmt.Errorf("failed to marshal JSON: %w", err)
    }
    return os.WriteFile(filename, data, 0644)
}

```

Marshals movie slice to formatted JSON

```

func main() {
    apiKey := "33be097c32c7ec8df2864b26e113d643"
    client := NewTMDBClient(apiKey)

    fmt.Println("Loading movie genres...")
    if err := client.loadGenres(); err != nil {
        fmt.Printf("Error loading genres: %v\n", err)
        return
    }

    query := "inception"
    fmt.Printf("Searching for: %s\n", query)
    movies, err := client.searchMovies(query)
    if err != nil {
        fmt.Printf("Search error: %v\n", err)
        return
    }

    fmt.Printf("Found %d movies\n\n", len(movies))

    if len(movies) > 0 {
        fmt.Println("Movie 1:")
        m := movies[0]
        fmt.Printf(` ID: %d\n Title: %s\n Release Date: %s\n Rating: %.1f/10\n
Genres: %v\n Overview: %s...`+"\n", m.ID, m.Title, m.ReleaseDate, m.Rating, m.Genres, m.Overview)
    }

    if err := saveMoviesToJson(movies, "tmdb_results.json"); err != nil {
        fmt.Printf("Error saving JSON: %v\n", err)
        return
    }

    fmt.Printf("Saved %d movies to tmdb_results.json\n", len(movies))
}

```

- Creates TMDB client with API key

- Loads genre mappings
 - Searches for movies with query "inception"
 - Displays the first movie's details
 - Saves all results to tmdb_results.json

Output:

