INSTITUTO POLITÉCNICO DE BRAGANÇA ipb Escola Superior de Tecnologia e Gestão

Kuban State Agrarian University

Critical data leak detection in institutions' public Web sites

Vasilenko Andrey Igorevich

Dissertation presented to the School of Technology and Management of Bragança to obtain the Master Degree in Information Systems, in the scope of the double diploma programme with the Kuban State Agrarian University.

Supervisors:

Prof. José Luís Padrão Exposto

Prof. Tiago Miguel Ferreira Guimarães Pedrosa

Prof. Tkachenko Vasiliy Vladimirovich

Bragança

2019-2020

INSTITUTO POLITÉCNICO DE BRAGANÇA ipb Escola Superior de Tecnologia e Gestão

Kuban State Agrarian University

Critical data leak detection in institutions' public Web sites

Vasilenko Andrey Igorevich

Dissertation presented to the School of Technology and Management of Bragança to obtain the Master Degree in Information Systems, in the scope of the double diploma programme with the Kuban State Agrarian University.

Supervisors:

Prof. José Luís Padrão Exposto

Prof. Tiago Miguel Ferreira Guimarães Pedrosa

Prof. Tkachenko Vasiliy Vladimirovich

Bragança

2019-2020

Abstract

Content of modern Web sites could be vulnerable to the data leaks, but could also already contain data leaks in itself, especially in the content of large institution's Web sites, where a lot of users have an access to large, constantly processed huge amounts of data, which can include sensitive data. Unlike content of databases the content of such Web sites are much less structured and therefore less trackable and even more vulnerable to leaks that could happen due to the human factor.

Most existing Data Leak Detection Systems are designed to detect data leaks on networks or in highly organized and structured systems like, for example, databases. During this work we will describe the process of creation of the multi-user Data Leak Detection System which will be capable of detecting critical types of data inside different institution's Web sites by using descriptive entities of such types received from users.

With this work we make a contribution to solving the problem of data leakage from educational institutions' Web sites by analyzing the problem and developing a Data Detection System capable of collecting data from Web sites independently of search engines and, with help of users, of detecting critical data types in the collected data, providing a user, on the end of detection process, with the basic type of the report, giving him the opportunity for further observation of the detected data in order to decide whether to remove those data from the corresponding Web pages or not.

Keywords: Data Leak Detection, Crawling, Nutch, Solr, Information Security, GDPR.

Resumo

O conteúdo de sites modernos pode ser vulnerável a fugas de dados, mas pode conter já fugas concretizadas, principalmente quando se trata do conteúdo de sites de grandes instituições, onde grandes quantidades de dados são frequentemente manipuladas e onde existe um grande número de utilizadores, e que geralmente processam dados confidenciais. Ao contrário, e.g das bases de dados, o conteúdo desses sites é muito menos estruturado e, menos rastreável e, portanto, ainda mais propicio a fugas de informação, que podem ocorrer não apenas devido a falhas técnicas, mas também por causa do fator humano.

A maioria dos sistemas de deteção de fugas de dados existentes é projetado para detetar fugas em sistemas altamente organizados e estruturados, como bases de dados ou redes. Neste trabalho, descreveremos o processo de criação de um sistema de deteção de fugas de dados multi-utilizador, capaz de detetar tipos críticos de dados em páginas de sites utilizando objetos descritivos desses tipos descritos pelos utilizadores.

Este trabalho contribui para a resolução do problema de fuga de dados dos sites de instituições de ensino, analisando o problema e desenvolvendo um sistema de deteção de dados que possa agregar dados de sites independentemente dos mecanismos de pesquisa e identificar dados críticos com a ajuda do utilizador - com a deteção de tipos críticos dados nos dados recolhidos, fornecendo no final um relatório do processo de descoberta, e criando a oportunidade de monitorizar ainda mais os dados detetados, com a finalidade de decidir se deve ou não remove-los das páginas da Web em que foram encontrados.

Palavras-chave: Deteção de fuga de dados, Crawling, Nutch, Solr, Segurança da Informação, RGPD.

Contents

1 Introduction **1**

1.1 Goal . . . 2

1.2 Methodology . . . 2

1.3 Document Structure . . . 3

2 Context and Technologies **5**

2.1 Node.js and Node Package Manager . . . 7

2.2 MongoDB . . . 8

2.3 Crawlers . . . 8

2.3.1 Scrapy . . . 9

2.3.2 Scrapestorm . . . 9

2.3.3 Octoparse . . . 10

2.3.4 80legs . . . 10

2.3.5 PySpider . . . 11

2.3.6 Apify Web Scraper . . . 11

2.3.7 Apache Nutch . . . 12

2.3.8 Crawler choice argumentation . . . 13

2.4 Full-Text Search Engines . . . 14

2.4.1 Apache Solr . . . 14

2.4.2 ElasticSearch . . . 15

2.4.3 Full-Text Search Engine choice argumentation . . . 15

3 Analysis and Approach **17**

3.1 Analysis 17

3.2 Approach 18

4 Components Deployment and System Implementation **25**

4.1 Nutch server 25

4.1.1 Setup 25

4.1.2 Nutch Data Concepts 26

4.1.3 Nutch Crawl Process 27

4.1.4 Nutch Configuration 28

4.1.5 Nutch Readdb Requests 31

4.2 Solr server 32

4.2.1 Setup 32

4.2.2 Functionality To Be Used 32

4.2.3 Indexing and Data Detection 33

4.3 Master Server (Node.js server) 35

4.3.1 MongoDB database 36

4.3.2 Server Initialization 36

4.3.3 Node.js NPM Modules 37

4.3.4 Node.js Core Modules 38

4.3.5 User Registration 38

4.3.6 User Authorization 41

4.3.7 User Accounts Updating and Deletion 43

4.3.8 User Datatypes 46

4.3.9 User Collections 48

4.3.10 Crawling 49

4.3.11 Crawldb Updating and Deletion 53

4.3.12 Critical Data Detection 55

4.3.13 Observing Detection Results 57

5 Tests and Results **61**
5.1 Testing of Preparatory Operations 61
5.1.1 Authorization 61
5.1.2 Crawling 64
5.1.3 Creating a Collection 67
5.2 Main operations testing 71
5.2.1 Indexing and Detecting 71
5.2.2 Observing Detection Results 73
5.3 Comparison with Search Engines 75

6 Conclusions and Future Work **81**
6.1 Conclusions 81
6.2 Future work 82

List of Figures

3.1 Use Case Diagram 19
3.2 "Three servers approach" scheme 23
3.3 MongoDB Collections (tables) 24

4.1 Registration Flowchart 39
4.2 Authorization Flowchart 42
4.3 Account Updating Flowchart 44
4.4 Account Deletion Flowchart 45
4.5 Datatypes Creation, Updating and Deletion Flowchart 46
4.6 Collections Creation, Updating and Deletion Flowchart 49
4.7 Crawl Preparations Flowchart 51
4.8 Crawling Cycle Flowchart 52
4.9 Update Crawldb Flowchart 54
4.10 Critical Data Detection Flowchart 55
4.11 Observing Detection Results Flowchart 57

5.1 User Creation Screenshot 62
5.2 User Creation Result Screenshot 62
5.3 Screenshot of user123 in the MongoDB 63
5.4 Login Screenshot 63
5.5 Login Result Screenshot 64
5.6 Cookies Screenshot 64
5.7 Crawl Setup Screenshot 65

5.8 Crawl Invocation Result Screenshot 65
5.9 Nutch Terminal Screenshot 66
5.10 Received Email Screenshot 67
5.11 Crawling rounds chart 67
5.12 Datatype Creation Screenshot 68
5.13 Datatype Creation Result Screenshot 68
5.14 Datatype in the DB Screenshot 69
5.15 Collection Creation Screenshot 70
5.16 Collection Creation Result Screenshot 70
5.17 Collection in the DB Screenshot 71
5.18 Collection Use Screenshot 72
5.19 Collection Use Result Screenshot 72
5.20 Nutch Indexing Log Screenshot 73
5.21 portuguese_phones Datatype Detection Results Screenshot 74
5.22 "portuguese_time_periods" Datatype Detection Results Screenshot 74
5.23 Google Search Results 76
5.24 Yandex Search Results 76
5.25 Indexed pages and pages with detected Portuguese phone numbers 77
5.26 "emails" Datatype Detection Screenshot 78
5.27 Google "emails" Datatype Detection Screenshot 79
5.28 Yandex "emails" Datatype Detection Screenshot 79
5.29 Indexed pages and pages with detected email addresses 80

Chapter 1

Introduction

Protection against critical data leaks was always one of the main aims of information technologies of all times, the approaches for achieving this aim are constantly evolving. Since ancient times up to modern ones, where the simplified access to information and improved ways of its retrieving lead to the fact that the level of visibility of the relationship of various information is so great that even a small part of it can be used extremely effectively, unfortunately, regardless of the purpose.

