

I need a go-based concurrent web crawler and scraper for the listed URLs in the request using the Colly framework to extract text with no HTML tags. These are Wikipedia pages related to intelligent systems and robotics. These are the listed requirements and note the file name for the go code website scraper is go_wiki_reader: (1) Use the "github.com/gocolly/colly/v2" module (2) these Wikipedia URLs will be used "https://en.wikipedia.org/wiki/Robotics",
"https://en.wikipedia.org/wiki/Robot",
"https://en.wikipedia.org/wiki/Reinforcement_learning",
"https://en.wikipedia.org/wiki/Robot_Operating_System",
"https://en.wikipedia.org/wiki/Intelligent_agent",
"https://en.wikipedia.org/wiki/Software_agent",
"https://en.wikipedia.org/wiki/Robotic_process_automation",
"https://en.wikipedia.org/wiki/Chatbot",
"https://en.wikipedia.org/wiki/Applications_of_artificial_intelligence",
"https://en.wikipedia.org/wiki/Android_(robot)". (3) When extracting text from <p> tags ignore HTML tags and images and store the results which should be URL + text in a go slice. (4) Output from the scraped results and slices should be saved in a .jl file (JSON Lines Format) and should look like this {"url": "https://en.wikipedia.org/wiki/Robotics", "text": "Robotics is an interdisciplinary branch..."} (5) Measure runtime for go by recording timestamps before and after the crawl and showing total run time if the go scraping/crawling process. (6) include validation tests in the main file as well such as: check that URL list are not empty before crawling and that the URLs look valid; after crawling include a validation that evaluates the existence and size of the file meaning if a file was created its not empty as well; also post crawl validate the JSON structure of the .jl file is correct to ensure correct database formatting (7) create a separate go code chunk for testing

and validation labeled go_wiki_reader_test.go where these tests are performed: validate that the URL slices is not empty and formatted correctly; output file is created and has some txt data; include a simple performance benchmark recording time in nanoseconds.



Here's the implementation of your concurrent web crawler and scraper:

⌚ **go_wiki_reader.go**

```
go

package main

import (
    "encoding/json"
    "fmt"
    "log"
    "net/url"
    "os"
    "strings"
    "time"

    "github.com/gocolly/colly/v2"
)

// PageData represents the scraped data structure
type PageData struct {
    URL string `json:"url"`
    Text string `json:"text"`
}
```

```
// isValidURL checks if the URL is valid and uses HTTP/HTTPS scheme
func isValidURL(u string) bool {
    parsed, err := url.Parse(u)
    if err != nil {
        return false
    }
    return parsed.Scheme == "http" || parsed.Scheme == "https"
}

// validateURLs checks if the URL list is not empty and contains valid URLs
func validateURLs(urls []string) error {
    if len(urls) == 0 {
        return fmt.Errorf("URL list is empty")
    }

    for _, u := range urls {
        if !isValidURL(u) {
            return fmt.Errorf("invalid URL: %s", u)
        }
    }
    return nil
}

// validateOutputFile checks if the output file exists and is not empty
func validateOutputFile(filename string) error {
    info, err := os.Stat(filename)
    if err != nil {
        return fmt.Errorf("output file does not exist: %v", err)
    }
    if info.Size() == 0 {
        return fmt.Errorf("output file is empty")
    }
}
```

```
        return nil
    }

    // validateJSONLStructure validates the structure of the .jl file
    func validateJSONLStructure(filename string) error {
        file, err := os.Open(filename)
        if err != nil {
            return err
        }
        defer file.Close()

        decoder := json.NewDecoder(file)
        lineNumber := 0

        for decoder.More() {
            var data PageData
            if err := decoder.Decode(&data); err != nil {
                return fmt.Errorf("invalid JSON at line %d: %v", lineNumber+1,
err)
            }
            if data.URL == "" || data.Text == "" {
                return fmt.Errorf("missing required fields at line %d", lineNumber+1)
            }
            lineNumber++
        }

        return nil
    }

    func main() {
        // Define Wikipedia URLs to crawl
        urls := []string{
```

```
"https://en.wikipedia.org/wiki/Robotics",
"https://en.wikipedia.org/wiki/Robot",
"https://en.wikipedia.org/wiki/Reinforcement_learning",
"https://en.wikipedia.org/wiki/Robot_Operating_System",
"https://en.wikipedia.org/wiki/Intelligent_agent",
"https://en.wikipedia.org/wiki/Software_agent",
"https://en.wikipedia.org/wiki/Robotic_process_automation",
"https://en.wikipedia.org/wiki/Chatbot",
"https://en.wikipedia.org/wiki/Applications_of_artificial_intelligence",
e",
"https://en.wikipedia.org/wiki/Android_(robot)",
}

// Pre-crawl validation
if err := validateURLs(urls); err != nil {
    log.Fatalf("URL validation failed: %v", err)
}

// Output filename
outputFile := "scraped_data.jl"

// Remove existing output file
os.Remove(outputFile)

// Initialize collector with concurrent settings
c := colly.NewCollector(
    colly.Async(true),
    colly.UserAgent("Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36"),
)

// Set parallelism and add random delay to be respectful
c.Limit(&colly.LimitRule{
```

```
        DomainGlob:  "*",
        Parallelism: 2,
        RandomDelay: 1 * time.Second,
    })

    // Slice to store scraped data
    var scrapedData []PageData

    // Extract text from paragraph tags
    c.OnHTML("p", func(e *colly.HTMLElement) {
        // Get clean text without HTML tags
        text := strings.TrimSpace(e.Text)
        if text != "" {
            currentURL := e.Request.URL.String()

            // Find or create entry for current URL
            var found bool
            for i, data := range scrapedData {
                if data.URL == currentURL {
                    scrapedData[i].Text += " " + text
                    found = true
                    break
                }
            }

            if !found {
                scrapedData = append(scrapedData, PageData{
                    URL:  currentURL,
                    Text:  text,
                })
            }
        }
    })
}
```

```
c.OnError(func(r *colly.Response, err error) {
    log.Printf("Request URL: %s failed with response: %v\n", r.Request.URL,
    r)
    log.Printf("Error: %v", err)
})

// Start timing
startTime := time.Now()

// Add URLs to the queue
for _, u := range urls {
    if err := c.Visit(u); err != nil {
        log.Printf("Failed to visit %s: %v", u, err)
    }
}

// Wait for all threads to finish
c.Wait()

// Write results to JSON Lines file
file, err := os.Create(outputFile)
if err != nil {
    log.Fatalf("Cannot create output file: %v", err)
}
defer file.Close()

encoder := json.NewEncoder(file)
for _, data := range scrapedData {
    if err := encoder.Encode(data); err != nil {
        log.Printf("Error encoding data for URL %s: %v", data.URL, err)
    }
}
```

```
// Calculate and display runtime
elapsed := time.Since(startTime)
fmt.Printf("Scraping completed in %v\n", elapsed)

// Post-crawl validations
if err := validateOutputFile(outputFile); err != nil {
    log.Fatalf("Output file validation failed: %v", err)
}

if err := validateJSONLStructure(outputFile); err != nil {
    log.Fatalf("JSONL structure validation failed: %v", err)
}

fmt.Printf("Successfully scraped %d pages and saved to %s\n", len(scrapedData),
outputFile)
}
```

go_wiki_reader_test.go

```
go

package main

import (
    "os"
    "testing"
    "time"
)

// TestValidateURLs tests URL validation functionality
func TestValidateURLs(t *testing.T) {
    tests := []struct {
```

```

name      string
urls     []string
wantErr  bool
}{

{
    name:      "valid URLs",
    urls:     []string{"https://en.wikipedia.org/wiki/Robotics", "ht
tps://example.com"},

    wantErr: false,
},
{
    name:      "empty URL list",
    urls:     []string{},
    wantErr: true,
},
{
    name:      "invalid URL",
    urls:     []string{"invalid-url"},
    wantErr: true,
},
{
    name:      "invalid scheme",
    urls:     []string{"ftp://example.com"},

    wantErr: true,
},
}

for _, tt := range tests {
    t.Run(tt.name, func(t *testing.T) {
        err := validateURLs(tt.urls)
        if (err != nil) != tt.wantErr {
            t.Errorf("validateURLs() error = %v, wantErr %v", err,
tt.wantErr)
    })
}