```
1 < [ {  
2   "id": 27205,  
3   "title": "Inception",  
4   "overview": "Cobb, a skilled thief who commits corporate espionage by infiltrating the subconscious of",  
5   "release_date": "2010-07-15",  
6   "rating": 8.37,  
7   "genres": [  
8     "Action",  
9     "Science Fiction",  
10    "Adventure"  
11  ],  
12  "poster_url": "https://image.tmdb.org/t/p/w500/ljsZTbVsrQ5qZgWeep2B1QiDKuh.jpg"  
13 },  
14 {  
15   "id": 1370380,  
16   "title": "Syndrome Halla, the Inception of Croatian Professional Film Born to Die",  
17   "overview": "One hundred years after the invention of film, Croatian film icon Mr. Fulir, who serves as",  
18   "release_date": "2017-01-01",  
19   "rating": 0,  
20   "genres": [  
21     "Documentary",  
22     "Drama"  
23   ],  
24   "poster_url": "https://image.tmdb.org/t/p/w500"  
25 },  
26 {  
27   "id": 613092,  
28   "title": "The Crack: Inception",  
29   "overview": "Madrid, Spain, 1975; shortly after the end of the Franco dictatorship. Six months after the",  
30   "release_date": "2019-10-04",  
31   "rating": 6.7,  
32   "genres": [  
33     "Drama",  
34     "Thriller"  
35   ],  
36   "poster_url": "https://image.tmdb.org/t/p/w500/kzgPu2CMxBr4YZZxC10ff4cUfR9.jpg"  
37 },  
38 {  
39   "id": 250845,  
40   "title": "WNA The Inception",  
41   "overview": "The first World Wrestling Allstars pay per view, live from Sydney, Australia! A tournament"
```

Task 2.2:

MovieInfo

- Unified movie data structure across all sources

- Fields: ID, Title, Director, Year, Description, Genres, Rating, Duration, Source, LastUpdated
- More comprehensive than the Movie struct from Task 2.1

MovieSource (interface)

- Defines contract for all movie data sources
- Methods: GetMovies(query, limit) and GetName()

TMDBSource

- Implements MovieSource for TMDB API
- Wraps the TMDBClient from previous code
- Converts Movie to MovieInfo format

MockScraperSource

- Implements MovieSource for simulated scraped data
- Returns fake movie data for testing
- Demonstrates how to add additional sources

MovieAggregator

- Combines multiple MovieSource implementations
- Orchestrates concurrent queries and result merging

Key Functions

NewTMDBSource(apiKey) *TMDBSource

- Creates TMDB source wrapper
- Initializes internal TMDB client

TMDBSource.GetMovies(query, limit) ([]MovieInfo, error)

- Loads genres if needed
- Searches TMDB using the client
- Converts results to MovieInfo format
- Extracts year from release date string

- Returns limited results

MockScraperSource.GetMovies(query, limit) ([]MovieInfo, error)

- Generates mock movie data based on query
- Returns predefined fake results for testing
- Useful for development without real scraper

NewMovieAggregator(sources...) *MovieAggregator

- Creates aggregator with variable number of sources
- Uses variadic parameter for flexibility

MovieAggregator.Search(query, limitPerSource) ([]MovieInfo, error)

- **Main aggregation function**
- Queries all sources **concurrently** using goroutines
- Each source runs in separate goroutine with go func()
- Uses sync.Mutex to protect shared allMovies slice
- Uses sync.WaitGroup to wait for all goroutines to complete
- Deduplicates results after collection
- Returns merged, deduplicated movie list

deduplicateMovies(movies) []MovieInfo

- Removes duplicate movies based on title similarity
- For each movie, finds all similar titles ($\geq 80\%$ similarity)
- Merges duplicate data:
 - Keeps highest rating
 - Combines all genres
 - Fills in missing director/duration
- Marks duplicates as "used" to skip them
- Sorts final results by rating (descending)

calculateSimilarity(title1, title2) float64

- Returns similarity score between two titles (0.0 to 1.0)
- Normalizes titles (lowercase, trim)
- Removes punctuation (:, -, ,)
- Calculates word overlap percentage
- Returns 0.9 if one title contains the other
- Uses word matching for similarity calculation
- Ignores short words (≤ 2 chars)

generateReport(movies)

- Prints comprehensive statistics:
 - Total movies found
 - Count by source
 - Top 10 genres by frequency
 - Average rating across all movies
- Sorts genres by count before displaying

saveToJson(movies, filename) error

- Marshals movies to pretty-printed JSON
- Writes to file with 0644 permissions
- Returns formatted error if fails

main()

- Creates aggregator with TMDB + mock scraper
- Searches for "spider-man" with limit of 10 per source
- Displays top 5 deduplicated results with details
- Generates statistical report
- Saves all results to "aggregated_movies.json"

Output:

```
1 ∨ [
2 ∨ {
3   "id": "tmdb-569094",
4   "title": "Spider-Man: Across the Spider-Verse",
5   "year": 2023,
6   "description": "After reuniting with Gwen Stacy, Brooklyn's full-time, friendly neighborhood Spider-Man",
7   "genres": [
8     "Action",
9     "Adventure",
10    "Animation",
11    "Science Fiction"
12  ],
13  "rating": 8.396,
14  "source": "TMDB",
15  "last_updated": "2025-11-06T23:14:35+07:00"
16 },
17 ∨ {
18   "id": "scraper-1",
19   "title": "Spider-Man: The Beginning",
20   "director": "John Director",
21   "year": 2019,
22   "description": "An epic tale about spider-man that captivated audiences worldwide.",
23   "genres": [
24     "Action",
25     "Adventure",
26     "Drama",
27     "Fantasy",
28     "Horror",
29     "Mystery",
30     "Science Fiction"
31  ],
32  "rating": 8.2,
33  "duration_minutes": 132,
34  "source": "MovieScraper",
35  "last_updated": "2025-11-06T23:14:34+07:00"
36 },
37 ∨ {
38   "id": "tmdb-634649",
39   "title": "Spider-Man: No Way Home",
40   "year": 2021,
41   "description": "Peter Parker is unmasked and no longer able to separate his normal life from the high-st
```

Task 3.1:

```
type Book struct {
    Title  string `json:"title"`
    Price  string `json:"price"`
    Rating string `json:"rating"`
    URL    string `json:"url"`
}

type ScraperStats struct {
    PagesScraped int
    BooksFound   int
    Errors       int
    StartTime    time.Time
    EndTime      time.Time
}
```

Book Structure

Represents a book with title, price, rating, and URL. JSON tags define how these fields appear when exported, making each book a simple, consistent object for storage, sharing, or processing.

ScraperStats Structure

Tracks scraping performance, including pages scraped, total books, errors, and start/end times, enabling efficiency measurement, issue detection, and result summarization.