The classification of critical information is already an almost separated science field, but we can state that there is one main factor basing on which the information could be considered critical or not - the amount of new information that could be obtained or revealed during the processing of the initial information. Depending on the amount of new, revealed information, sometimes, data could be classified as critical, and since modern communication technologies and ways of information exchange are developed greatly, this leads to necessity to protect even the most basic types of information especially if the data in on a particular institution's public Web site, where, usually, lots of data uploaded daily.

Obviously, before preventing important information from being leaked, organizations must be able to identify and detect such critical types of information. Data security algorithms are existing and constantly improving, but algorithms for detecting and identifying

different types of critical information are often delegated to the employees of security departments of organizations, which leads to other set of risks that are associated with the human factor. Organizations can try to avoid risks of critical data leaks by using balanced task delegation between human mind capabilities and trustworthiness of different software algorithms. But if the identification of certain types of critical data is difficult both to actually and legally delegate to software algorithms, the detection of already identified types of critical data is often automated.

1.1 Goal

The main goal of this work is to create an automated, multiuser, critical data detection system which will let users to collect the public data from institutions' Web sites, define sets of descriptive entities for data to be detected and detect critical data, that were not planned to be publicly available on those Web sites, whose data was collected, by applying sets of descriptive entities to the collected data, or to get some certain guarantee of absense of such critical data.

The main advantages of this system are expected to be:

1. the process of collecting the data to be scanned for critical data leaks from the Web by the system itself, without relying on indexes of search engines;
2. the process of detection of multiple critical types of data simultaneously, as well as the ability of using the system by multiple users simultaneously;
3. the fact that it will be, obviously, free to use;

1.2 Methodology

For creating the system, we will search for already existing tools and programs for collecting all data from the institution's Web sites, including both documents with different formats and just plain text. We are also going to search for already existing tools suitable

for storing and processing large amounts of text data. We plan to integrate the tools, that will be found, as a part of our system's functionality, so the users would be able to use these tools without interacting with them directly.

Then, we are going to adapt the system for simultaneous use by several users and make its computational resources available for future scaling which will make the system more efficient.

As a final step, we plan to check if the system is working as expected and which improvements can be scheduled for future works related with the system.

1.3 Document Structure

The first chapter introduces the reader to an overview of the current situation about the problem of data leakage, which motivated us to do this work.

The second chapter adds more detailed description of the problem by adding more problem-related definitions and existing examples of its solving, presenting an overview of a set of technologies that were created for solving of the problem-related tasks, highlighting and justifying their usage in case if some of those technologies were chosen for integration with our system.

The third chapter breaks the problem down to tasks to be solved and automated in order to create the system. In such way the chapter is justifying and presenting the elaborated approach with the definition of the functional roles of the technologies chosen in the previous chapter, which will be used during actual system implementation.

The forth chapter describes the actual implementation and deployment of the different system's components according to the approach described previously.

The fifth chapter describes several tests that were performed in order to assure system's functions are working as expected and demonstrates these tests' results.

The sixth chapter draws conclusions of the done work and presenting a number of improvement suggestions that most likely will positively affect the system and will make it more effective.

Chapter 2

Context and Technologies

Data leakage is an uncontrolled or unauthorized transmission of classified information to the outside [1].

Data leaks involve the release of sensitive information to an untrusted third party, intentionally or otherwise. Organizations increasingly may be harmed by data being revealed to unauthorized parties [2].

Data leak incidents range from, for example, uploading a confidential document on a public Web site instead of placing it on a secured file server, to printing a sensitive document on a widely accessible printer and then forgetting that the document was printed [3].

For solving the data leaking problem there are existing two closely related types of systems:

1. Data Leak Detection system (DLD) - this type of system will not provide absolute protection, but it is essential for identifying data leakage as soon as possible [1]. An example of the DLD system could be Hivecode Data Leak Detection service, which is offering user's database tracking. For tracking users' databases, Hivecode will need to insert its own records with generated emails and phone numbers. All the emails seeded in users' databases will be regularly checked by Hivecode in the case of detection of inappropriate emails. The emails seeded in the database are not being

used, so as soon as one unauthorized email gets, it becomes suspicious, and the Hivecode's algorithms will analyse it and once the system detects multiple emails with similar content on different seeded emails from a particular user's database, Hivecode will notify the user about probable data leak by using user's messengers [4];

2. Data Leak Prevention system (DLP) - this is a type of system which combines the DLD system and additional functionality that will perform some actions on the data, found previously by DLD system, in order to prevent futher disclosure of that data. An example of the DLP system could be Google Cloud Data Loss Prevention service, which is offering data detection functionality with usage of their built-in 120 Infotypes (entities that are describing information to detect) as well as tools for classifying, masking, tokenizing, and transforming data [5].

One of the purposes of this work is to contribute to further development of Data Leak Detection systems.

There were already done a number of works that are contributing to Data Leak Detection algorithms: works that aimed at detection of information leaks in business process models [6]; works that contributed to the Security of mobile applications by proposing approaches to execute static analysis tools in order to detect probable data leaks [7].

We are going to implement a multiuser Data Leak Detection System aimed, but not limited to capturing critical data on institution's Web sites, basing on descriptive data entities received from users. To create such system and use it for detecting the critical data leaks we will need the following components for the system:

1. a crawler, so we could crawl the Web sites' data and be able to analyze it in a safer, local environment;

2. a search platform for performing data indexing and detection processes;

3. and in order to, at least, partially automate the overall process, we will build a multiuser application with a database which would be able to communicate with a

crawler, a search platform and users.

Further sections are describing those components and detailing their roles in the system.

2.1 Node.js and Node Package Manager

Node.js is an asynchronous event-driven JavaScript runtime, that is designed to build scalable network applications. Upon each Node.js connection, the callback is fired, but if there is no work to be done, Node.js will wait for it. This is in contrast to today's more common concurrency model, in which OS threads are employed. Thread-based networking is relatively inefficient and very difficult to use. Furthermore, users of Node.js are free from problems of dead-locking the process, since there are no locks. Almost no function in Node.js directly performs input or output operations, so the process never blocks. Because nothing blocks, scalable systems are very reasonable to develop in Node.js.

Node.js is similar in design to, and influenced by, systems like Ruby's Event Machine and Python's Twisted. Node.js presents an event loop as a runtime construct instead of as a library. In other systems, there is always a blocking call to start the event-loop. Typically, such behavior is defined through callbacks at the beginning of a script, and at the end a server is started through a blocking call. In Node.js, there is no such call for starting the event loop. Node.js simply enters the event loop after executing the input script. Node.js exits the event loop when there are no more callbacks to perform. This behavior is like browser JavaScript — the event loop is hidden from the user [8].

Speaking about Node.js it is worth to mention the Node Package Manager (NPM) - the world's largest software registry. It gives Node.js developers a possibility to expand the initial set of functions of Node.js in a short time, by downloading the necessary packages from the registry [9].

In our system, Node.js will be the core development tool. We will use it for development of the main server which will receive user inputs and basing on them - delegate tasks

to other functional parts of the system: to the database, to the crawler, to the search engine and process their responses along with generation of responses for users.

2.2 MongoDB

MongoDB is a document database which stores data in flexible, JSON-like documents, meaning that fields can be different from document to document and data structure can be changed over time.

MongoDB is a distributed database at its core, so high availability, horizontal scaling, and geographic distribution are built in and ready to use.

