International Journal of Electrical and Computer Engineering (IJECE)
Vol. 9, No. 4, August 2019, pp. 3185~3193
ISSN: 2088-8708, DOI: 10.11591/ijece.v9i4.pp3185-3193

A regexcriteria api to complete the power of regular expressions engine

Boulchahoub Hassan, Rachiq Amina, Labriji Amine, Labriji Elhoussine, Mohamed Azouazi
Department of Mathematics and Computer Science, Faculty of Sciences Ben M'SIK, Morocco

Article Info

Article history:

Received Aug 21, 2018
Revised Mar 21, 2019
Accepted Apr 4, 2019

Keywords:

Criteria API
Finite automata
Programming API
Regular expression

ABSTRACT

Regular expressions are heavily used in the field of computer programming. They are known by their strength to search or replace parts of strings according to a given structure (mails, phone, numbers, etc.). Currently regular expressions are only used to search for some patterns or to make some substitutions in strings. However, the need may be wider than that when it comes to order the results of a regular expression or to group them according to some criteria. Developers are always called to analyze the results of a regular expression by doing some restrictions such as (equal, not equal, between) or some projections like (maximum, average, grouping by) or sorts. Unfortunately, to do these treatments, the developer must implement his own algorithms which cost him a remarkable effort and a waste of time. We propose in this paper an API called RegexCriteria inspired from Hibernate Criteria to support developer while analyzing the results of a regular expression.

Corresponding Author:

Boulchahoub Hassan,
Department of Mathematics and Computer Science,
Faculty of Sciences Ben M'SIK,
Casablanca, Morocco.
Email: hboulchahoub@gmail.com

1. INTRODUCTION

Regular expressions consist of a set of characters and symbols used to match one or more patterns in strings or files. The IT experts give them a great importance due to their strong utility such as user data validation, network data filtering, data extraction, intrusion detection etc. In the world of computer programming, mastering regular expressions is a powerful tool that can help to create programs easily and quickly. It is clear that a properly defined regular expression can replace multiple lines of code, multiple loops, or even an entire program. Regular expressions are widely used in computer fields and especially when using programming languages. According to the studies of Chapman C and Stolee KT [1], 50% of professional developers compose regular expressions at least once a week and 42% of the projects studied contain regular expressions. The increasing use of regular expressions has led many researchers to consider them as the domain of their work. Wüstholz V [2] proposed a tool called "Exploiter" to secure applications that use regular expressions. Cochran RA [3] offered a tool called "CROWDBOOST" to help developers find the most appropriate regular expression. Bartoli A [4] has tried to reduce the complexity of creating regex by proposing an automatic generation. Spishak E [5] proposed to validates regular expression syntax and capturing group at compile time instead of at run time.

Unfortunately, regular expressions only find specific pieces in a string and never act on the found values. For example, using only regular expressions, we can not calculate the average, the max, or the min of digits found, we can not compare the results found, we can not do filters, we can not do sorting. Some symbols must be added to the dictionary of symbols to perform these tasks automatically. Today, to implement this type of needs, developers must use both regular expressions and a programming language and must create several lines of code. It is clear, then, that the current implementation of these needs will takes a lot of time for developers, it will generate several useless lines of code, and usually we found a completely different code instructions to solve the same problem.

Developers know that using an already tested API or framework greatly minimizes the risk of bugs and improves the readability, the quality and the homogeneity of programs. In this paper, we propose an API called Regex Criteria to simplify, standardize, reduce the programming lines used to complete the task of a regular expression. We are inspired by the success of the Hibernate Criteria API [6-8] to standardize all aggregations, transformations, reductions, restrictions and all projections that developers must implement while using a regular expression. All programming languages have APIs for processing regular expressions [5, 9], but no API has provided for the already mentioned processing. For this reason, several researchers have done work to complete the lack of APIs currently offered by programming languages. Spishak E proposed an annotation-based API to help the developer find the errors of regular expressions during the compilation phase of a program [10]. Other works aimed at helping the developer to simplify, validate or automatically generate regular expressions [2-4]. All features built into our Regex Criteria API will be illustrated by examples implemented using the Java language syntax.