```
func fetchPage(pageURL string) (*html.Node, error) {
    resp, err := http.Get(pageURL)
    if err != nil {
        return nil, err
    }
    defer resp.Body.Close()

    doc, err := html.Parse(resp.Body)
    if err != nil {
        return nil, err
    }
    return doc, nil
}
```

fetchPage Function

Purpose: Downloads the HTML content from a given URL and parses it into a tree

structure.

How it works:

1. Sends an HTTP GET request to the URL.
2. Reads the response body.
3. Parses the HTML into a structured node tree.
4. Returns the parsed document, or an error if the HTTP request or parsing fails.

This function is the foundation of the scraper, because all further processing relies on a correctly parsed HTML document.

```
func extractBooks(doc *html.Node, baseURL string) []Book {
    var books []Book
    var f func(*html.Node)
    f = func(n *html.Node) {
        if n.Type == html.ElementNode && n.Data == "article" {
            for _, a := range n.Attr {
                if a.Key == "class" && strings.Contains(a.Val, "product_pod") {
                    var b Book

                    // Find title and relative link
                    findLink := func(n *html.Node) {
                        if n.Type == html.ElementNode && n.Data == "a" {
                            for _, a := range n.Attr {
                                if a.Key == "title" {
                                    b.Title = a.Val
                                }
                                if a.Key == "href" {
                                    link, _ := url.JoinPath(baseURL, a.Val)
                                    b.URL = link
                                }
                            }
                        }
                    }

                    // Find price and rating
                    findPrice := func(n *html.Node) {
                        if n.Type == html.ElementNode && n.Data == "p" {
                            for _, a := range n.Attr {
                                if a.Key == "class" && strings.Contains(a.Val, "price_color") && n.FirstChild != nil {
                                    b.Price = n.FirstChild.Data
                                }
                                if a.Key == "class" && strings.Contains(a.Val, "star-rating") {
                                    b.Rating = strings.TrimPrefix(a.Val, "star-rating ")
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

```

// Walk inside <article>
var inner func(*html.Node)
inner = func(c *html.Node) {
    findLink(c)
    findPrice(c)
    for child := c.FirstChild; child != nil; child = child.NextSibling {
        inner(child)
    }
}
inner(n)

if b.Title != "" {
    books = append(books, b)
}

for c := n.FirstChild; c != nil; c = c.NextSibling {
    f(c)
}
f(doc)
return books
}

```

extractBooks Function

Purpose: Extracts detailed information about each book from the HTML document.

How it works:

1. Walks through the HTML node tree recursively.
2. Searches for <article> elements with the class "product_pod".
3. Inside each article, it finds:
 - o **Title and URL** from <a> tags
 - o **Price** from <p> tags with class "price_color"
 - o **Rating** from <p> tags with class "star-rating"
4. Builds absolute URLs by combining the base URL with relative paths.
5. Adds only complete book entries (those with a title) to the results list.