MongoDB is free to use. Versions released prior to October 16, 2018 are published under the AGPL. All versions released after October 16, 2018, including patch fixes for prior versions, are published under the Server Side Public License (SSPL) v1 [10].

In our system, MongoDB will be the core database. We will use it for storing all necessary information like users' accounts, users' data associated with users' accounts. Basically all data that will be used by system, except the data which will be collected from Web sites, will be stored in MongoDB.

2.3 Crawlers

Web crawlers, or spiders, are programs that automatically browse and download web pages by following hyperlinks in a methodical and automated manner. Various types of web crawlers exist. Universal crawlers are intended to crawl and index all web pages, regardless of their content. Others, called preferential crawlers, are more targeted towards a specific focus or topic. Web crawlers are known primarily for supporting the actions of search engines, particularly in web indexing. However, web crawlers are also used in other applications that are intended to collect and mine online data, such as web page content mining applications [11].

The important part that should be underlined is that our system will not try to get

the data available only for authorized websites' users, because roughly speaking, types of the data behind the authorization forms mostly depends on the type of authorizing account and these are questions of the university's structure, their site and their security systems.

In this chapter a set of web crawlers will be mentioned and described in order to justify the usage of one of them.

2.3.1 Scrapy

Scrapy is an application framework for writing web spiders that crawl web sites and extract data from them. It is written in Python and since it is a framework – it can be included in any Python environment, also it provides a built-in mechanism for extracting data (called selectors) but it is possible to use other Python parsing libraries instead. Scrapy has healthy community with 31k stars, 7.5k forks and 1.8k watchers on GitHub and 11k questions on StackOverflow which makes it one of the most popular tool among its kind [12].

2.3.2 Scrapestorm

ScrapeStorm is an AI-Powered visual web scraping tool, which can be used to extract data from almost any websites without writing any code. It is a desktop application available for Windows, Mac, and Linux users. User only needs to enter the URLs, ScrapeStorm can intelligently identify the content and next page button, reduced amount of complicated configurations required from user.

ScrapeStorm built by ex-Google crawler team. Has a structured documentation system with video tutorials and examples. Constant monthly updates (usually even twice per month) [13].

2.3.3 Octoparse

Octoparse is a visual web data extraction software. Designed to bulk extract information from websites, for most of scraping tasks no coding needed. This software designed for MS Windows OS only (currently available on win7/8/10).

Octoparse has two modes in the Operation panel: the Wizard mode and the Advanced mode. In the Operation panel, user can choose one of these two modes to start an extraction task. Wizard mode is more suitable for beginners. User can grab data from simple web pages by just following the instructions step by step to configure his own task. Advanced mode is most commonly used. With advanced mode, user can deal with any complex page structures by more powerful features like scheduling feature and cloud servers.

Octoparse has the documentation and community, but mostly relies on its Support teams and email support which have different respond time and quality depending on the type of user's subscription [14].

2.3.4 80legs

80legs is a web crawling service that allows its users to create and run web crawls through its software as a service platform. Built on top of a distributed grid computing network. The grid consists of approximately 50,000 individual computers, distributed across the world, and uses bandwidth monitoring technology to prevent bandwidth cap overages. So, this service is platform – independent.

For more customization options – user must use 80legs javascript-based app framework to fully customize behavior for the 80legs web crawling service. Also important details are: the service has RESTful API which is giving user a possibility to automate web crawl creation, result retrieval, etc.; 80legs distributes request to web pages across a large collection of servers with different IP addresses, which led to the fact that 80legs has been criticised by numerous site owners for its technology effectively acting as a Distributed Denial of Service attack and some rulesets for modsecurity block all access to the webserver

from 80legs in order to prevent a DDOS (blocking by user-agent).

80legs has the documentation and community, but mostly relies on its Support teams and email support which have different respond time and quality depending on the type of user's subscription [15].

2.3.5 PySpider

PySpider is a web crawler system that can run python scripts. It is written in Python and can be launched in any Python environment, it also has graphical interface where user can edit scripts, monitor ongoing tasks and view results. PySpider supports AJAX heavy websites, but partly relies on PhantomJS, which development is currently suspended. Also, unlike Scrapy, where user needs to install scrapy-splash to render the javascript website, Pyspider provides Puppeteer, which is a library developed by Google in Javascript for web crawling.

Licensed under the Apache 2 license, Pyspyder is still being actively developed on GitHub and has documentation with sample code examples [16].

2.3.6 Apify Web Scraper

Apify is the place to find, develop, order and run cloud programs called actors. Actors are using for scraping web pages, processing data or integrating with Web applications. Web Scraper is a part of Apify's system, designed for crawling arbitrary websites using the Chrome browser and extracts data from pages using a provided JavaScript code. The actor supports both recursive crawling and lists of URLs and automatically manages concurrency for maximum performance. It is Apify's basic tool for web crawling and scraping.

Apify, itself, is a web scraping and automation platform that can extract structured data from any website or automate any workflow on the Web. For using its services, including Web Scraper – user must have an Apify account. For even more flexibility and control, user must develop a new actor from scratch in Node.js using Apify SDK.

Apify Web Scraper has the documentation (mostly just examples and starting guides) and community (but mostly businessmen, not developers), which is mostly relies on its Support teams and email support which have different respond time and quality depending on the type of user's subscription [17].

2.3.7 Apache Nutch

Apache Nutch is a highly extensible and scalable open source web crawler software project. Stemming from Apache Lucene, the project has diversified and now comprises two codebases, namely:

1. Nutch 1.x: A well matured, production ready crawler. 1.x enables fine grained configuration, relying on Apache Hadoop data structures, which are great for batch processing;

2. Nutch 2.x: An emerging alternative taking direct inspiration from 1.x, but which differs in one key area; storage is abstracted away from any specific underlying data store by using Apache Gora for handling object to persistent mappings. This means we can implement an extremely flexibile model/stack for storing everything (fetch time, status, content, parsed text, outlinks, inlinks, etc.) into a number of NoSQL storage solutions.

Nutch provides extensible interfaces such as Parse, Index and ScoringFilter's for custom implementations, e.g., Apache Tika for parsing. Additionally, pluggable indexing exists for Apache Solr, Elastic Search and some other indexers. Also both versions of Nutch can run on a single machine, but also able to run on top of Hadoop cluster. Nutch crawler is a project of the Apache Software Foundation and is a part of the larger Apache community of developers and users [18].

2.3.8 Crawler choice argumentation

We will use Nutch 1.x because it is free, production-ready crawler with large number of users, ability to run on top of Hadoop cluster and large support base. Unlike other crawling solutions, listed above, Nutch is:

1. Free to use (according to version 2.0 of the Apache License);
2. Both Nutch versions have a RESTful API while not all other crawlers have it, although some are having their own SDK;
3. Able to crawl not only Web sites' HTML content, but the entire Web sites' content along with text documents placed on them and do not require additional packages or configurations for that;
4. Those crawlers that have a RESTful API too, like, for example 80legs crawler, are distributing crawling requests across their own collection of servers with different IP addresses, while Nutch is infrastructure-independent, but not limited to a single machine's computational resources because it is relying on Hadoop and therefore could be placed on top of large Hadoop cluster, and Hadoop is known for its scalability;
5. And what is the most important - by running on top of Hadoop cluster, Nutch is using Google MapReduce algorithm for processing huge amounts of data in parallel, which is giving Nutch the ability to perform several crawling processes with different parameters simultaneously [19]. That means that only using Nutch we can create a multiuser system, because other crawlers are mostly already multiuser systems by themselves, which are offering crawling solutions as a personal service. Others are standalone crawling applications or frameworks which were designed for personal use and in order to be used by multiple users at the same time, they would require a additional efforts for achieving that.

Nutch is a project of the Apache Software Foundation, so it is also a part of the larger Apache community of developers and users. Last Nutch versions (1.16 from 1.x branch and 2.4 from 2.x branch) were released at the 11 of October, 2019.

We prefer 1.16 version over 2.4, because Nutch Developers, in their release announcement for Nutch 2.4, noted that they expect v2.4 to be the last release on the 2.X series and they have decided to freeze the development on the 2.X branch, as no committer is actively working on it [20].