This paper is organized as follows. Section 2 tells about the history of regular expression and its symbols. Section 3 presents two programs using the regular expressions. As for section 4, it talks about the limitations of current API dealing with the regular expressions. Related works are illustrated at Section 5. Section 6 briefly describe some features of Regex Criteria API. The last section contains final conclusions and points to further work.

2. THE REGULAR EXPRESSION

A regular expression is a string of text that defines the format of a set of strings (email format, phone number format, IP address format, etc.). Regular expressions are useful for finding patterns in strings and possibly replacing them with another pattern. Since "regular expressions" is a mouthful, we will usually use the term abbreviated "regex". In the next section we will briefly discuss the story behind the popularity of regex.

2.1. Regular expression history

The idea of regular expressions was discovered by Warren McCulloch and Walter Pitts in 1943 through the publication of their article "A logical calculation of ideas immanent to nervous activity" in the Bulletin of Mathematical biophysics [11]. In 1956, the mathematician Stephen Kleene developed a model that presents a simple algebra in the article "Representation of events in networks of nerves and finite automata" [12]. At that time the terms regular sets and regular expressions were born. In 1968, Ken Thompson published the article "Regular Expression Search Algorithm" [12, 13] to describe a regular expression compiler. He was based on the Kleene notation to create the QED editor to help users search through text files. The work of Ken Thompson has made it possible to do global search in Unix files using the \Global\Regex\Print (GREP) commands, which are currently well known. From the 80s, regular expressions began to gain great popularity in the world of computer programming. Thanks to the multiple works of Henry Spencer, Perl language is considered among the first languages that have invested a lot in the field of regular expressions.

2.2. Regular expression symbols

Several special characters are used to build a regex, Paul Boersma and David Weenink gave a good explanation to these symbols with simple examples to facilitate understanding [14]. In the following Table 1 we summarize the meaning of each character.

2.3. Regular expression examples

Expressions are now available on the internet. They are even integrated with all development frameworks to help developers validate data entered by users and also to help them extract data according to the context of their work. We give below some examples with the necessary explanations in order to initiate the global structure of a regular expression. It is important to note that the world of regular expression is not totally accurate. Thus, the examples below can admit several regular expressions according to the simplicity and precision desired. Always the symbol **(^)** denotes the beginning of a string and **($)** denotes its end.

Table 1. Regular expression symbols

	The meaning	Examples
\	Gives special meaning to the character following it	\w a word. \n a new line.\+ disable the caracter +
^	Start of the string, or the negation symbol	^c matches c at the start of the string. [^0-9] matches any non digit
$	End of the string	e$ matches "e" at the end of a line
{ }	A Range quantifier	b{2,3} matches "bb" or "bbb"
[ ]	Defines a character class to match a *single* character	[a-c] matches "a", "b" or "c"
( )	Used for grouping characters	(ee)\1 matches "eeee"
.	any character except the newline symbol	.b two consecutive characters where the last one is "b" .*\.pdf$ all strings that end in ".pdf"
*	Zero or more quantifier	^.*$ matches an entire line
+	One or more quantifier	\d+ one or more digits \w+ one or more words
?	Zero or one quantifier. ? is Also used in special constructs with parentheses.	The same as {0,1} ab?c looks for "a" followed by zero or one "b", and then "c"
|	Series of alternatives	"(a|b|c)b" matches "ab" or "bb" or "cb"
< >	Left or right word boundary. \b can be used like a shorter way for matching boundaries.	[[:<:]]cat matches "cat"in "catom" cat[[:>:]] matches "cat"in tomcat [[:<:]]cat[[:>:]] matches only "cat"
-	Indicates a range in a character class	[A-Z] matches any uppercase character
&	Substitute complete match	Replacing \d+ with [\&] in 1a2b yields [1]a[2]b

2.3.1. Email regex

Email validation, has been the subject of several Regex because of the continuous search for simplicity and precision. To give introductory explanations, let's take the following Email's expression.