Key techniques include nested helper functions for clean extraction (findLink, findPrice, inner) and recursive tree traversal to ensure no book is missed.

```

func getNextPageURL(doc *html.Node, baseURL string) (string, bool) {
    var nextURL string
    var found bool

    var f func(*html.Node)
    f = func(n *html.Node) {
        if n.Type == html.ElementNode && n.Data == "li" {
            for _, a := range n.Attr {
                if a.Key == "class" && a.Val == "next" {
                    // find <a> inside it
                    for c := n.FirstChild; c != nil; c = c.NextSibling {
                        if c.Type == html.ElementNode && c.Data == "a" {
                            for _, ha := range c.Attr {
                                if ha.Key == "href" {
                                    nextURL, _ = url.JoinPath(baseURL, ha.Val)
                                    found = true
                                    return
                                }
                            }
                        }
                    }
                }
            }
            for c := n.FirstChild; c != nil && !found; c = c.NextSibling {
                f(c)
            }
        }
        f(doc)
        return nextURL, found
    }
}

```

getNextPageURL Function

Purpose: Identifies the URL of the next page in a paginated catalog.

How it works:

1. Looks for elements with class "next".
2. Finds the <a> tag inside the list item.
3. Extracts the href attribute.
4. Converts the relative path to an absolute URL.
5. Returns the URL and a flag indicating whether a next page exists.

This function allows the scraper to continue automatically through multiple pages without hardcoding URLs.

```

func scrapePaginatedBooks(baseURL string, maxPages int) ([]Book, *ScraperStats, error) {
    stats := &ScraperStats{StartTime: time.Now()}
    var allBooks []Book
    currentURL := baseURL

    for page := 1; page <= maxPages; page++ {
        fmt.Printf("Scraping page %d/%d...\n", page, maxPages)

        doc, err := fetchPage(currentURL)
        if err != nil {
            fmt.Printf(" Error loading page: %v\n", err)
            stats.Errors++
            time.Sleep(2 * time.Second)
            continue
        }

        books := extractBooks(doc, baseURL)
        fmt.Printf(" Found %d books\n", len(books))
        allBooks = append(allBooks, books...)
        stats.PagesScraped++
        stats.BooksFound += len(books)

        nextURL, ok := getNextPageURL(doc, currentURL)
        if !ok {
            break
        }
        currentURL = nextURL

        // Rate limit
        time.Sleep(1 * time.Second)
    }

    stats.EndTime = time.Now()
    return allBooks, stats, nil
}

```

scrapePaginatedBooks Function

Purpose: Main scraping function that handles pagination.

How it works:

1. Initializes statistics tracking with current time
2. Starts with the base URL
3. For each page (up to maxPages):
 - o Fetches and parses the page
 - o Extracts all books from the page
 - o Updates statistics

- Finds the next page URL
 - Implements rate limiting (1 second delay between requests)
4. Stops if there's no next page or max pages reached
 5. Records end time and returns all books and statistics

Error handling:

- Logs errors but continues scraping other pages
- Implements a 2-second delay after errors to avoid overwhelming the server
- Increments error counter for reporting

Rate limiting: Sleeps for 1 second between pages to be respectful to the server.

```
func printStats(stats *ScraperStats) {
    duration := stats.EndTime.Sub(stats.StartTime).Seconds()
    avgBooks := 0.0
    if stats.PagesScraped > 0 {
        avgBooks = float64(stats.BooksFound) / float64(stats.PagesScraped)
    }
    fmt.Println("\n==== Scraping Statistics ===")
    fmt.Printf("Pages scraped: %d\n", stats.PagesScraped)
    fmt.Printf("Total books found: %d\n", stats.BooksFound)
    fmt.Printf("Errors: %d\n", stats.Errors)
    fmt.Printf("Duration: %.2f seconds\n", duration)
    fmt.Printf("Average books per page: %.1f\n", avgBooks)
}
```

func printStats(stats *ScraperStats)

Purpose: Displays scraping statistics in a formatted report.