In our system, Nutch will be the part of the system responsible for crawling processes and for storing crawled Web sites in Nutch-exclusive crawl-databases.

2.4 Full-Text Search Engines

Full-text searching is the type of search a computer performs when it matches terms in a search query with terms in individual documents in a database and ranks the results algorithmically. This type of searching is ubiquitous on the Internet and includes the type of natural language search we typically find in commercial search engines, Web site search boxes, and in many proprietary databases. The term full-text searching has several synonyms and variations, including keyword searching, algorithmic searching, stochastic searching, and probabilistic searching [21].

In this chapter, two, most commonly used with Nutch, Full-Text Search Engines will be mentioned and described in order to justify the usage of one of them.

2.4.1 Apache Solr

Solr is an open-source enterprise-search platform, written in Java, from the Apache Lucene project. Its major features include full-text search, hit highlighting, faceted search, real-time indexing, dynamic clustering, database integration, NoSQL features and rich document (e.g., Word, PDF) handling. Providing distributed search and index replication, Solr is designed for scalability and fault tolerance. Solr is widely used for enterprise search and analytics use cases and has an active development community and regular releases.

Solr runs as an independent http-server. It utilizes the Lucene Java search library at its core for full-text indexing and search, and has REST-like HTTP/XML and JSON APIs that make it usable from most widespread programming languages. Solr's external

settings allow users to customize it without Java coding, and it has a module design to help even more advanced customization.

Apache Lucene and Apache Solr are both developed by the same Apache Software Foundation group [22].

2.4.2 ElasticSearch

Elasticsearch is a distributed, open source search and analytics engine for all types of data, including textual, numerical, geospatial, structured, and unstructured. Elasticsearch, as well as Solr, is built on Apache Lucene and was first released in 2010 by Elasticsearch N.V. (now known as Elastic). Known for its simple REST APIs, distributed nature, speed, and scalability, Elasticsearch is the central component of the Elastic Stack, a set of open source tools for data ingestion, enrichment, storage, analysis, and visualization. Commonly referred to as the ELK Stack (after Elasticsearch, Logstash, and Kibana), the Elastic Stack now includes a rich collection of lightweight shipping agents known as Beats for sending data to Elasticsearch [23].

2.4.3 Full-Text Search Engine choice argumentation

Both of described products based on Apache Lucene and both have rich documentation for integration with Nutch. But while Solr is a standalone software, ElasticSearch is a part of Elastic Stack. We have chosen Solr, because it is maintained by same Apache foundation, and because we don't need ElasticSearch's or Elastic Stack's advanced features, since we are going to implement most of them specifically for our aims.

In our system, we will use Solr 7.7.2, which will be the part of the system responsible for indexing data from Nutch crawl-databases into Solr cores and for performing actual detection of critical data.

Chapter 3

Analysis and Approach

This chapter describes the requirements and the approach that are required for successful implementation of the system and also describes the tasks that the system should implement.

3.1 Analysis

To create the multiuser system for detecting the critical data leaks we will need to provide users with the following abilities in the system (Figure 3.1):

1. to crawl the Web sites' data and to delete already crawled data, where under "crawling" we mean the process of navigating through all available Web pages of a Web site with the simultaneous extraction of all text data including documents containing on the Web pages or provided through links located on the Web pages;

2. to create, edit and delete datatypes to be detected in the crawled data, where under "datatype" we mean the entity that will be using for detection of certain types of critical data among crawled Web sites' data. Systems which are able to detect critical data inside another data (DLD systems) - usually, except having a functionality which allow them to collect and access the data, also having a set of descriptive entities that are using for data identifying processes and since we are

concentrated more on data detection processes than on data identifying ones, we decided to implement functionality that will allow users to create and manage their own data descriptive entities which the system will be able to use during processes of critical data detection inside crawled Web sites' data. Each such entity, or as we named it - "datatype" will contain the name and a standard regular expression pattern for matching the desirable type of critical data;

3. to create, edit and delete collections (sets) of datatypes, for detecting multiple datatypes simultaneously, where under "collection" we mean the entity that contains a set of datatypes, organized as a JSON object, where the keys are datatypes' names and the values are their regular expressions, which a particular user had chosen to be included in a collection, a name which will be used for collection identification purposes and an identifier of the owner - the user that created the collection and have an access to it;

4. to detect collections of datatypes inside the crawled data, where under "detection" we mean the process of extraction regular expressions and their names from datatypes entities placed in a particular collection, and performing process of matching regular expressions with user's crawled data and in case of match - to mark the matched piece of data with the name of datatype, that was used for matching;

5. and, at last, to create, edit and delete accounts in the system for separating resources of different users.

3.2 Approach

In accordance with the requirements defined previously in the use case diagram, for building the system, we will implement a set of functions which will allow users to:

1. Have an access to the set of initial datatypes for further data detection in collected Web sites' data.

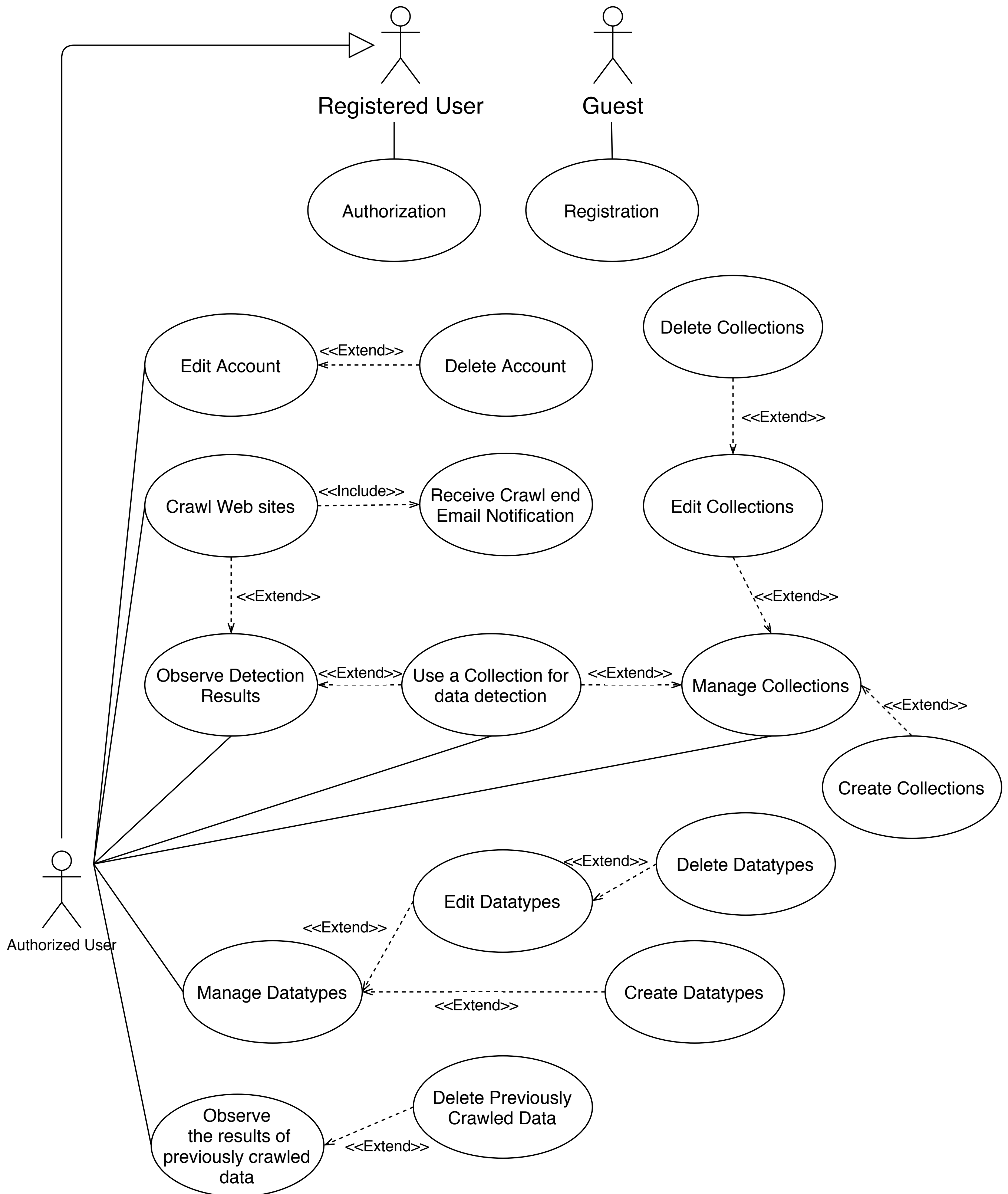

Figure 3.1: Use Case Diagram

We decided both to create an initial set of datatypes, so all the users would be able to start to detect or test the detection of critical data using them on their crawled data, and to give users a functionality that will allow them to create their own datatypes. While datatypes created by users will be available for further edition and usage only to their creators - the initial set of datatypes will be available for all users, but without an ability to edit it;