^[_A-Za-z0-9-\\+] + (\\. [_A-Za-z0-9-] +) *@[A-Za-z0-9-] + (\\. [A-Za-z0-9] +) *(\\. [A-Za-z] {2,}) $

This regex is divided in Table 2 into several parts to give the necessary explanations.

Table 2. Email regular expression description

Email Regular Expression	Description
[_A-Za-z0-9-\\+] +	An Email must start with a character in the bracket [ ] and it must contains one or more characters (+)
(\\. [_A-Za-z0-9-] +) *	(→ Starting the first group \\. [_A-Za-z0-9-] + → Follow by a dot "." and character in the bracket [ ], must contains one or more characters (+)) * → Ending the first group, this group is optional (*)
@[A-Za-z0-9-] +	@ → An Email must contain a "@" symbol [A-Za-z0-9-] + → Follow by character in the bracket [ ], must contains one or more characters (+)
(\\. [A-Za-z0-9] +) *	(→ Starting the second group \\. [A-Za-z0-9] + → Follow by a dot "." and character in the bracket [ ], must contains one or more characters (+)) * → Ending the second group, this group is optional (*)
(\\. [A-Za-z] {2,})	(→ Starting the third group \\. [A-Za-z] {2,}: Follow by a dot "." and character in the bracket [ ], with minimum length of 2) → Ending the third group

2.3.2. Phone number regex

Another case of using the regex is the validation of the phone number formats. Again, identical to the case of email, several expressions are designed to check the phone numbers and also to replace them with the correct format defined by the regex. Imagine in this example that we want to determine if a user has entered a North American phone number and we want to create a regex that detects that numbers: 1235566880, 123-556-6880, 123.556.6880, 123 556 6880, (123) 556 6880 are valid and 35566880, 23.556.6880 are invalid. A regex proposition may look like this.

^\ (? ([0-9]{3}) \)? [-.]? ([0-9]{3}) [-.]? ([0-9]{4}) $

This regular expression checks if a number contains exactly three groups of digits. A first group of exactly 3 digits that can possibly be surrounded by parenthesis, a second group of exactly 3 digits followed by a choice of three separators (hyphen, point or space) and a last group of exactly 4 digits followed also by a choice of three separators (hyphen, point or space). More explanations for this expression are given in Table 3.

Table 3. Phone regular expression description

Phone Regular Expression	Description
\(?	\ (→ Match a literal "(".? → zero and one time.
([0-9]{3})	(→ Start capturing group 1. [0-9] Match a digit. {3} Exactly three times.) → End capturing group 1.
\)?	\) → Match a literal ")".? → zero and one time.
[-.]?	Match dot, hyphen or a space.? → zero and one time.
([0-9]{3})	(→ Start capturing group 2. [0-9] → Match a digit. {3} → exactly three times.) → End capturing group 2.
[-.]?	Match dot, hyphen or a space.? → zero and one time.
([0-9]{4})	(→ Start capturing group 3. [0-9] → Match a digit. {4} → exactly three times.) → End capturing group 3.

2.3.3. URL regex

The URL validation is another highly used example in the http communication protocol which is also based on regex. The expression explained in Table 4 verifies whether an URL is valid or not.

```
/^(https?:\/\/)?([\da-z\.-]+)\.([a-z\.]{2,6})([\/\w \.-]*)*\/?$/
```

Table 4. URL regular expression description

URL Regular Expression	Description
(https?:\/\/)?	The String "http://" or "https://" at the beginning. This String is optional (?).
([\da-z\ .-]+)	Numbers (\d), Letters (a-z), Dots (.) or Hyphens (-) one or more time (+)
\.([a-z\.]{2,6})	A dot followed by two to six letters or dots.
([/\w \.-]*)*/?:	Zero or more Letters, Numbers, Underscores, dots or hyphens with an optional forward slash.