Metrics displayed:

- Total pages successfully scraped
- Total books found
- Number of errors encountered
- Total execution time
- Average books per page

Calculation: Computes average books per page and total duration for performance analysis.

```
func main() {
    baseURL := "http://books.toscrape.com/catalogue/page-1.html"
    maxPages := 5

    fmt.Printf("Starting paginated scraper...\n")
    fmt.Printf("Max pages: %d\n\n", maxPages)

    allBooks, stats, err := scrapePaginatedBooks(baseURL, maxPages)
    if err != nil {
        fmt.Printf("Scraping failed: %v\n", err)
        return
    }

    printStats(stats)

    data, _ := json.MarshalIndent(allBooks, "", "    ")
    filename := "paginated_books.json"
    _ = os.WriteFile(filename, data, 0644)

    fmt.Printf("\nSaved %d books to %s\n", len(allBooks), filename)
}
```

func main()

Purpose: Entry point that orchestrates the entire scraping process.

Execution flow:

1. Defines configuration (base URL and max pages = 5)
2. Prints initial configuration
3. Calls `scrapePaginatedBooks` to perform the scraping
4. Prints statistics report
5. Converts results to pretty-printed JSON (with indentation)
6. Saves JSON data to "paginated_books.json" file
7. Prints confirmation message

Output:

```
PS C:\Users\VUTHANHHUNG\Desktop\Netcen Pro\VuThanhNhan - ITITIU21267 - Lab4\Task3.1> go run .\main.go
Starting paginated scraper...
Max pages: 5

Scraping page 1/5...
  Found 20 books
Scraping page 2/5...
  Found 20 books
Scraping page 3/5...
  Found 20 books
Scraping page 4/5...
  Found 20 books
Scraping page 5/5...
  Found 20 books

==== Scraping Statistics ====
Pages scraped: 5
Total books found: 100
Errors: 0
Duration: 8.90 seconds
Average books per page: 20.0

Saved 100 books to paginated_books.json
```

Task 4.1:

```
func NewTMDBClient(apiKey string) *TMDBClient {
    return &TMDBClient{
        APIKey:     apiKey,
        baseURL:   "https://api.themoviedb.org/3",
        httpClient: &http.Client{Timeout: 10 * time.Second},
        genres:     make(map[int]string),
    }
}
```

NewTMDBClient

Purpose: Constructor function that initializes the TMDB client.

- Sets 10-second timeout to prevent hanging requests
- Initializes empty genre map
- Returns pointer for efficient memory usage

```
func (c *TMDBCClient) loadGenres() error {
    endpoint := fmt.Sprintf("%s/genre/movie/list?api_key=%s", c.BaseURL, c.APIKey)
    resp, err := c.HTTPClient.Get(endpoint)
    if err != nil {
        return err
    }
    defer resp.Body.Close()

    var genreResp TMDBGenreResponse
    if err := json.NewDecoder(resp.Body).Decode(&genreResp); err != nil {
        return err
    }

    for _, genre := range genreResp.Genres {
        c.Genres[genre.ID] = genre.Name
    }
    return nil
}
```

loadGenres

Purpose: Fetches genre list from TMDB and builds genre ID to name mapping.

- Makes GET request to genre endpoint
- Decodes JSON response directly from response body
- Populates genre map for later use
- Must be called before searching movies

```

func (c *TMDBClient) searchMovies(query string) ([]TMDBMovie, error) {
    endpoint := fmt.Sprintf("%s/search/movie?api_key=%s&query=%s",
        c.BaseURL, c.APIKey, url.QueryEscape(query))

    resp, err := c.HTTPClient.Get(endpoint)
    if err != nil {
        return nil, err
    }
    defer resp.Body.Close()

    if resp.StatusCode != http.StatusOK {
        body, _ := io.ReadAll(resp.Body)
        return nil, fmt.Errorf("API error: %d - %s", resp.StatusCode, string(body))
    }

    var searchResp TMDBSearchResponse
    if err := json.NewDecoder(resp.Body).Decode(&searchResp); err != nil {
        return nil, err
    }

    // Map genre IDs to names
    for i := range searchResp.Results {
        for _, genreID := range searchResp.Results[i].GenreIDs {
            if genreName, exists := c.Genres[genreID]; exists {
                searchResp.Results[i].Genres = append(searchResp.Results[i].Genres, genreName)
            }
        }
    }

    return searchResp.Results, nil
}

```

searchMovies

Purpose: Searches TMDB for movies matching query string.