For accomplishing those tasks Node.js and MongoDB functionalities will be used;

2. Create, edit and delete their own, desirable datatypes by themselves;

   As we wrote previously, basically any data that reveals another data could be considered critical or even personal, in case if it is related with information about a real person. But even a less general "personal data" term is not always clearly enough identified in data regulation laws for being easily integrated to our datatype-based detection system;

   For example, in GDPR, "personal data" term means any information relating to an identified or identifiable natural person ("data subject"), an identifiable natural person is one who can be identified, directly or indirectly, in particular by reference to an identifier such as a name, an identification number, location data, an online identifier or to one or more factors specific to the physical, physiological, genetic, mental, economic, cultural or social identity of that natural person [24];

   For accomplishing the implementation tasks described above - Node.js and MongoDB functionalities will be used;

3. Create, edit and delete collections (sets) of datatypes using both their own and the initial datatypes for further critical data detection processes, so the users would be able to detect multiple datatypes at a time;

   Considering the fact that a particular user after the end of the crawling process, most likely, would like to detect multiple types of critical data in a dataset simultaneously, instead of only one at a time, we decided to implement functionality that will allow

users to create and manage their own collections of datatypes which they would be able to use during processes of their critical data detection inside crawled Web sites' data;

For accomplishing those tasks Node.js and MongoDB functionalities will be used;

4. Collect (crawl) data from desirable Web sites for further detection of critical datatypes inside collected data as well as to delete already collected data;

   Retrieving public data from institutions' Web sites is an essential task that must be achieved during the system implementation;

   For accomplishing those tasks Node.js and Apache Nutch functionalities will be used;

5. Receive notification messages on the end of data collecting process initiated previously;

   Considering the fact that the amount of time which process of crawling Web sites could require depends heavily on the number of Web sites, their amount of content and the computing resources of a particular machine - we came to conclusion that user will need some kind of notification on end of the crawling process;

   For accomplishing those tasks we will rely on the functionality of Node.js and in particular on "nodemailer" npm module;

6. Observe the crawldb stats upon the end of crawling processes;

   The Nutch crawl database, or crawldb - is a place where Nutch stores information about every crawled Web page, including whether it was fetched, and, if so, when.

   After the end of crawling process users must be not only notified about the actual end of the process, but, obviously, also must have an ability to observe the stats of the crawldb such as a number of fetched pages, unfetched pages etc. - for being able to decide if the data collected during crawling process is satisfying their needs and expectations or they wish to crawl more data or to re-create the entire crawldb;

7. Use their collections of datatypes for performing detection of multiple critical datatypes at a time inside their collected data from Web sites;

   The main function of the system that will be used for detection of all the datatypes, included in a particular, chosen by user, collection, inside the Web sites' data, crawled previously;

   For accomplishing those tasks Node.js, MongoDB and Solr functionalities will be used;

8. Observe the results of detection process;

   After the end of detection process users must have an ability to observe the detected pieces of data along with the url-addresses which are containing those data, so the users would have a certain amount of information for deciding if the data detected during detection process should stay in content of the respective url or not;

   For accomplishing those tasks Node.js, MongoDB and Solr functionalities will be used;

9. Register and authorize in the system and therefore restrict access to individual resources such as user's collected data, user's email address, user's crawled data etc;

   Registration and authorizing functions are an essential part of any multi-user system. We will use Node.js for defining logic of those functions and MongoDB for storing user's credentials;

   For user registration the user will be asked to input a name, a password and an email. Email will be used only for notification purposes, name will be a unique user identifier, and therefore must be required to be unique, password will low down the chances of illegal access to user's account. For providing essential safety measures - the passwords must be hashed before storing in MongoDB;

10. Edit and delete their previously registered accounts;

After finishing account creation process user must have the possibility to edit some of the account parameters as well as to freely delete it. Those functions are related to registration and authorizing functions and also are an essential part of any multi-user system;

For accomplishing those tasks Node.js and MongoDB functionalities will be used;

The approach to implementation (Figure 3.2) is to create a system that consists of three logical parts, where each part is represented by a separate server that is responsible for a certain part of the general functionality of the whole system.

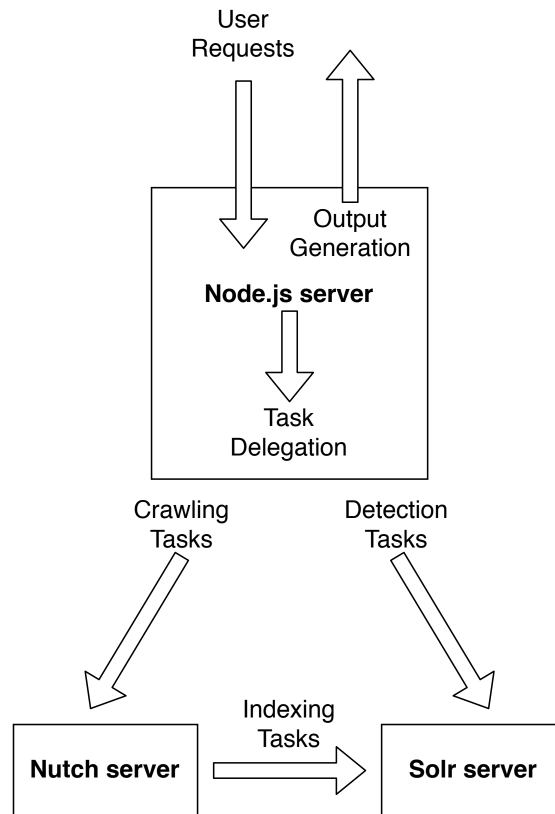

Figure 3.2: "Three servers approach" scheme

Those three servers would be Nutch and Solr servers that are logically integrated with Node.js master server.

Nutch Server will be responsible for crawling Web sites processes and storing crawled data.

Solr server will be responsible for critical data detection operations inside data crawled by Nutch.

Node.js server will be a master server - the only server with which the user will interact directly. It will have a functionality for accepting and processing inputs of multiple users and basing on the results of this processing - delegating tasks to Nutch and Solr servers.

Node.js server will also be responsible for retrieving and processing the results of operations delegated previously to Nutch and Solr servers and later returned by them, into user responses.

Obviously, that all the functionality that is not existing inside Nutch and Solr servers like, for example, user's registration and creation of collections or datatypes - must be implemented in Node.js server. And for some of those functions or for data created during work of those functions' work a implemented and working database is required. So, using the use case diagram as a reference - we have created a MongoDB diagram for the future system's database (Figure 3.3).

