# Programming assignment 1

Aljaž Koželj

Fakulteta za računalništvo in informatiko
Univerza v Ljubljani
ak6689@student.uni-lj.si

## 1 Introduction

In the first programming assignment, we implemented our own web crawler. The crawler received some seed sites and started crawling from there. The goal for the crawler was to retrieve one hundred thousand pages.

## 2 Technologies

### 2.1 Java

Java was selected as the programming language for the project. It was chosen mainly for it's familiarity, strong library (and framework) support and performance. It also has a strong concurrency support, which was needed for this project.

### 2.2 Spring Boot

Spring Boot is the main reason that Java was chosen. It's a pre-configured Spring framework which allows us to easily build stand-alone, production-grade applications. This allows us quick setup of features like logging, object-oriential mapping and dependecy injection. These features make development faster and easier. Business logic and data manipulation beans are created behind the scenes and the developer doesn't have to bother with that. The Spring Data module, for example, provides basic CRUD operations, over an entity, by simply injecting an interface.

### 2.3 PostgreSQL

The project demanded the use of the PostgreSQL database for storing the data. The schema for it was provided, but we were allowed to add additional stuff. The database was ran inside a docker container. The command is provided in the README.md file.

### 2.4 Maven

Maven is a build tool for Java projects. It provides a straightforward way of adding new libraries and building the project with them. By building the Spring boot project it provides us with a runnable .jar.

### 2.5 Other libraries used

**Table 1.** List of other libraries

| | |
|---|---|
| Selenium browser | Selenium is a headless web browser. It allowed us to load and render web pages and retrieve the rendered content. It also supports JavaScript execution, unlike many other headless web browsers. |
| JSoup | A Java library which provides an easy of parsing HTML content. It supports jQuery-like searching of DOM tree nodes. |
| Crawler Commons | Google's very own crawling utility library. We used it for parsing robots.txt rules and sitemap content. |
| StreamEx | A library that enchances Java 8 streams. It provides additional methods and is more readable than vanilla streams. |

## 3 Components

### 3.1 RobotsRuleHandler

This component of the program handles the robots.txt of sites. Given a requested URL, it returns an object describing the robot rules. First the component extracts the domain part of the url and checks a map, if the rules object was previously created. If not, it checks if the domain has already been crawled. Should a proper site entry be located in the database, it uses that to construct the robot rules object. Else, it uses a selenium web driver and retrieves the robots.txt file. While first retrieving a file for the site, we also check if a sitemap is defined. Should a sitemap be defined, it gets loaded and all the entries are created as page table entries (with page type set to frontier).

A newly loaded site then gets saved into the database with both the robots content and sitemap content. Here, we added a new constraint into the database: the domain column of the site table has to be unique. This decision seems reasonable, sites are defined by their domain column. This also allows us to not worry about concurrency errors while saving a site entry, since all we have to do is handle an exception.

### 3.2 SiteFrontierHandler

This component provides the workers with the next URL they must visit. It contains a concurrent queue and is accessed with synchronized methods. This prevents concurrency errors. This handler provides, upon request, the next visitable URL from the queue. Should the queue become empty, it loads the next 200 frontier pages. These are ordered by their accessedTime column, the oldest entries are selected first.

After the page table contains more than one hundred thousand entries, this component also provides information, if new frontier pages should be added.

### 3.3 WebCrawlerService

This component runs right after the Spring Boot framework is setup. If the page table is empty, it first creates frontier pages for all the seed sites. Then it initializes the robot rules, for these sites. This is done by calling the RobotsRuleHandler.

It then uses a task executor instance, to create worker instances. Each instance is started in a new thread. The pool of these workers is capped by a number defined by a main method argument. An additional 200 workers are added to the queue in the task executor. Should a pool instance fail, it will be replaced with a new one from the queue. An instance can fail due to an unhandled exception or losing it's connection to the database.

### 3.4 WebCrawlerWorker

The final component does all the heavy work during web crawling. There are always multiple instances of this component running, as defined in the WebCrawlerService. Each instance starts by initializing a Selenium webdriver. Here headlessness, timeouts, window size and other parameters are set.

Next the main loop of the worker starts. We describe this in further detail in the next section.