- URL-escapes query to handle special characters
- Checks HTTP status code for errors
- Converts genre IDs to human-readable names
- Returns array of TMDBMovie structs

```

func (db *MovieDatabase) Add(movie MovieInfo) error {
    movieID := movie.ID

    // Check if already exists
    if _, exists := db.Movies[movieID]; exists {
        return nil
    }

    // Add movie to Movies map
    db.Movies[movieID] = movie

    // Update genre index
    for _, genre := range movie.Genres {
        db.Genres[genre] = append(db.Genres[genre], movieID)
    }

    // Update director index
    if movie.Director != "" {
        db.Directors[movie.Director] = append(db.Directors[movie.Director], movieID)
    }

    // Update year index
    if movie.Year > 0 {
        db.Years[movie.Year] = append(db.Years[movie.Year], movieID)
    }

    // Update count
    db.TotalCount++

    return nil
}

```

Add Method

Purpose: Adds a movie to database and updates all indexes.

- Prevents duplicates by checking if movie ID exists
- Updates all secondary indexes (genres, directors, years)
- Each index maps to a slice of movie IDs
- Increments total count

```
func (db *MovieDatabase) Get(id string) (*MovieInfo, error) {
    movie, exists := db.Movies[id]
    if !exists {
        return nil, fmt.Errorf("movie not found: %s", id)
    }
    return &movie, nil
}
```

Get Method

Purpose: Retrieves a movie by ID (O(1) lookup).

- Direct map lookup - very fast
- Returns pointer to avoid copying large struct
- Returns error if movie doesn't exist

```
func (db *MovieDatabase) Search(query string) ([]MovieInfo, error) {
    var results []MovieInfo
    query = strings.ToLower(query)

    for _, movie := range db.Movies {
        if strings.Contains(strings.ToLower(movie.Title), query) {
            results = append(results, movie)
        }
    }

    return results, nil
}
```

Search Method

Purpose: Searches movies by title (case-insensitive).

- Converts query to lowercase for case-insensitive matching
- Iterates through all movies (O(n) complexity)
- Uses substring matching with strings.Contains
- Returns all matching movies

```
func (db *MovieDatabase) GetByGenre(genre string) ([]MovieInfo, error) {
    var results []MovieInfo

    movieIDs, exists := db.Genres[genre]
    if !exists {
        return results, nil
    }

    for _, id := range movieIDs {
        if movie, err := db.Get(id); err == nil {
            results = append(results, *movie)
        }
    }

    return results, nil
}
```

GetByGenre Method

Purpose: Gets all movies in a specific genre using index.

- Uses genre index for fast lookup ($O(1)$) to find IDs
- Iterates through movie IDs and retrieves full movie data
- Returns empty slice if genre doesn't exist
- Efficient: only looks up relevant movies

```

func (db *MovieDatabase) GetByYear(year int) ([]MovieInfo, error) {
    var results []MovieInfo

    movieIDs, exists := db.Years[year]
    if !exists {
        return results, nil
    }

    for _, id := range movieIDs {
        if movie, err := db.Get(id); err == nil {
            results = append(results, *movie)
        }
    }

    return results, nil
}

func (db *MovieDatabase) GetByDirector(director string) ([]MovieInfo, error) {
    var results []MovieInfo

    movieIDs, exists := db.Directors[director]
    if !exists {
        return results, nil
    }

    for _, id := range movieIDs {
        if movie, err := db.Get(id); err == nil {
            results = append(results, *movie)
        }
    }

    return results, nil
}

```

GetByYear & GetByDirector

Purpose: Query movies by year or director using respective indexes.

