

Shangqing Hu – SEC01 (NUID 001374342)

Big Data System Engineering with Scala

Spring 2023

Assignment No.6(WebCrawler)



-List of Tasks Implemented

Implement the primitive web crawler that is partly complete in the crawler package on our class repo (you must use the Fall2022 branch).

There are three *TO BE IMPLEMENTED* to complete, with a total point value of 32. You may also earn up to 10 bonus points for suggestions on how to improve the web crawler (detailed code not required but you do need to explain in words what you would do). Two of these are in *WebCrawler.scala*. The other is in *MonadOps.scala* as follows: Please ensure that you pull the latest versions

of *WebCrawler.scala*, *HTMLParser.scala* and *MonadOps.scala*. You can get these from the class repo (see Course Material/Resources/Class Repository), the module name for this assignment is **assignment-web-crawler**.

For the processing of a Node, we will need to refer back to the way Scala processes XML documents which we covered in Serialization (that's the purpose of the tagsoup library). Hint: you can get all of the anchor, viz. the "a" nodes using

```
ns \ "a"
```

You can get the "href" property from these nodes using "\" and "@href".

There is also the main program but if you run that, you will need to provide Program arguments consisting of URL(s) at which to start crawling.

I strongly advise you to take advantage of the various hints. Also, make sure you know the types of any intermediate results that you derive: either by adding type annotation (in IDEA, just do option/alt + return/enter and it will give you the option of adding a type annotation) or by just selecting an identifier and displaying its type (in IDEA, that would be ctrl/shift/P). Any time you know the type you're starting with--and you know the type of result you need--now you just have to find existing methods to convert from one to the other.














-Code

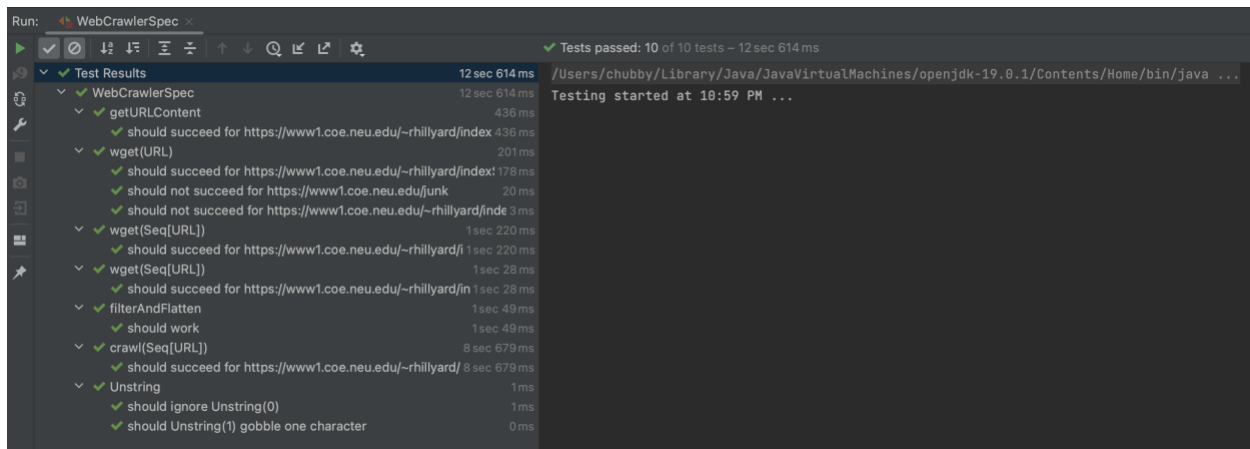
```
def wget(url: URL)(implicit ec: ExecutionContext): Future[Seq[URL]] = {
  // Hint: write as two nested for-comprehensions: the outer one (first) based on Seq, the inner (second) based on Try.
  // In the latter, use the method createURL(Option[URL], String) to get the appropriate URL for a relative link.
  // Don't forget to run it through validateURL.
  // 16 points.
  def getURLs(ns: Node): Seq[Try[URL]] = {
    for {
      n <- ns \\ "a"
      nh <- n \ "@href"
    } yield {
      for {
        u <- createURL(Some(url), nh.toString)
        v <- validateURL(u)
      } yield v
    }
  }
  // TO BE IMPLEMENTED
}
```

```
// Hint: write as a for-comprehension, using getURLContent (above) and getLinks above. You will also need MonadOps.asFuture
// 9 points.
for {
  s <- getURLContent(url)
  us <- MonadOps.asFuture(getLinks(s))
} yield us // TO BE IMPLEMENTED
```

```
def asOption[X](xe: Either[Throwable, X]): Option[X] = xe.toOption // TO BE IMPLEMENTED
```

-Unit tests

Run: MonadOpsSpec x		
            		
Test Results		754 ms
MonadOpsSpec		754 ms
LiftFuture		48 ms
should work		48 ms
AsFuture		2 ms
SequenceForgivingWithLogging		8 ms
SequenceWithLogging		3 ms
sequenceLax		2 ms
SequenceForgiving		3 ms
LiftTry		2 ms
zip(Option,Option)		3 ms
zip(Try,Try)		3 ms
zip(Future,Future)		15 ms
OptionToTry		5 ms
asEither		1 ms
Sequence		11 ms
sequence of Iterable		16 ms
sequenceImpatient		616 ms
should work for 1		107 ms
should work for 1, goodURL, 1/0 (less patient)		384 ms
should work for 1, goodURL, 1/0 (more patient)		125 ms
LiftOption		2 ms
Map2		2 ms
Flatten		12 ms
should work1		2 ms
should work2		5 ms
should work3		5 ms



-Findings and analysis

Suggestions about improve the WebCrawler (Attribute form CHATGPT)

1. Implement multithreading or asynchronous processing: This will allow your web crawler to fetch multiple pages simultaneously, making it more efficient and faster.
2. Use caching: Caching can help reduce the number of requests made to a website and improve the performance of the crawler. You can use tools like Redis or Memcached to cache the results.
3. Implement rate limiting: It is important to limit the number of requests made to a website to avoid overwhelming the server or getting blocked. You can implement rate limiting by setting a delay between requests or using a library like requests-threads that provides built-in rate limiting.
4. Use a better user agent: Some websites block certain user agents or throttle requests from suspicious user agents. You can change the user agent string to a more common or legitimate one to avoid getting blocked.
5. Handle errors and exceptions: Make sure to handle errors and exceptions that may occur while crawling the web. This can help prevent your crawler from crashing or getting stuck in an infinite loop.
6. Store data efficiently: Decide on a storage format that best suits your needs and optimize it for quick retrieval. You can use databases like MongoDB or SQLite to store data.
7. Improve the selection of URLs to crawl: Consider using more sophisticated algorithms to select URLs to crawl, such as those that prioritize popular pages or those that are likely to contain important information.
8. Use a robots.txt parser: The robots.txt file can contain information about which pages can and cannot be crawled. You can use a robots.txt parser to ensure your crawler is adhering to the rules set by the website.