## 4 WebCrawlerWorker crawling

### 4.1 Loop

The workers work as long as the main loop runs. At the start of the loop, a new url is requested from the SiteFrontierHandler component. Should none be received, the worker stops it's work. All the new urls are frontier page entries in the database. The request for the new url is made thread safe, by using synchronize keyword and using a concurrent data structure.

### 4.2 Construct URL object

First, we try to construct an Java URL object from the frontier url. This allows us for easy extraction the domain part of the url. It also check, if the URL has a proper format. If this fails, we update the existing page object, by assigning it a http code of 403 and setting it's content to "MALFORMED". We also set the page type to html, since we don't want it in the frontier queue.

### 4.3 Retrieve robot rules

Here we a robots rule object from the RobotRulesHandler component. In case a new site was discovered, the handler will create a new site instance, retrieve robots.txt content and sitemap content. As was described above.

Then we check, if the provided frontier url is allowed to be access. The user agent we use is "*", when checking for robot rules.

### 4.4 Document check

Next we determine if the provided frontier url points to a binary document file. We check for the following extensions: .pdf, .doc, .docx, .ppt, .pptx. If one of these is present, then the page is a binary document. We retrieve the page object for this url, set it's type to binary and save. We also create a page data object, that holds the binary data for the binary. Unfortunately, due to time constraints and selenium limitations, we weren't able to implemented proper binary data retrieval. We create an empty page data object, where we set it's type and what page it points to, but the data remains empty. The object is still saved for statistical purposes.

### 4.5 HTML parsing

If the url doesn't point to a document, then we can retrieve the proper HTML markup. We do this, by providing the url to the Selenium webdriver. During this, we make sure to prepend the url with "http://" if necessary. Without the protocol, the webdriver will fail loading the webpage.

Should the retrieval of the markup fail at any point, we set the HTML content to "ERROR" and assign a 404 code to the page object. In case of a succesful retrieval, the content is saved into the page object and parsed with the JSoup document parser.

This parser returns a document object, which allows for DOM node traversal. We use this to extract links and images.

### 4.6 Extracting images

Using the document object, we can extract <img>tags from the markup. This allows us to access the "href" attribute, which defines the image content. The "href" attribute usually provides us with an absolute or relative link to the image location. By using the Java URL object we can easily construct a new target url for the image. Sadly, due to similar limitations already mentioned in the "Document check" subsection, we don't actually load the data. The image entry is constructed solely for statistical purposes.

A different scenario happens, when a data URI is present in the "href" attribute. In this case, all the necessary binary data is already provided in the "href" attribute itself and no additional loading is required.

### 4.7 Limiting page number

All the above happens every time a frontier url is provided. For each frontier url a page instance exists in the database. The next subsection describes the retrieval of new frontier urls and creation of their page instances. Since we limited ourselves to one hundred thousand pages, the next subsections are skipped, if we already have that many instances in the database. This info is provided by the SiteFrontierHandler component, which does this check everytime it loads new frontier urls.

Here we also check if a crawl delay is set for the site. If not, we apply a 4 second delay by default.

### 4.8 Extracting links

If the page quouta isn't fulfilled yet, we can proceed with creating new frontier urls. We retrieve those by parsing the html markup. Again, the JSoup Document object is used to filter out the correct DOM nodes. We search for <a>tags and "onclick" attributes.

Link tags (<a>) contain the link information in the "href" attribute. We construct a new frontier url as we already described in the "Extracting images" subsection.

Attributes of type "onclick" have their url defined in the following ways: onclick="location.href='<url>'" or onclick="document.location='<url>'". Once the url part is extracted, it will be constructed into a frontier url.

### 4.9 New frontier page instances

After all links are extracted, their page instances have to be created. First we filter out all urls, that don't have a seed site as their domain. Next, to avoid duplicate performance errors, if any of the extracted links in the page already has a page object in the database, we load them.

For those that aren't saved in the database yet, we create a new page entry in the database. We assign it a page type of frontier and time of creation. The time of creation is necessary, because it allows us to track when a link was extracted. SiteFrontierHandler always retrieves the oldest frontier page entries. Thus we have implemented bread-first searching of the pages.

### 4.10 Linking pages

Finally, a new link entry is created between the frontier page entry and all extracted frontier page entries. Both newly created and old page instances are included in this. What this provides us with, is a graph of how the pages are connected between themselves.