- Same pattern as GetByGenre
- Demonstrates index-based querying strategy

```

func (db *MovieDatabase) Update(movie MovieInfo) error {
    if _, exists := db.Movies[movie.ID]; !exists {
        return fmt.Errorf("movie not found: %s", movie.ID)
    }
    db.Movies[movie.ID] = movie
    return nil
}

func (db *MovieDatabase) Delete(id string) error {
    if _, exists := db.Movies[id]; !exists {
        return fmt.Errorf("movie not found: %s", id)
    }
    delete(db.Movies, id)
    db.TotalCount--
    return nil
}

func (db *MovieDatabase) Save(filename string) error {
    data, err := json.MarshalIndent(db, "", " ")
    if err != nil {
        return err
    }
    return os.WriteFile(filename, data, 0644)
}

func (db *MovieDatabase) Load(filename string) error {
    data, err := os.ReadFile(filename)
    if err != nil {
        return err
    }
    return json.Unmarshal(data, db)
}

```

Update Method

Purpose: Updates an existing movie's information.

- Checks if movie exists first
- Simple map assignment to update
- Note: This simplified version doesn't rebuild indexes

Delete Method

Purpose: Removes a movie from the database.

- Validates movie exists before deleting
- Uses built-in delete() function
- Decrements total count
- Note: This simplified version doesn't clean up indexes

Save & Load Methods

Purpose: Persists database to disk and loads it back.

- Save: Converts database to pretty-printed JSON
- Load: Reads JSON file and unmarshals into database
- File permissions 0644: owner read/write, others read-only
- Enables data persistence between program runs

```
func (db *MovieDatabase) PrintStatistics() {
    fmt.Println("\n==== Movie Database Statistics ===")
    fmt.Printf("Total Movies: %d\n", db.TotalCount)
    fmt.Printf("Total Genres: %d\n", len(db.Genres))
    fmt.Printf("Total Directors: %d\n", len(db.Directors))
    fmt.Printf("Year Range: %d entries\n\n", len(db.Years))

    // Genre statistics
    fmt.Println("Movies by Genre:")
    type genreCount struct {
        name string
        count int
    }
    var genreCounts []genreCount
    for genre, movieIDs := range db.Genres {
        if len(movieIDs) > 0 {
            genreCounts = append(genreCounts, genreCount{genre, len(movieIDs)})
        }
    }
    sort.Slice(genreCounts, func(i, j int) bool {
        return genreCounts[i].count > genreCounts[j].count
    })
    for _, gc := range genreCounts {
        fmt.Printf(" - %s: %d movies\n", gc.name, gc.count)
    }
}
```

PrintStatistics Method

Purpose: Displays comprehensive database statistics.

- Shows totals for movies, genres, directors, years
- Sorts genres by movie count (descending)
- Uses anonymous struct for sorting pairs
- Provides overview of database composition

```
func buildMovieDatabase(apiKey string) (*MovieDatabase, error) {
    db := NewMovieDatabase()
    client := NewTMDBClient(apiKey)

    // Load genres first
    fmt.Println("Loading movie genres from TMDB...")
    err := client.loadGenres()
    if err != nil {
        return nil, fmt.Errorf("failed to load genres: %w", err)
    }
    fmt.Printf("Loaded %d genres\n", len(client.Genres))

    // Define search queries
    searchQueries := []string{
        "marvel", "star wars", "harry potter", "batman",
        "comedy", "horror", "romance", "action",
        "2023", "2022", "2021", "classic",
    }

    fmt.Println("\nBuilding movie database...")

    for i, query := range searchQueries {
        fmt.Printf("[%d/%d] Searching for: %s\n", i+1, len(searchQueries), query)

        // Search movies using TMDB client
        movies, err := client.searchMovies(query)
        if err != nil {
            fmt.Printf(" Error: %v\n", err)
            continue
        }
    }
}
```

```

    // Add to database
    added := 0
    for _, movie := range movies {
        movieInfo := MovieInfo{
            ID:          fmt.Sprintf("%d", movie.ID),
            Title:       movie.Title,
            Year:        extractYear(movie.ReleaseDate),
            Description: movie.Overview,
            Genres:      movie.Genres,
            Rating:     movie.Rating,
            Source:      "TMDB",
            LastUpdated: time.Now().Format(time.RFC3339),
        }

        // Only add if not duplicate
        if _, exists := db.Movies[movieInfo.ID]; !exists {
            db.Add(movieInfo)
            added++
        }
    }

    fmt.Printf("  Added %d new movies (found %d total)\n", added, len(movies))

    // Rate limiting: 1 request per second
    time.Sleep(1 * time.Second)
}

db.LastUpdated = time.Now()
fmt.Printf("\nDatabase building complete!\n")

return db, nil
}