3. SOME SIMPLE PROGRAMS USING REGEX

Imagine that we ask a developer to calculate the average of the call durations made each month and we give him the data in a file as shown in Table 5.

Table 5. The average of the call durations

Phone Number	Total Call Duration by Month (TCD.Mx)									
	TCD M1	TCD M2	TCD M3	TCD M4	TCD M5	TCD M6	TCD M7	TCD M8	TCD M9	TCD M10
1235566880	10	20	11	22	34	22	12	40	30	21
1234446878	13	15	10	09	12	30	23	45	12	45
1235557080	13	15	10	09	12	15	22	11	22	33
1235512553	10	11	12	13	13	14	22	43	10	31
1235235531	12	08	12	11	05	22	11	23	11	11
1235445537	09	11	20	09	05	10	21	22	11	09
1235665532	11	08	02	21	15	20	11	33	22	11

If the developer has chosen to use the regex in this problem, he will probably build a regex to represent the structure of an entire line using the named groups to represent each column. Here is a possible proposal for this regex if we consider [\s] as separator.

```
\b(?<phone>\(?([0-9]{3})\)?[-. ]?
([0-9]{3})[-. ]?([0-9]{4}))[\s]
(?<HM1>\d+)[\s]
(?<HM2>\d+)[\s]
(?<HM3>\d+)[\s]
(?<HM4>\d+)[\s]
(?<HM5>\d+)[\s]
(?<HM6>\d+)[\s]
(?<HM7>\d+)[\s]
(?<HM8>\d+)[\s]
(?<HM9>\d+)[\s]
(?<HM10>\d+)\b
```

We note that using the named group to represent a column in this file will be very useful for finding the values and knowing their type. It is partly the objective of the declaration of a variable in the programming languages. For instance, we can easily deduce from the previous regular expression, the following statements.

```
\b(?<phone>\(?([0-9]{3})\)?[-. ]?
([0-9]{3})[-. ]?([0-9]{4}))[\s] ⇔ String phone;
(?<HM1>\d+)[\s] ⇔ Long HM1;
(?<HM2>\d+)[\s] ⇔ Long HM2;
(?<HM3>\d+)[\s] ⇔ Long HM3;
(?<HM4>\d+)[\s] ⇔ Long HM4;
(?<HM5>\d+)[\s] ⇔ Long HM5;
(?<HM6>\d+)\b ⇔ Long HM6;
```

This technique will help us to know the possible operations relating to a regex group. It does not make sense, for instance, to do Max or Min for the "phone" group in our example because its type is a String. A possible program to meet the need illustrated in this example is given in Table 6 and the results obtained are also shown in Table 7.

Table 6. Program to calculate the average of regex's results

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Program1 {
public static void main(String[] args) {
  String regex = "\\b(?<phone>\\(?([0-9]{3})\\)?"
                + "[-. ]?([0-9]{3})[-. ]?([0-9]{4}))[\\s]"
                + "(?<HM1>\\d+)[\\s]"
                + "(?<HM2>\\d+)[\\s]"
                + "(?<HM3>\\d+)[\\s]"
                + "(?<HM4>\\d+)[\\s]"
                + "(?<HM5>\\d+)[\\s]"
                + "(?<HM6>\\d+)\\b";
 Pattern p = Pattern.compile(regex);
 int count = 0;
 double sum = 0;
 for (int i = 1; i < 7; i++) {
                for (String line : args) {
                 Matcher m = p.matcher(line);
                  while (m.find()) {
                   sum += Integer.parseInt(m.group("HM" + i));
                                count++;
                                } // End while
                } // End 2sd for
  System.out.println("the HM" + i + " average is :" + (count == 0 ? 0 : sum / count));
 } // End 1st for
 }// End main
} // End Class Program 1
```

Table 7. Results of calculating the average of regex's results