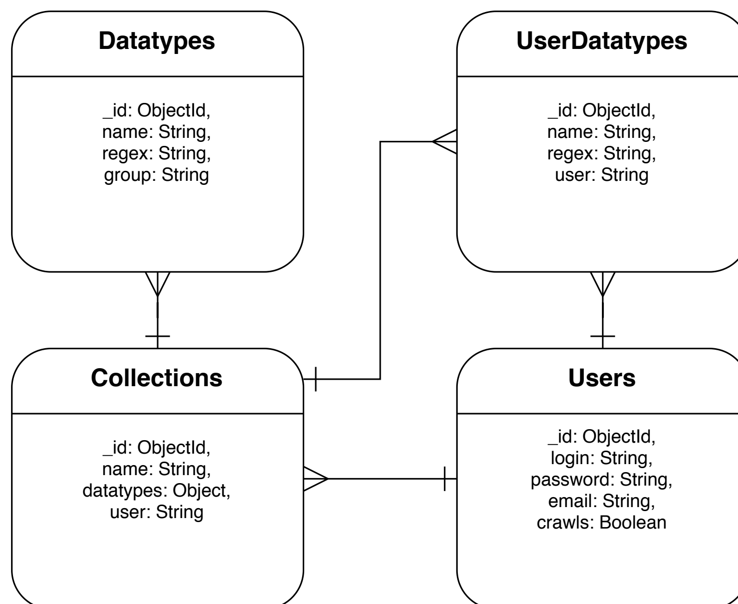

Figure 3.3: MongoDB Collections (tables)

Note that MongoDB is NoSql database, that is why we created the non-standard tables' diagram and not the ER diagram.

Chapter 4

Components Deployment and System Implementation

Following the "three servers" approach described in the previous chapter, this chapter is divided into three parts where each part is corresponding to a certain server as to a part of the whole system.

4.1 Nutch server

4.1.1 Setup

Nutch requires Java Development Kit in order to run. In our case, we used OpenJDK java 8. After downloading Nutch 1.16 from the official Web page, for starting to use it either as an application or as a server we should customize some of the crawl properties first. Default crawl properties can be viewed and edited within `conf/nutch-default.xml` file - where most of these can be used without modification. The file `conf/nutch-site.xml` serves as a place to custom crawl properties that overwrite `conf/nutch-default.xml`. The modification that we need for launching crawl processes is to override the value field of the `http.agent.name` to our desirable value:

```
<property>
```

```
 <name>http.agent.name</name>
 <value>Spider_Student</value>
</property>
```

Also we will create a `apache-nutch-1.16/users_files` folder for storing user-related files.

When modifications are done - we are ready to start Nutch server. For this we should navigate to Nutch core folder and invoke `bin/nutch startserver` command. This will start the server on port number 8081.

4.1.2 Nutch Data Concepts

Nutch data is composed of:

The crawl database, or crawldb. This contains information about every url known to Nutch, including whether it was fetched, and, if so, when.

The link database, or linkdb. This contains the list of known links to each url, including both the source url and anchor text of the link.

A set of segments. Each segment is a set of urls that are fetched as a unit. Segments are directories with the following sub-directories:

- a crawl_generate subdirectory names a set of urls to be fetched;
- a crawl_fetch subdirectory contains the status of fetching each url;
- a content subdirectory contains the raw content retrieved from each url;
- a parse_text subdirectory contains the parsed text of each url;
- a parse_data subdirectory contains outlinks and metadata parsed from each url;
- a crawl_parse subdirectory contains the outlink urls, used to update the crawldb [25].

4.1.3 Nutch Crawl Process

As we wrote before - we will use Nutch for crawling Web sites. Nutch crawl process is actually a set of sub-processes manually or algorithmically invoked one by one. Those sub-processes or crawling steps are: injecting, generating, fetching, parsing, updating (crawldb), inverting (links) and indexing (to Solr). Each of those steps can be placed to Nutch server as a job by sending a POST request to `host:8081/job/create` path with the following JSON body:

```
{
  "crawlId":crawlId,
  "type":"TYPE",
  "confId":confId",
  "args":{}
}
```

In that body the value of "crawlId" key is a unique name of crawldb to which the job will be applied (e.g. "crawlId":"myCrawl"); The "type" is an uppercased name of the job that will be invoked (e.g. type:"INJECT"); The "confId" is a name of the configuration which will be using as a source of different parameters that will be applied to the job during its execution (e.g. "confId":"default"); The "args" is a JSON that could contain all other optional parameters that are not existing or couldn't be placed inside the configuration. Detailed information about Nutch configurations can be found in the "Nutch Configuration" subsection.

A brief description of each Nutch job type is given below.

Inject job. After the request will be received by Nutch and the injection operation will be invoked, the inject class will take a flat file of user's previously created urls list, the path to which is provided in the configuration received with request, and adds those urls to the list of Web sites to be crawled. Injecting step is useful for bootstrapping the crawl process, it is often, actually, an initial part of it, because if there were no crawldb before - it will be created after injecting step. The url file is a plain text file that must

contain one url per line, optionally followed by custom metadata separated by tabs with the metadata key separated from the corresponding value by '=' symbol [26].

Generate job. After the request is received by Nutch and the generation operation is invoked, the generate class will create a subset of a crawldb to fetch and will put it in a segment folder which will be created as well. The user's regex-urlfilter file will be used during generation process as a set of restrictions of adding urls to the crawldb subset [27].

Fetch job. After the request is received by Nutch and the fetch operation is invoked, the fetch class will crawl the urls in previsouly generated segment [28].

Parse job. After the request is received by Nutch and the parse operation is invoked, the parse class will analyze and separate the data of the segment fetched previously into crawled content and crawled Web links inside the content [29].

Updatedb job. After the request is received by Nutch and the update operation is invoked, the update class will take the output of the fetched segment's Web links and update the crawldb accordingly [30].

Invertlinks job. After the request is received by Nutch and the invert links operation is invoked, the corresponding class will invert all of the links in the fetched segment, so that we may index anchor texts of all Web links on the parsed segment's content [31].

Index job. After the request is received by Nutch and the index operation is invoked, the index class will take the content from one or multiple segments and will pass it to all enabled IndexWriter plugins which send the documents to Solr, Elasticsearch, or to various other specified index back-ends [32].

4.1.4 Nutch Configuration

Nutch configuration is a JSON object that defines a set of parameters and paths to different files that could contain those parameters that will be applied to Nutch jobs during their invocations. Since we want to let multiple users to crawl Web sites by using Nutch simultaneously - some key configuration files must be unique for each user: the file with regular expression-based url filters and the file with Solr index writer with url

path to user's Solr core. Those files must be created directly in file system, but the paths to them must be provided via configuration. Also the file containing list of urls to be injected into crawldb during Nutch Inject job must be unique for each user too, but paths to those files provided as a separate parameter during only Nutch Inject job and therefore are separated from the Nutch configurations.

For creating a new configuration a POST request to `host:8081/config/create` path should be sent with the following JSON body:

```
{
  "configId":configId,
  "params":{}
}
```

In that body the value of `configId` key is a unique name for the configuration to be created and the value of `params` is a JSON object that contains parameters that must be changed in new configuration, those parameters that are not mentioned in the JSON - will be set to the same values as such parameters have in the default configuration.

For forcing Nutch to perform multiple different crawl processes with different parameters, which are expected to be dynamically received from user's input - we must create three files in the file system for each user as well as to create a Nutch configuration for each user providing paths to some of those files in the configuration. Those files are:

1. `seed.txt` file - for keeping the list of urls to be injected during Nutch Inject job. The file will be placed inside `apache-nutch-1.16/users_files/user/` folder, where `"user"` must be replaced with the user's unique identifier, the file will be passed as a separate configuration parameter during Nutch Inject jobs, and although it is not going to be included into Nutch configurations - it is still a configuration file by itself;

2. `regex-urlfilter.txt` file - for keeping the list of regular expression-based url filters which will be used during Nutch Generate job. The file will be placed inside `apache-nutch-1.16/conf/` folder;

3. `index-writers.xml` file - for keeping the list of Nutch index writers which will be used during Nutch Index job. In our case there will be only Solr index writer which url will point to user's Solr core. Example of an index writer (considering that the user's core named 'myCore'):

```
<writer id="indexer_solr_1"
 class="org.apache.nutch.indexwriter.solr.SolrIndexWriter">
  <parameters>
    <param name="type" value="http"/>
    <param name="url"
     value="http://localhost:8983/solr/myCore/">
    <param name="collection" value=""/>
    <param name="weight.field" value=""/>
    <param name="commitSize" value="1000"/>
  </parameters>
</writer>
```

   The file will be placed inside `apache-nutch-1.16/conf/` folder. Detailed information about Solr and Solr cores can be found in the "Solr server" section.