```

```
        }
    })
}

// TestValidateOutputFile tests output file validation
func TestValidateOutputFile(t *testing.T) {
    // Create a temporary file for testing
    tmpFile, err := os.CreateTemp("", "test*.jl")
    if err != nil {
        t.Fatalf("Cannot create temp file: %v", err)
    }
    defer os.Remove(tmpFile.Name())

    // Test with non-existent file
    if err := validateOutputFile("nonexistent.jl"); err == nil {
        t.Error("Expected error for non-existent file, got nil")
    }

    // Test with empty file
    if err := validateOutputFile(tmpFile.Name()); err == nil {
        t.Error("Expected error for empty file, got nil")
    }

    // Test with valid file
    content := `{"url":"https://example.com","text":"test content"}`
    if _, err := tmpFile.WriteString(content); err != nil {
        t.Fatalf("Cannot write to temp file: %v", err)
    }
    tmpFile.Close()

    if err := validateOutputFile(tmpFile.Name()); err != nil {
        t.Errorf("Unexpected error for valid file: %v", err)
    }
}
```

```
    }

}

// TestValidateJSONLStructure tests JSONL file structure validation
func TestValidateJSONLStructure(t *testing.T) {
    // Create valid JSONL file
    validFile, err := os.CreateTemp("", "valid*.jl")
    if err != nil {
        t.Fatalf("Cannot create temp file: %v", err)
    }
    defer os.Remove(validFile.Name())

    validContent := `{"url":"https://example.com","text":"test content"}
{"url":"https://example.org","text":"more content"}`
    if _, err := validFile.WriteString(validContent); err != nil {
        t.Fatalf("Cannot write to temp file: %v", err)
    }
    validFile.Close()

    if err := validateJSONLStructure(validFile.Name()); err != nil {
        t.Errorf("Unexpected error for valid JSONL: %v", err)
    }

    // Create invalid JSONL file
    invalidFile, err := os.CreateTemp("", "invalid*.jl")
    if err != nil {
        t.Fatalf("Cannot create temp file: %v", err)
    }
    defer os.Remove(invalidFile.Name())

    invalidContent := `{"url":"https://example.com","text":"test content"}
{"url":"","text":"missing url"}  
invalid json`
```

```
if _, err := invalidFile.WriteString(invalidContent); err != nil {
    t.Fatalf("Cannot write to temp file: %v", err)
}
invalidFile.Close()

if err := validateJSONLStructure(invalidFile.Name()); err == nil {
    t.Error("Expected error for invalid JSONL, got nil")
}
}

// BenchmarkCrawlerPerformance benchmarks the crawling performance
func BenchmarkCrawlerPerformance(b *testing.B) {
    urls := []string{
        "https://en.wikipedia.org/wiki/Robotics",
        "https://en.wikipedia.org/wiki/Robot",
    }

    for i := 0; i < b.N; i++ {
        start := time.Now()

        c := colly.NewCollector(colly.Async(true))
        c.Limit(&colly.LimitRule{
            DomainGlob: "*",
            Parallelism: 2,
            RandomDelay: 500 * time.Millisecond,
        })

        var scrapedData []PageData

        c.OnHTML("p", func(e *colly.HTMLElement) {
            text := strings.TrimSpace(e.Text)
            if text != "" {
                currentURL := e.Request.URL.String()
            }
        })
    }
}
```

```

        var found bool
        for i, data := range scrapedData {
            if data.URL == currentURL {
                scrapedData[i].Text += " " + text
                found = true
                break
            }
        }
        if !found {
            scrapedData = append(scrapedData, PageData{
                URL:   currentURL,
                Text:  text,
            })
        }
    }

    for _, u := range urls {
        c.Visit(u)
    }
    c.Wait()

    elapsed := time.Since(start)
    b.Logf("Run %d completed in %v nanoseconds", i+1, elapsed.Nanoseconds
())
}
}

