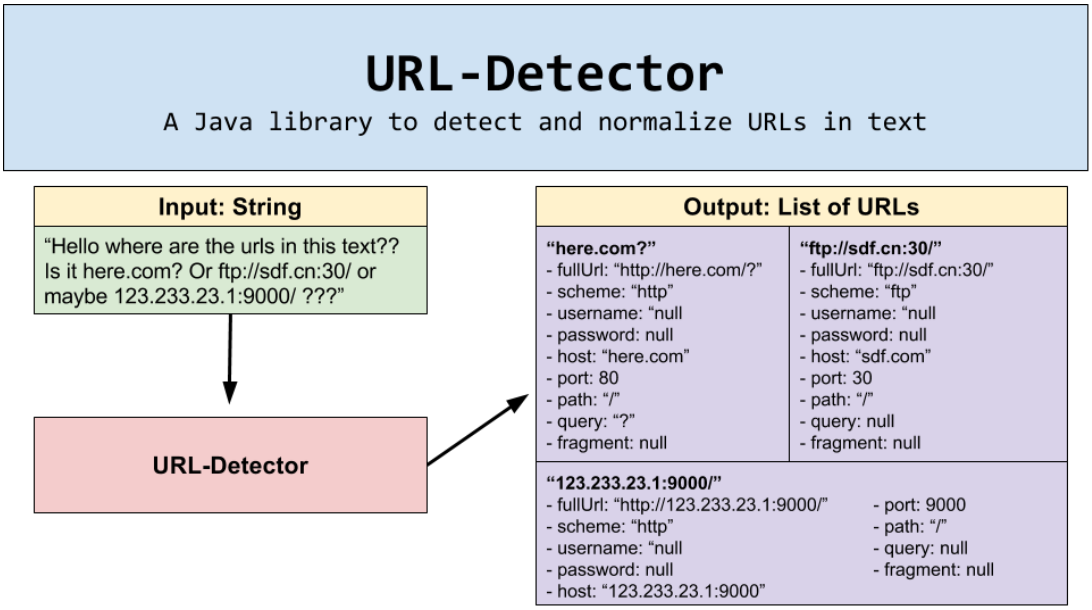
Open Sourcing URL-Detector

A Java Library to Detect and Normalize URLs in Text

Tzu-Han Jan June 30, 2016



Today, we're excited to share that Linkedin is open-sourcing our URL-Detector Java library. LinkedIn checks hundreds of thousands of URLs for malware and phishing every second. In order to guarantee that our members have a safe browsing experience, all user-generated content is checked by a backend service for potentially dangerous content. As a prerequisite for us to be able to check URLs for bad content at this scale, we need to be able to extract URLs in text at scale.

URLs get sent to our service in two different ways:

- As a single URL
- As a large piece of text

If it is sent as a single URL, we proceed to check the URL through our content-validation service. If it is sent as a large piece of text, we run our URL-Detector algorithm to try to search the text for any potential URLs. Before we proceed in visiting how this URL-Detector works and what functionalities it can provide, let's visit the motivation behind this project.

We want to detect as many malicious links as possible and to do this, we do not want to limit ourselves to checking URLs as defined in RFC 1738 and instead define a URL to be anything that can resolve into a real site when typed into the address bar of a browser. The browser address bar's definition of a URL is very loosely defined while the RFC is very strict. And of course, there are many browsers and different browsers have different behaviors, so we try to find text that would resolve in the most popular ones. Thus, it wasn't as simple as following the grammar defined in the RFC.

Initially, we started out with a solution based on regular expressions. It detected many potential URLs, many of which were real intentional URLs, many of which were not,

and many of which we have missed. It was a very iterative process to catch more URLs. It would start out with something as innocent as:

```
Regex:
(ftp|http|https):\/\/(\w+:{0,1}\w*@)?(\S+)(:[0-9]+)?(\/|\/([\w#!:.?+=&%@!\-\/]))?
```

Then, what if you wanted to detect URLs that may not contain a scheme? This is one of the many examples where browser address bars resolve a URL that does not fit the RFC.

```
Regex:
((ftp|http|https):\/\/)?(\w+:{0,1}\w*@)?(\S+)(:[0-9]+)?(\/|\/([\w#!:.?+=&%@!\-\/]))?
```

We ended up with this mess:

```
Regex:
((((f|ht)tps?:)?//)?([a-zA-Z0-9!#$%&'*+-/=?^_`{|}~]+(:[^ @:]+)?@)?((([a-zA-Z0-9\\-]
{1,255}|xn--[a-zA-Z0-9\\-]+)\\.)+(xn--[a-zA-Z0-9\\-]+|[a-zA-Z]
{2,6}|\\d{1,3})|localhost|(%[0-9a-fA-F]{2})+|[0-9]+)(:[0-9]{1,5})?([/\\?][^ \\s/]*)*)
```

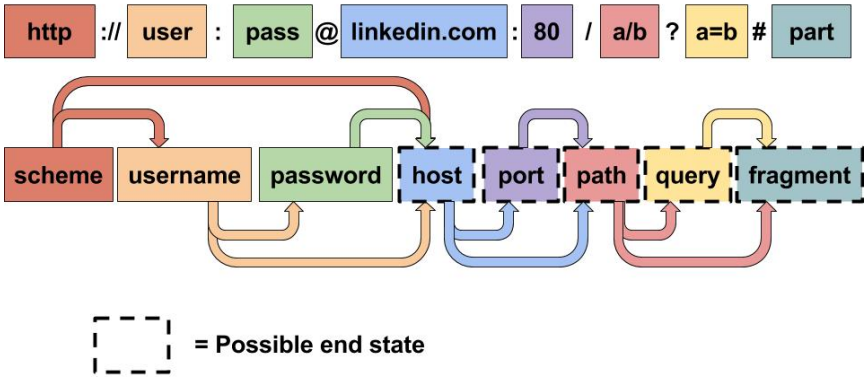
As you can see, every single little edge case adds at least a few more characters to the regex. You can see how complicated this topic can get here. (Warning: some of these regexes are over 5500 characters long.) We found that many of these detected potential URLs were technically URLs, but the result was not very useful to us. The more flexible we made our regex, the more false positives we found; the less flexible, the more false negatives we missed. For example, we would find "URLs" like "1.2". Finding way more URLs than we needed put unnecessary stress on our system.