4. `regex_urlfilter.txt` and `index-writers.xml` files are expected to be in same folder and in order to be properly identified must be renamed, accordingly, to `user_regex_urlfilter.txt` and `user_index-writers.xml`, where `"user"` must be replaced with the user's unique identifier.

Considering the written above - the final version of the POST request body for configuration creation is expected to be:

```
{
  "configId":"conf_of_user",
  "params":{
    "urlfilter.regex.file":"user_regex-urlfilter.txt",
```

```
    "indexer.indexwriters.file":"user_index-writers.xml"
  }
}
```

In the request body a "user" substring must be replaced with the unique identifier of the user which is creating the configuration.

4.1.5 Nutch Readdb Requests

Readdb is an alias for Nutch CrawlDbReader class. The CrawlDbReader provides us with a read utility for the crawldb.

We will need CrawlDbReader functionality for obtaining and writing to the file the crawldb stats after the end of each user's crawling process. For placing Nutch Readdb job and obtaining crawldb stats, a POST request with the following body must be sent to `host:8081/db/crawldb` path:

```
{
  "type":"stats",
  "confId":"conf_of_user",
  "crawlId":"crawl_of_user",
  "args":{}
}
```

In the actual request bodies it is expected that all "user" substrings will be replaced with actual user's `userId` value.

After obtaining the response with the data about the stats of the crawldb we will create a file with the name of `userId` in the Nutch `apache-nutch-1.16/users_files/` folder and will write the data to that file.

We will use the file for providing and displaying, on user request, information about several crawldb parameters such as number of fetched urls, number of unfetched urls etc.

The processes of creation, writing to the file as well as reading it - will be performed by Node.js file system modules, specially designed for those purposes.

4.2 Solr server

Solr is an open-source enterprise-search platform. It runs as an independent http-server and will be used in our system for detection of the particular user's datatypes inside the user's crawled data.

4.2.1 Setup

We have downloaded and are using Solr 7.7.2. For starting to use Solr server we don't need to perform additional configurations. We need to navigate to Solr core folder and invoke `bin/solr startserver` command. This will start the server on port number 8983.

4.2.2 Functionality To Be Used

Out of great number of Solr's functional possibilities we need just a few: create a Solr core as well as to clear, reload or delete it, apply user's collection of datatypes as a set of Solr's CharFilters and index the user's crawled content.

In Solr, the term "core" is used to refer to a single index and associated transaction log with the configuration files. For creating a core, first, we need to create a configuration for the core by copying the `solr-7.7.2/server/solr/configsets/_default/` folder as a template and after applying desirable changes - renaming and saving it in the `solr-7.7.2/server/solr/configsets/` directory, and then to send a GET request to Solr `host:8983/solr/admin/cores?action=CREATE` path along with the "name" and the "configSet" query parameters, which are the name of the core that will be created and the configuration set for it. More details about core configSets can be found in the next subsection.

Other core related operations that will be used in our system are core reload, clear and delete operations. Core reloading is an operation that is required to perform for submitting changes made in the core's configset after the core was started. Core clearing is an operation of unlinking data contained in the core's index, deleting - an operation of unloading the core from the Solr.

For core deletion it is necessary to send a GET request to Solr `host:8983/solr/admin/cores?action=UNLOAD` path along with the "core" query parameter, which is the name of the core to be deleted. For clearing core's index it is necessary to send a POST request to Solr `host:8983/solr/update?&commit=true&overwrite=true` path with the following XML body:

```
<delete>
    <query>*:*</query>
</delete>
```

For core reloading it is necessary to send a GET request to `host:8983/solr/admin/cores?action=RELOAD` path along with the "core" query parameter, which is the name of the core to be reloaded.

4.2.3 Indexing and Data Detection

Processes of indexing crawled data into Solr cores will be performed by Nutch Index jobs, the processes of detecting collections of datatypes will be performed by Solr and these processes are described in this subsection.

Each crawled Web page's content will be converted to a Solr document during Nutch index job. Each Solr core, among other settings, has schema.xml configuration file which is defining a way each indexed document's field will be processed and stored in Solr. For example, in default core configset, field 'content' has the following configuration:

```
<field name="content" type="text_general"
indexed="true" stored="true"/>
```

From this configuration we can understand that if field with name "content" will be encountered during indexing - it will be treated as a "text_general" type field, and after that will be indexed and stored.

Schema.xml also contains a list of possible fields' types elements. Each 'fieldType' element configured inside schema.xml is defining a name for a certain sequence of Solr processing components that will be applied to all fields that have such name assigned as

a value to their 'type' parameter (e.g. "type"="text_general"). For example, in default core configset, fieldType "text_general" has the following configuration:

```
<fieldType name="text_general" class="solr.TextField"
positionIncrementGap="100">
    <analyzer type="index">
      <tokenizer class="solr.StandardTokenizerFactory"/>
      <filter class="solr.StopFilterFactory"
      words="stopwords.txt"
      ignoreCase="true"/>
      <filter class="solr.LowerCaseFilterFactory"/>
    </analyzer>
</fieldType>
```

From this configuration we can understand, that those fields in which 'type' parameter equals "text_general", will be processed by the corresponding Solr analyzer that is containing a chained set of Solr Tokenizer component and Solr filter components placed after Tokenizer.

Solr Tokenizer is a component responsible for breaking field data into lexical units, or tokens, where each token is, usually, a sub-sequence of the characters in the text.

Indexing to Solr, when it is necessary, will be performed by Nutch indexing job and will be processed with schema.xml configuration parameters described previously. After the data from crawled Web sites is indexed into Solr core - we will need to take user's collection of datatypes and search for those datatypes inside the core. But we managed to achieve simultaneous indexing and detecting user's datatypes by using the Solr CharFilters.

CharFilters are Solr components that pre-processes input characters. They can be chained and placed in front of a Tokenizer. CharFilters can add, change, or remove characters while preserving the original character offsets to support features like highlighting. A CharFilter has a `pattern` parameter which contains a regular expression that will be used as a detection filter for incoming data and a `replacement` parameter which contains

a string for replacement operation. For example we created a CharFilter which will filter emails by using a regular expression and in case of finding an email - will replace it with a "datatype_emails" string:

```
<charFilter class="solr.PatternReplaceCharFilterFactory"
 pattern="(\w+@\w+\.\w+(.\w+)?)"
 replacement="datatype_emails"/>
```

So, CharFilter usage is solving multiple potential problems simultaneously. First, CharFilters are Tokenizer-independent, because they are applied before them and are applied to a raw data stream, which means that, for example, if a Tokenizer is creating tokens based on whitespaces between words and there is a CharFilter with a regular expression which includes a whitespace - that regular expression will be applied successfully, because it will be applied to the data before the whitespace-based Tokenizer which, after the end of CharFilters' job will split all data into tokens, removing all the whitespaces. Second, we can use standard-syntaxed regular expression instead of lucene-syntaxed regular expressions which are applied during regular Solr search queries. And third, as it was already mentioned before, CharFilters can be chained, so we can apply as many CharFilters as there are datatypes in user's collection, and they are preserving original character offsets, so they will not actually replace the matched string, they will only mark it with a value of 'replacement' parameter. Also important things are that by using CharFilters we are detecting desirable datatypes simultaneously with the indexing process without separating the detection from indexing and as it was explained before - CharFilters will replace the matched substring, but will keep the reference to original string's offsets. After content will be indexed - we will be able to perform usual Solr search queries for displaying desirable datatypes by passing their replacements as search tokens inside the query.

4.3 Master Server (Node.js server)

This section is divided into 13 subsections which are describing the implementation of the Node.js server and its functions and most of the subsections are logically referencing the

corresponding requirements from the use case diagram from Chapter 2.

4.3.1 MongoDB database

Creation of MongoDB database is not required to be separated from data insertion processes. If the database does not exist, MongoDB creates the database when a user will insert some data to the database for a first time. If a collection does not exist, MongoDB will create the collection when user will insert some data to the collection for a first time. So, we will start to implement all database-related operations during implementation of other functions where database operations required.