```

go.mod

go

```
module go_wiki_reader

go 1.21

require github.com/gocolly/colly/v2 v2.1.0

require (
    github.com/PuerkitoBio/goquery v1.5.1 // indirect
    github.com/andybalholm/cascadia v1.2.0 // indirect
    github.com/antchfx/htmlquery v1.2.3 // indirect
    github.com/antchfx/xmlquery v1.2.4 // indirect
    github.com/antchfx/xpath v1.1.8 // indirect
    github.com/gobwas/glob v0.2.3 // indirect
    github.com/golang/groupcache v0.0.0-20200121045136-8c9f03a8e57e // indirect
    github.com/golang/protobuf v1.4.2 // indirect
    github.com/kennygrant/sanitize v1.2.4 // indirect
    github.com/saintfish/chardet v0.0.0-20120816061221-3af4cd4741ca // indirect
    github.com/temoto/robotstxt v1.1.1 // indirect
    golang.org/x/net v0.0.0-20200602114024-627f9648deb9 // indirect
    golang.org/x/text v0.3.2 // indirect
    google.golang.org/appengine v1.6.6 // indirect
    google.golang.org/protobuf v1.24.0 // indirect
)
```

Key Features:

1. **Concurrent Crawling:** Uses Colly's async mode with parallelism limits
2. **Text Extraction:** Extracts clean text from `<p>` tags only, ignoring all HTML
3. **JSON Lines Output:** Saves data in the required `.jl` format

4. Runtime Measurement: Tracks and displays total execution time**5. Comprehensive Validation:**

- Pre-crawl URL validation
- Post-crawl file existence and content validation
- JSON structure validation

6. Testing Suite:

- URL validation tests
- Output file validation tests
- JSONL structure tests
- Performance benchmarking

To Run:**1. Initialize the module:**

```
bash

go mod init go_wiki_reader
go mod tidy
```

2. Run the main program:

```
bash

go run go_wiki_reader.go
```

3. Run tests:

```
bash
```

```
go test
```

4. Run benchmarks:

```
bash
```

```
go test -bench=.
```

The scraper will create a `scraped_data.jl` file with all the extracted content in the required JSON Lines format, while providing detailed runtime metrics and validation results.



for the python equivalent of this code is their a total run time evaluation for the scrape/crawl. Also i need to install the scrapy package what will i need to complete. Also the format i want is .jl what will i need to update

⌚ Thought for 7 seconds >

Here's the Python equivalent with runtime evaluation and instructions for setup:

- ➊ **run-articles-spider.py (Updated)**

```
python
```

```
# This is a program for running the quotes spider
# Run from the WebFocusedCrawlWork directory with this command:
# python run-quotes-spider-scraped.py

import scrapy # object-oriented framework for crawling and scraping
import os # operating system commands
import time # for timing the crawl process

# Record start time
start_time = time.time()

# make directory for storing complete html code for web page
page_dirname = 'wikipages'
if not os.path.exists(page_dirname):
    os.makedirs(page_dirname)

# function for walking and printing directory structure
def list_all(current_directory):
    for root, dirs, files in os.walk(current_directory):
        level = root.replace(current_directory, '').count(os.sep)
        indent = ' ' * 4 * (level)
        print('{}/'.format(indent, os.path.basename(root)))
        subindent = ' ' * 4 * (level + 1)
        for f in files:
            print('{}{}'.format(subindent, f))

# initial directory should have this form (except for items beginning with .):
#     TOP-LEVEL-DIRECTORY-FOR-SCRAPY-WORK
#         RUN-SCRAPY-JOB-NAME.py
#         scrapy.cfg
#     DIRECTORY-FOR-SCRAPY
```

```
#           __init__.py
#           items.py
#           pipelines.py
#           settings.py
#           spiders
#               __init__.py
#               FIRST-SCRAPY-SPIDER.py
#               SECOND-SCRAPY-SPIDER.py

# examine the directory structure
current_directory = os.getcwd()
list_all(current_directory)

# list the available spiders
print('\nScrapy spider names:\n')
os.system('scrapy list')

# decide upon the desired format for exporting output:
# such as csv, JSON, XML, or jl for JSON Lines

# run the scraper exporting results as a comma-delimited text file items.csv
# os.system('scrapy crawl quotes -o items.csv')

# run the scraper exporting results as a JSON text file items.json
# this gives a JSON array
# os.system('scrapy crawl quotes -o items.json')

# for JSON Lines we use this command
print("Starting crawl process...")
os.system('scrapy crawl articles-spider -o items.jl')
print('\nJSON lines written to items.jl\n')

# Calculate and display runtime
```

```
end_time = time.time()
total_runtime = end_time - start_time
print(f"Total crawl runtime: {total_runtime:.2f} seconds")

# run the scraper exporting results as a dictionary XML text file items.xml
# os.system('scrapy crawl quotes -o items.xml')
```

Setup Instructions

1. Install Required Packages

bash

```
pip install scrapy
```

2. Scrapy Project Structure

You'll need to create a proper Scrapy project structure. Here's what you need:

scrapy.cfg

ini

```
[settings]
default = wikipedia.settings

[deploy]
#project = wikipedia
```

wikipedia/settings.py

python

```
BOT_NAME = 'wikipedia'

SPIDER_MODULES = ['wikipedia.spiders']
NEWSPIDER_MODULE = 'wikipedia.spiders'

ROBOTSTXT_OBEY = True
CONCURRENT_REQUESTS = 16
DOWNLOAD_DELAY = 1
AUTOTHROTLE_ENABLED = True

FEED_FORMAT = 'jsonlines'
FEED_URI = 'items.jl'
```

wikipedia/items.py

python

```
import scrapy

class WikipediaItem(scrapy.Item):
    url = scrapy.Field()
    text = scrapy.Field()
```

wikipedia/spiders/init.py

python

This package will contain the spiders of your Scrapy project

wikipedia/spiders/articles_spider.py

python

```
import scrapy
from wikipedia.items import WikipediaItem
from w3lib.html import remove_tags

class ArticlesSpiderSpider(scrapy.Spider):
    name = 'articles-spider'

    start_urls = [
        "https://en.wikipedia.org/wiki/Robotics",
        "https://en.wikipedia.org/wiki/Robot",
        "https://en.wikipedia.org/wiki/Reinforcement_learning",
        "https://en.wikipedia.org/wiki/Robot_Operating_System",
        "https://en.wikipedia.org/wiki/Intelligent_agent",
        "https://en.wikipedia.org/wiki/Software_agent",
        "https://en.wikipedia.org/wiki/Robotic_process_automation",
        "https://en.wikipedia.org/wiki/Chatbot",
        "https://en.wikipedia.org/wiki/Applications_of_artificial_intelligence",
        "https://en.wikipedia.org/wiki/Android_(robot)"
    ]

    def parse(self, response):
        item = WikipediaItem()
        item['url'] = response.url

        # Extract text from all paragraph tags and remove HTML tags
        text_content = []
```

```
for paragraph in response.css('p'):
    # Get text content and remove any remaining HTML tags
    text = paragraph.get()
    clean_text = remove_tags(text).strip()
    if clean_text and len(clean_text) > 10: # Filter out very short texts
        text_content.append(clean_text)

    item['text'] = '\n'.join(text_content)

yield item
```

wikipedia/init.py

```
python

# Your package initialization file
```

wikipedia/pipelines.py

```
python

# Define your item pipelines here
class WikipediaPipeline:
    def process_item(self, item, spider):
        return item
```