Since we were finding too many matches, the approach we took to reduce the number of matches was to try to find false positives within our matches. Editing the original regular expression has proven to be extremely difficult without introducing even more false positives. Therefore, we needed multiple regexes. Here's a sample of one of the extra regexes we needed for excluding “localhost” and any combination of digits separated by dots except for ones in the form of an IPv4 address.

```
Blacklisted Regex: ^((\\d+(\\.\\d+){0,2})|(\\d+(\\.\\d+){4,})|localhost)$
```

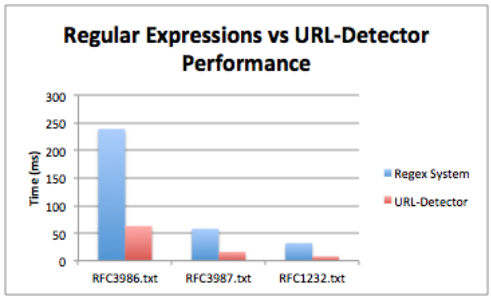
The result is that parsing large pieces of text started to take a long time, even on the order of seconds. In a system where we're trying to parse hundreds of thousands of URLs a second, this solution cannot scale. We found that regular expressions are good for matching, but not for parsing. And thus, we arrived at the URL-Detector library.

Instead of using regular expressions, we hand-built a finite state machine to parse out URLs in text. A finite state machine is a system consisting of a set of states where each state can transition into other states depending on the input event. In this case, the input event is the current character in the text we are parsing. You can learn more about it here.



This finite state machine has a couple of states, mostly based on the parts of a URL. The state is kept by a series of boolean variables, shifting from state to state, consuming one character at a time. Fortunately for performance, this finite state machine is represented by a topological ordering, where there are no arrows from one state pointing to a previous state. An example of this is that if you have already detected the "/" after the host, the engine no longer needs to try to trace the scheme since it is past that point. If at any point the state machine hits an unexpected character, it will return with the latest possible end state and restart the algorithm. The trickiest part about this is matching characters that actually have the possibility of being in multiple states. One example of this would be the colon. It can appear in at least three places: after the scheme, between username and password, and between host and port. It gets even more complex when we start considering IPv6, since IPv6 addresses also include colons. Backtracking primarily happens in some odd cases, like when the text contains a sequence of non-whitespace characters containing multiple colons, whereas regular expressions backtrack very often. Thus, the average runtime is significantly improved.

Here are some statistics on performance improvement:



Functionalities of this library

It is able to find and detect any URLs such as:

- **HTML 5 Scheme** – //www.linkedin.com
- **Usernames** – user:pass@linkedin.com
- **Email** – fred@linkedin.com
- **IPv4 Address** – 192.168.1.1/hello.html
- **IPv4 Octets** – 0x00.0x00.0x00.0x00
- **IPv4 Decimal** – http://123123123123/
- **IPv6 Address** – ftp://[::]/hello
- **IPv4-mapped IPv6 Address** – http://[fe30:4:3:0:192.3.2.1]/

As an additional bonus, it is also able to identify the parts of the identified URLs. For example, for the URL http://user@example.com:39000/hello?boo=ff#frag, it is able to

identify the following parts:

- Scheme – "http"
- Username – "user"
- Password – null
- Host – "example.com"
- Port – 39000
- Path – "/hello"
- Query – "?boo=ff"
- Fragment – "#frag"

It is also able to handle quote matching and HTML input. Depending on your input string, you may want to handle certain characters in a special way. For example if you are parsing HTML, you probably want to break out of things like quotes and brackets. For example, if your input looks like:

linkedin.com, then you probably want to make sure that the quotes and brackets are extracted. For that reason, this library has the capability to change the sensitivity level of detection based on your expected input type by functioning in different modes as specified in the UrlDetectorOptions java class. This way you can detect **linkedin.com** instead of **linkedin.com**.

Using this library

To use this library, simply clone the GitHub repository and import the URL-Detector library. Here's an example of how you can use it:

```
   import com.linkedin.urls.detection.UrlDetector;
   import com.linkedin.urls.detection.UrlDetectorOptions;
   ...
   UrlDetector parser = new UrlDetector("hello this is a url Linkedin.com",
UrlDetectorOptions.Default);
    List<Url> found = parser.detect();

    for(Url url : found) {
        System.out.println("Scheme: " + url.getScheme());
        System.out.println("Host: " + url.getHost());
        System.out.println("Path: " + url.getPath());
    }
```

For more information, take a look at the "How to Use" section in the Readme.

Acknowledgements

Special thanks to Vlad Shlosberg and Yulia Astakhova for contributing heavily to this library.

Conclusion

Let us know through GitHub if there are any improvements we can make to this library. Also, feel free to contact any of us if you have any questions. We'd love for you to share how you use this library. Happy detecting!

Topics

Java, Open Source, Security