```
the HM1 average is:11.142857142857142
the HM2 average is:11.857142857142858
the HM3 average is:11.571428571428571
the HM4 average is:12.035714285714286
the HM5 average is:12.371428571428572
the HM6 average is:13.214285714285714
```

Another example called "Word Count" which counts the number of occurrences for each word in a file stored in the Hadoop HDFS [15] is very evoked when it comes to the parallel processing done by MapReduce [16]. In this paper and as shown in Table 8 we parse the strings without having concerns about parallel processing. The use of our RegexCriteria API in the context of parallel processing will be the subject of another paper.

Table 8. Program and results of calculating occurrences for each word in a String

```
import java.util.HashMap;
import java.util.Map;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class Program2 {
        public static void main(String[] args) {
                Map<String, Integer> occurrences = new HashMap<String, Integer>();
                Pattern pword = Pattern.compile(args[0]);
                String source = args[1];
                Matcher mword = pword.matcher(source);
                while (mword.find()) {
                        String word = mword.group();
                        if (occurrences.containsKey(word))
                                continue;
                        Pattern p = Pattern.compile(word);
                        Matcher m = p.matcher(source);
                        int count = 0;
                        while (m.find()) {
                                count++;
                        }
                        occurrences.put(word, count);
                }
                System.out.println(occurrences);
        }
}
```
```
Results: {A=2, B=1, C=3, D=1, E=1, F=2, M=1}
```

Even though these two programs are very simple, the number of lines they contain is very high compared to the task they perform. In addition to this, they depend a lot on how the developer thinks. To have a homogeneous code understood by everyone, the developer must be guided by an API to write the same instructions for the same problem. Some limitations of the current APIs are discussed in the next section.

4. SOME LIMITATIONS OF CURRENT REGEX APIs

All current programming languages have integrated some APIs to handle regular expressions and to simplify the work of developers and especially to make their code simple and readable [17]. But unfortunately, the developer still has to make efforts to design and implement missing algorithms. As shown in the two previous examples, to calculate the occurrences or the average of the numbers found in a String, the developer must design his own algorithm and must implement it in several lines of code with possible errors. In addition, no API has integrated tools to help the developer build his regular expressions despite the complexity of this task. In this work we propose to supplement the current APIs with highly requested functions that are implemented in a repetitive way in programs using regular expressions. Most of the methods proposed in Section 6 are inspired from Hibernate Criteria API [18] and mentioned in Table 9. To motivate the usefulness of this API, we first propose to study the previous works and then describe the strength of RegexCriteria API in processing regex's results and we will focus later on how this API can avoid the weaknesses encountered by developers when using the regex in programs.

Table 9. Features added to RegexCriteria API

	Current Regex APIs	Regex Criteria API
Narrowing the result	×	√
Ordering the results	×	√
Grouping results	×	√
Projections	×	√
Aggregations	×	√

5. RELATED WORKS

The regular expression language has undergone several enhancements and extensions from 1943 until today. Spishak E [10] remarked that errors due to the structure of a regex in a program will not be detected until the execution time. So, he proposed to validate regular expression syntax and capturing group at compile time instead of at run time. Chapman C and Stolee KT [1] have studied 4000 Open Source projects to measure the use of Regular expressions. They showed that 50% of professional developers compose regular expressions at least once a week and that 42% of the projects studied contain regular expressions. These results imply an urgent need for refactoring regex and creating tools to better support developers when using regex. It is true that regular expressions help to implement several use cases in a program, but their use can give rise to vulnerability risks. Wüstholz V [2] used a tool called "Exploiter" to find security vulnerabilities caused by regex in Java Web applications. In addition to those works, Cochran RA [3] tells the story of a regex collection experiment to validate the URL structure on the web [19]. The leader of this experiment showed that the majority of the proposals did not cover all possible cases of URLs. Thus Cochran RA [3] offers a tool called CROWDBOOST to help developers find the most appropriate regular expression. Also, Bartoli A [4] has tried to reduce the complexity of creating regex by proposing an automatic generation based on a set of examples entered by the developer.

6. FEATURES OF THE PROPOSED “REGEX CRITERIA” API

To illustrate some methods of the API Regex Criteria we consider the regular expression which represents the phone numbers with three groups "PART1", "PART2" and "PART3" as following.