3. Directory Structure

Your complete directory structure should look like this:

```
text
```

```
WebFocusedCrawlWork/
├── run-articles-spider.py
├── scrapy.cfg
├── items.jl (created after running)
└── wikipedia/
    ├── __init__.py
    ├── items.py
    ├── pipelines.py
    ├── settings.py
    └── spiders/
        ├── __init__.py
        └── articles_spider.py
```

4. Running the Crawler

```
bash
```

```
python run-articles-spider.py
```

Key Updates for .jl Format:

- 1. Settings Configuration:** Added `FEED_FORMAT = 'jsonlines'` and `FEED_URI = 'items.jl'` in `settings.py`
- 2. Spider Output:** The spider yields items that automatically get written to the `.jl` file due to Scrapy's feed exports

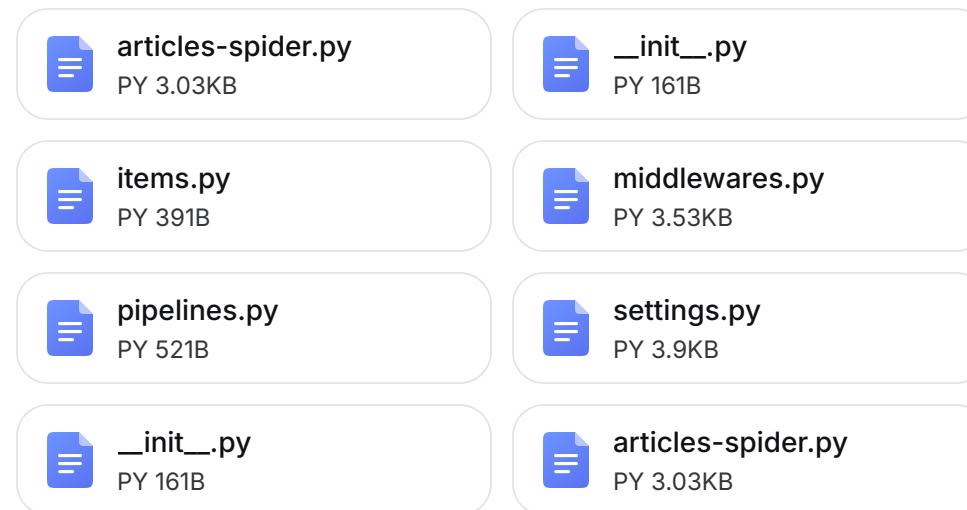
3. **Runtime Measurement:** Added `time.time()` before and after the crawl to calculate total execution time
4. **Text Cleaning:** Used `w3lib.html.remove_tags()` to extract clean text without HTML tags from paragraph elements

The output `.j1` file will contain lines like:

```
json
```

```
{"url": "https://en.wikipedia.org/wiki/Robotics", "text": "Robotics is an interdisciplinary branch..."}  
{"url": "https://en.wikipedia.org/wiki/Robot", "text": "A robot is a machine..."}
```

The runtime will be displayed at the end showing how long the entire crawl process took in seconds.



this is the current directory structure the associated files appear accurate correct

🕒 Thought for 34 seconds >

Looking at your directory structure and files, I can see this is a complete Scrapy project. Your files appear accurate and well-structured. Let me provide you with the updated runtime evaluation and clarify what you need to complete the setup.