```

buildMovieDatabase Function

Purpose: Main pipeline that collects movies from TMDB.

- **Step 1:** Load genres (required for mapping genre IDs)
- **Step 2:** Define diverse search queries (12 categories)
- **Step 3:** Loop through queries with progress tracking
- **Step 4:** Transform TMDB data to MovieInfo format
- **Step 5:** Check for duplicates before adding
- **Step 6:** Rate limiting (1 second between requests)
- **Step 7:** Update timestamp when complete

Search Strategy:

- Franchises: "marvel", "star wars", "harry potter", "batman"
- Genres: "comedy", "horror", "romance", "action"
- Years: "2023", "2022", "2021", "classic"
- Target: 100+ unique movies

```
func extractYear(dateStr string) int {  
    if len(dateStr) >= 4 {  
        var year int  
        fmt.Sscanf(dateStr[:4], "%d", &year)  
        return year  
    }  
    return 0  
}
```

extractYear Helper Function

Purpose: Extracts year from TMDB date format ("YYYY-MM-DD").

- Takes first 4 characters (year part)
- Uses fmt.Sscanf to parse integer
- Returns 0 if date string is too short

```
func main() {
    fmt.Println("==== Movie Database Builder ====\n")

    apiKey := "33be097c32c7ec8df2864b26e113d643" // Replace with your key

    // Build database
    fmt.Println("Starting database collection...")
    db, err := buildMovieDatabase(apiKey)
    if err != nil {
        fmt.Printf("Error building database: %v\n", err)
        return
    }

    // Print statistics
    db.PrintStatistics()

    // Test search
    fmt.Println("\n==== Testing Search Functions ====")

    // Search by title
    fmt.Println("\nSearching for 'spider':")
    results, _ := db.Search("spider")
    for i, movie := range results {
        if i < 3 {
            fmt.Printf(" %d. %s (%d) - Rating: %.1f\n",
                      i+1, movie.Title, movie.Year, movie.Rating)
        }
    }
}
```

```

// Search by genre
if len(db.Genres) > 0 {
    var firstGenre string
    for genre := range db.Genres {
        firstGenre = genre
        break
    }

    fmt.Printf("\nMovies in genre '%s':\n", firstGenre)
    genreMovies, _ := db.GetByGenre(firstGenre)
    for i, movie := range genreMovies {
        if i < 3 {
            fmt.Printf(" %d. %s (%d)\n", i+1, movie.Title, movie.Year)
        }
    }
}

// Save to file
filename := "movie_database.json"
err = db.Save(filename)
if err != nil {
    fmt.Printf("Error saving database: %v\n", err)
    return
}

// Get file size
fileInfo, _ := os.Stat(filename)
sizeKB := fileInfo.Size() / 1024

fmt.Printf("\n✓ Database saved successfully!\n")
fmt.Printf(" File: %s\n", filename)
fmt.Printf(" Size: %d KB\n", sizeKB)
fmt.Printf(" Movies: %d\n", db.TotalCount)
}

```

main Function

Purpose: Orchestrates the entire program flow.

Execution Steps:

1. **Build Database:** Collect movies from TMDB API
2. **Display Statistics:** Show database composition
3. **Test Searches:** Demonstrate query functionality

- Title search: "spider"
 - Genre search: First available genre
4. **Save Database:** Persist to JSON file
 5. **Show Results:** Display file info and success message