```
String regexPhone = "\\b\\(?
(?<PART1>[0-9]{3})\\)?[-. ]?
(?<PART2>[0-9]{3})[-. ]?
(?<PART3>[0-9]{4})\\b";
```

The following String is considered as a source for doing the necessary tests.

```
String source = "123.344.5678 2337681234 dd 345-908-1234 fff 200-908-1234 2345435000";
```

To use the features of this API, we must first create a Regex Criteria object.

```
RegexCriteria regexCriteria = RegexCriteria.create(regexPhone, source);
```

6.1. Some restrictions using regex criteria API

In the code shown in the Table 10, we treat the following restrictions "*Not Equal*", "*Equal*", ”Less *Than*”, “*Greater Than*”, “*IN*” using RegexCriteria.

Table 10. Some restrictions made by regex criteria

// list of elements matching the regex with “part1” is different from "123" *List<String> result=regexCriteria.add(RegexRestrictions.ne("PART1", "123")).list();*
// list of elements matching the regex with “part1” is equal to "123" *List<String> result=regexCriteria.add(RegexRestrictions.eq("PART1", "123")).list();*
// list of elements matching the regex with “part1” Is less than "123" *List<String> result=regexCriteria.add(RegexRestrictions.lt("PART1", "123")).list();*
// list of elements matching the regex with “part1” Is greater than "123" *List<String> result=regexCriteria.add(RegexRestrictions.gt("PART1", "123")).list();*
// list of elements matching the regex with “part1” Is equal to "123" or “400” *List<String> result=regexCriteria.add(RegexRestrictions.in("PART1", {"123","400"})).list();*

6.2. Some aggregations and projections using regex criteria API

In the code shown in the Table 11, we treat the following aggregations "*Maximum*", "*Average*", ”Minimum”, “*Grouping by*”, “*Counting*” using RegexCriteria. It is quickly remarkable that each program illustrated in section 3 can be replaced by only one line as shown in the instructions noted by (***) in Table 11.

Table 11. Some aggregations and projections made by regex criteria API

*** // Calculate the average of digits found by the regex. Only digits of the group "Part1" are used double avg=regexCriteria.setRegexProjection(RegexProjections.avg("PART1")).value();
// Find the maximum of digits found by the regex. Only digits of the group "Part1" are used double max=regexCriteria.setRegexProjection(RegexProjections.max("PART1")).value();
// Find the minimum of digits found by the regex. Only digits of the group "Part1" are used double min=regexCriteria.setRegexProjection(RegexProjections.min("PART1")).value();
// Calculates the sum of digits found by the regex. Only digits of the group "Part1" are used double sum=regexCriteria.setRegexProjection(RegexProjections.sum("PART1")).value();
*** // Count the number of occurrences for each element found by the regex. List list = regexCriteria.setRegexProjection(RegexProjections.projectionList().add(RegexProjections .count()).add(RegexProjections.groupBy("PART1"))).list();
// count the number of elements found by the regex int count=regexCriteria.setRegexProjection(RegexProjections.count()).value();
// count number of distinct elements found by the regex int countDistinct=regexCriteria.setRegexProjection(RegexProjections.countDistinct()).value();

6.3. Ordering results by regex criteria API

Another need that developers implement in a repetitive way concerns the sorting of results according to different criteria and the limitations of the results to be displayed to the final user. We show in the Table 12, how to do sorting using RegexCriteria.

Table 12. Asc and desc sort made by regexcriteria

// Asc. Sort of the elements found by the regex. Only digits of the group "Part1" are concerned List<String> orderedAscResult = regexCriteria.addOrder(RegexOrder.asc("PART1")).list();
// Desc. Sort of the elements found by the regex. Only digits of the group "Part1" are concerned List<String> orderedAscResult = regexCriteria.addOrder(RegexOrder.desc("PART1")).list();

7. CONCLUSION