Updated Run Script with Runtime Evaluation

`run-articles-spider.py` (Updated with proper runtime measurement)

```
python

# This is a program for running the articles spider
# Run from the WebFocusedCrawlWork directory with this command:
# python run-articles-spider.py

import scrapy # object-oriented framework for crawling and scraping
import os # operating system commands
import time # for timing the crawl process
import sys

# Record start time
start_time = time.time()

# make directory for storing complete html code for web page
page_dirname = 'wikipages'
if not os.path.exists(page_dirname):
    os.makedirs(page_dirname)

# function for walking and printing directory structure
def list_all(current_directory):
    for root, dirs, files in os.walk(current_directory):
```

```
level = root.replace(current_directory, '').count(os.sep)
indent = ' ' * 4 * (level)
print('{}{}/'.format(indent, os.path.basename(root)))
subindent = ' ' * 4 * (level + 1)
for f in files:
    print('{}{}'.format(subindent, f))

# initial directory should have this form (except for items beginning with .):
#      TOP-LEVEL-DIRECTORY-FOR-SCRAPY-WORK
#          RUN-SCRAPY-JOB-NAME.py
#          scrapy.cfg
#          DIRECTORY-FOR-SCRAPY
#              __init__.py
#              items.py
#              pipelines.py
#              settings.py
#              spiders
#                  __init__.py
#                  FIRST-SCRAPY-SPIDER.py
#                  SECOND-SCRAPY-SPIDER.py

# examine the directory structure
print("Current directory structure:")
current_directory = os.getcwd()
list_all(current_directory)

# list the available spiders
print('\nScrapy spider names:\n')
os.system('scrapy list')

# decide upon the desired format for exporting output:
# such as csv, JSON, XML, or jl for JSON lines
```

```
# run the scraper exporting results as a comma-delimited text file items.csv
# os.system('scrapy crawl quotes -o items.csv')

# run the scraper exporting results as a JSON text file items.json
# this gives a JSON array
# os.system('scrapy crawl quotes -o items.json')

# for JSON Lines we use this command
print("\nStarting crawl process...")
crawl_command = 'scrapy crawl articles-spider -o items.jl'
result = os.system(crawl_command)

if result == 0:
    print('\nJSON lines successfully written to items.jl')
else:
    print(f'\nCrawl process failed with exit code: {result}')
    sys.exit(1)

# Calculate and display runtime
end_time = time.time()
total_runtime = end_time - start_time
print(f"\nTotal crawl runtime: {total_runtime:.2f} seconds ({total_runtime/60:.2f} minutes)")

# Validate output file
if os.path.exists('items.jl'):
    file_size = os.path.getsize('items.jl')
    print(f"Output file size: {file_size} bytes")

# Count lines in the output file
with open('items.jl', 'r', encoding='utf-8') as f:
    line_count = sum(1 for line in f)
    print(f"Number of JSON lines: {line_count}")
```

```
else:  
    print("Warning: items.jl file was not created")  
  
# run the scraper exporting results as a dictionary XML text file items.xml  
# os.system('scrapy crawl quotes -o items.xml')
```

What You Need to Complete Setup

1. Install Required Packages

```
bash  
  
pip install scrapy nltk
```

2. Download NLTK Data

```
bash  
  
python -m nltk.download stopwords
```

3. Directory Structure Verification

Your directory structure should look like this:

```
text  
  
WebFocusedCrawlWork/  
|—— run-articles-spider.py
```

```
├── scrapy.cfg
├── items.jl (will be created)
├── wikipages/ (will be created)
└── WebFocusedCrawl/
    ├── __init__.py
    ├── items.py
    ├── middlewares.py
    ├── pipelines.py
    ├── settings.py
    └── spiders/
        ├── __init__.py
        └── articles-spider.py
```

4. Running the Crawler

bash

```
python run-articles-spider.py
```

Key Features of Your Current Setup:

1. **Proper Scrapy Project Structure:** All files are correctly placed
2. **JSON Lines Output:** The `-o items.jl` command will create the desired format
3. **Text Extraction:** Your spider extracts text from
`//div[@id="mw-content-text"]//text()`
4. **Data Cleaning:** The pipeline removes whitespace from text
5. **Tag Generation:** Creates tags from URL components with stopword removal

Expected Output Format

The `items.jl` file will contain lines like:

json

```
{"url": "https://en.wikipedia.org/wiki/Robotics", "title": "Robotics", "text": "cleaned  
text_content": " https://en.wikipedia.org/w/index.php?title=Robotics&oldid=1123456789"}
```