## 5 Results

Retrieval started at 5:00. It ended when we reached one hundred thousand non frontier pages at 00:35. The last frontier page was retrieved at 11:27, at that point there were 16199 non frontier pages. Interestingly there were 380 out of 383 binary pages parsed at that point. Which is a vast majority.

### 5.1 Sites

In the instructions, it was stated, that we should only save pages, that use the .gov.si as the top level domain. This filtering was implemented by allowing only pages that contain this string. This kind of filtering doesn't seem to be perfect, since it allowed some other domains into our collection of pages. Those included the target top level domain in query parameters. This would be solved by better filtering.

**Table 2.** Pages per site

| Site domain | Number of collected urls |
|---|---|
| evem.gov.si | 85665 |
| podatki.gov.si | 9163 |
| e-uprava.gov.si | 4163 |
| www.e-prostor.gov.si | 594 |
| www.projekt.e-prostor.gov.si | 211 |
| www.linkedin.com | 69 |
| www.facebook.com | 68 |
| twitter.com | 68 |
| e-prostor.gov.si | 2 |
| www.entrust.net | 1 |

### 5.2 Pages

The goal here was to reach one hundred thousands parsed pages or pages that were attempted to be parsed. Pages, that encounter errors are also included here. Note that this are only pages, that fit the above mentioned criteria.

While one hundred thousand pages were attempted to be parsed, a bunch failed the parsing. We detected errors while loading or attempting to retrieve the response content. Timeouts were the biggest issue here. Some pages weren't allowed to be parse, according to robots.txt rules.

Looking at the results, there was actually no saved html content at all. This happened due to a bug, that was introduced while the crawler was tested for longterm stability. All the links and images remain detected, but the markup that held them, is missing.

**Table 3.** Pages by page type

| Page type | Number of collected urls |
|---|---|
| FRONTIER | 7178 |
| HTML | 99621 |
| BINARY | 383 |

**Table 4.** Error pages

| TYPE | NUMBER |
|---|---|
| Retrieval error (ERROR) | 16141 |
| Not allowed (UNALLOWED) | 8898 |
| Other | 82143 |

**Table 5.** Retrieval error pages by site

| SITE | NUMBER |
|---|---|
| evem.gov.si | 15325 |
| e-uprava.gov.si | 375 |
| podatki.gov.si | 259 |
| www.projekt.e-prostor.gov.si | 122 |
| www.e-prostor.gov.si | 59 |
| twitter.com | 1 |

**Table 6.** Unallowed pages by site

| SITE | NUMBER |
|---|---|
| podatki.gov.si | 8557 |
| e-uprava.gov.si | 131 |
| www.linkedin.comtwitter.com | 69 |
| www.facebook.com | 68 |
| twitter.com | 67 |
| www.e-prostor.gov.si | 5 |
| www.entrust.net | 1 |

### 5.3 Binary files

**Table 7.** Binary files by type

| TYPE | NUMBER |
|---|---|
| PDF | 325 |
| DOC | 42 |
| DOCX | 27 |
| PPT | 1 |
| PPTX | 0 |

### 5.4 Images

**Table 8.** Image stats

| TYPE | NUMBER |
|---|---|
| Number of images | 459152 |
| Average images per non-error html page | 459152 / 74965 = 6.125 |
| Most images on one page | 70 |

### 5.5 Links

**Table 9.** Link stats

| TYPE | NUMBER |
|---|---|
| Number of links | 180536 |
| Average links per non-error html page | 180536 / 74965 = 2.409 |
| Most link from one page | 5813 |
| Most link to one page | 2004 |

Interestingly, there is a website, that contained more than five thousand links. Next best one contained around one thousand links. Upon further inspection the vast majority of the targeted pages aren't allowed as target. In fact, 5758 pages of the 5813 pages aren't permitted to be access by a web crawler.

## 6 Conclusion

While the majority of the requested features were implemented, this project still couldn't be finished completely. Storage of binary data and url canonicalization are the main features, that were missing.

Html content saving was implemented at some point, but when running the crawler to one hunder thousand pages, it somehow got disabled. As proof of content being retrieved, we have all the new pages that were discovered. But sadly no content.

Performance wise, the web crawler did well. At least compared to the Selenium performance noted in the instructions. Granted, if we didn't render JavaScript on every single page, it would gain a lot more performance.

The most successful part of this project would be the handling of concurrency. Conflicts in the database and url handling are minimal, which means that the performance doesn't suffer from it.