Regular expressions are widely used by developers to implement the different cases of their applications. For this reason, all programming languages have integrated tools and APIs to guide and support the developer when handling regex. Unfortunately, these APIs do not manage to process the results obtained by a regex to make, for example, restrictions, projections or sorting. In this paper, we proposed an API called RegexCriteria in which we have implemented all the features that a developer can use to act on the results of a regex. We believe that this API will be very useful in the parallel processing of Big Data and especially when creating MapReduce programs, so our next work will be focused on this issue.

REFERENCES

[1] Chapman C, Stolee KT., "Exploring regular expression usage and context in Python," In: *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*, [Online], 2016, Available from: http://dx.doi.org/10.1145/2931037.2931073.
[2] Wüstholz V, Olivo O, Heule MJH, Dillig I., "Static Detection of DoS Vulnerabilities in Programs that Use Regular Expressions," In: *Lecture Notes in Computer Science*, 2017, pp. 3–20.
[3] Cochran RA, D'Antoni L, Livshits B, Molnar D, Veanes M., "Program Boosting," *ACM SIGPLAN Notices*, vol. 50, pp. 1, pp. 677–88, 2015.
[4] Bartoli A, Davanzo G, De Lorenzo A, Mauri M, Medvet E, Sorio E., "Automatic generation of regular expressions from examples with genetic programming," In: *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion - GECCO Companion '12*, [Online]. 2012. Available from: http://dx.doi.org/10.1145/2330784.2331000.
[5] "7.2. re — Regular expression operations — Python 2.7.15 documentation" [Online]. Available from: https://docs.python.org/2/library/re.html, [cited 2018 Dec 26].
[6] Kisman, Kisman, Isa SM., "Hibernate ORM query simplification using hibernate criteria extension (HCE)," In: *2016 3rd National Foundation for Science and Technology Development Conference on Information and Computer Science (NICS)*, [Online], 2016. Available from: http://dx.doi.org/10.1109/nics.2016.7725656
[7] "Querying Objects with Criteria," In: *Pro Hibernate 3*. pp. 131–44.
[8] Linwood J, Minter D., "Advanced Queries Using Criteria," In: *Beginning Hibernate*, 2010, pp. 215–25.

[9] "Pattern (Java Platform SE 7),” [Online], 2018, Available from: https://docs.oracle.com/javase/7/docs/api/java /util/regex/Pattern.html, [cited 2018 Dec 26].

[10] Spishak E, Dietl W, Ernst MD., "A type system for regular expressions," In: *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs - FTfJP ’12* [Online], 2012, Available from: http://dx.doi.org /10.1145/2318202.2318207

[11] McCulloch WS., Pitts W., "A logical calculus of the ideas immanent in nervous activity," *Bull Math Biophys,* vol. 5, no. 4, pp. 115–33, 1943.

[12] Kleene SC., "Representation of Events in Nerve Nets and Finite Automata. In: *Automata Studies (AM-34)*, 1951.

[13] Thompson K., "Programming Techniques: Regular expression search algorithm," *Commun ACM*, vol. 11, no. 6, pp. 419–22, 1968.

[14] Regular expressions, [Online], Available from: http://www.fon.hum.uva.nl/praat/manual/Regular_expressions.html, [cited 2018 Aug 15].

[15] HDFS Architecture Guide, [Online], Available from: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, [cited 2018 Aug 18].

[16] MapReduce Tutorial, [Online], Available from: https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html, [cited 2018 Jul 31].

[17] java.util.regex (Java Platform SE 7), [Online], 2018, Available from: https://docs.oracle.com/javase/7/docs/api /java/util/regex/package-summary.html, [cited 2018 Aug 13].

[18] org.hibernate (Hibernate API Documentation), [Online]. Available from: https://docs.jboss.org/hibernate /core/3.2/api/org/hibernate/package-summary.html, [cited 2018 Aug 13].

[19] In search of the perfect URL validation regex, [Online], Available from: https://mathiasbynens.be/demo/url-regex, [cited 2018 Dec 26].