4.3.2 Server Initialization

We will place downloaded Nutch 1.16 and Solr 7.7.2 in one directory together with the Master server folder, inside Master server's folder, we created the following files:

1. "index.js" file, which will be the primary server file. It will be responsible for starting the server up and establishing server routes;

2. "dtb.js" file, which will contain and which will be responsible for establishing connections to the database and invocation of all database-related functions;

3. "nutch.js" file, which will contain all Nutch-related functions and which will be responsible for their invocations;

4. "solr.js" file, which will contain all Solr-related functions and which will be responsible for their invocations;

5. "handlers.js" file, which will contain all functions for processing requests to server's routes defined in "index.js" and which will be responsible for their invocations;

6. "helpers.js" file, which will contain all secondary functions that may be needed during execution of functions from "handlers.js", except database-related functions.

And the following folders:

1. "HTMLtemplates" folder, which will contain all the templates that will be needed during page rendering, including templates of header and footer of the pages;

2. "public" folder, which will contain all files related with page rendering besides html templates. For now, we will create only one file inside it - "app.css", which will contain most part of styling code for the pages to be rendered by using templates from "HTMLtemplates" folder.

4.3.3 Node.js NPM Modules

Node.js module is a set of functions that could be included in the application. Modules could be created by developers and used locally or could be created by third parties and uploaded into NPM registry.

During the implementation of the system, for simplifying the process, the following Node.js modules were loaded from NPM registry for further usage:

1. fs-extra module - adds file system methods that aren't included in the native File System module [33];

2. express - a web framework for Node.js [34];

3. body-parser - Node.js body parsing middleware, parsing incoming request bodies in a middleware before request handlers [35];

4. cookie-parser - for parsing HTTP request cookies [36];

5. express-session - for creating a session middleware with different options [37];

6. bcrypt - a library for simplifying password hashing processes [38];

7. nodemailer - a Node.js module for simpifying email sending processes [39];

8. mongodb - the official MongoDB driver for Node.js. Provides a high-level API on top of mongodb-core [40].

4.3.4 Node.js Core Modules

Node.js module is a set of functions that could be included in the application. Some of the modules are already included into the core of Node.js and do not needed to be downloaded in order to be used.

During the implementation of the system, for simplifying the process, the following of Node.js core modules were used:

1. HTTP module - allows Node.js to transfer data over the Hyper Text Transfer Protocol and to create an HTTP server that listens to server ports and gives a response back to the client [41];

2. File System module - provides an API for interacting with the file system in a manner closely modeled around standard POSIX functions [42].

4.3.5 User Registration

This subsection can be logically associated with the "Registration" use case from the use case diagram.

For creating an account, the user will need to fill the login, password and email. If during check in database no user with such login is found - user's password will be hashed, a Solr configSet for new user's Solr core under the name equal to user's login will be created, as well as the core itself, and, as a final step, a user's MongoDB document (record) will be created and added to the database. Account creation process also can be seen on the Registration Flowchart (Figure 4.1).

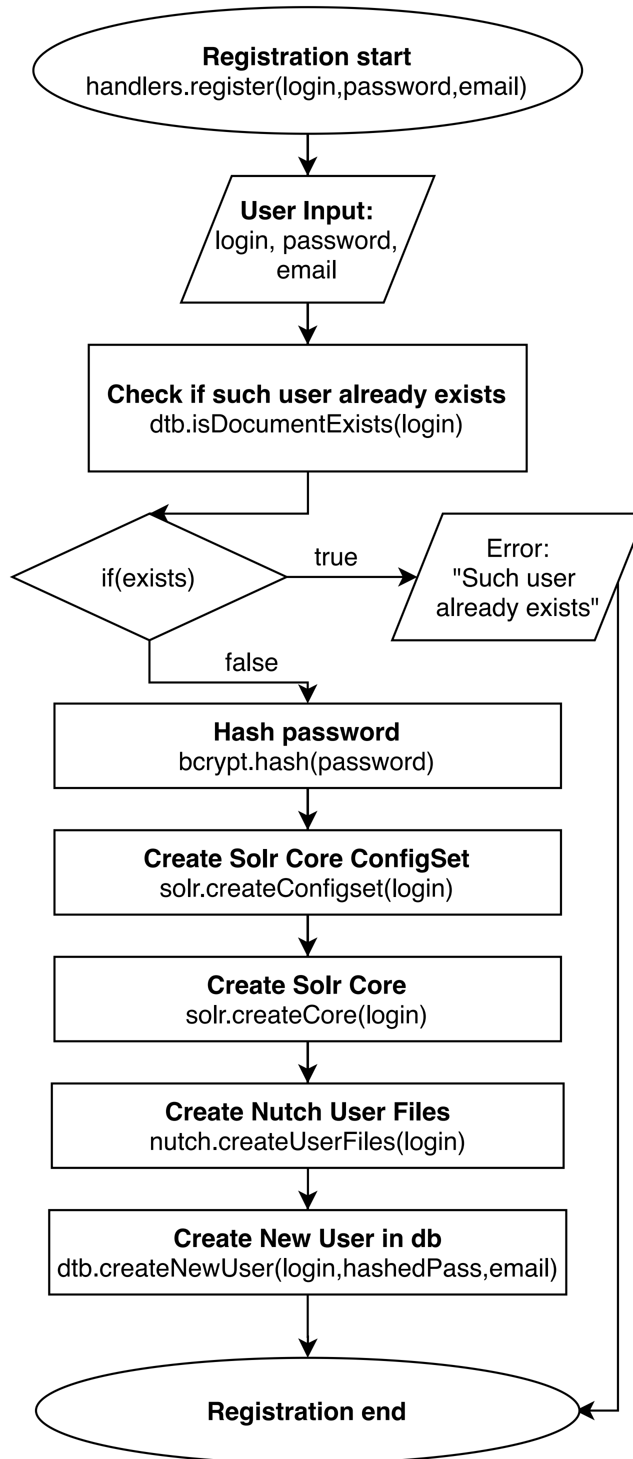

Figure 4.1: Registration Flowchart

During the creation of the Solr configSet the following Solr FieldType will be added

to Solr schema.xml:

```
<fieldType name="Detect" class="solr.TextField"
 positionIncrementGap="100">
    <analyzer type="index">
      <!--{{charFilters}}-->
      <tokenizer class="solr.StandardTokenizerFactory"/>
    </analyzer>
    <analyzer type="query">
      <tokenizer class="solr.StandardTokenizerFactory"/>
    </analyzer>
  </fieldType>
```

And the settings of Solr field named "content" in order to use our "Detect" FieldType will be changed to:

```
<field name="content" type="Detect"
indexed="true" stored="true"/>
```

We decided to create our own Solr "Detect" FieldType, because since we want the set of CharFilters to be applied only on the Solr field named "content" - it is better to create a FieldType specially for it and assign it as a value of `type` parameter of the desirable Solr field, instead of modifying already existing FieldTypes which could be used by other Solr fields as well.

Then `seed.txt`, `regex_urlfilter.txt` and `index-writers.xml` files will be created in Nutch directories and a Nutch configuration, based on those files will be created as well, and, at last, the user's login, hashed password and email will be put into the database as a new document in "Users" collection. File `seed.txt` will be placed inside newly created folder inside `users_files` folder which name will be equal to newly-created user's login. Since all user logins are expected to be unique - each user will have his unique folder inside `apache-nutch-1.16/users_files/` folder. Files `regex_urlfilter.txt` and `index-writers.xml` will be placed inside the `apache-nutch-1.16/conf` folder. All files

will be created but not filled, except `index-writers.xml` file, because it is expected to be a static file it will be filled immediately after creation with the Solr index writer, where "`coreName`" substring must be replaced with the name of the user's core created previously:

```
<writer id="indexer_solr_1"
 class="org.apache.nutch.indexwriter.solr.SolrIndexWriter">
  <parameters>
    <param name="type" value="http"/>
    <param name="url"
     value="http://localhost:8983/solr/coreName/">
    <param name="collection" value=""/>
    <param name="weight.field" value=""/>
    <param name="commitSize" value="1000"/>
  </parameters>
</writer>
```

4.3.6 User Authorization

This subsection can be logically associated with the "Authorization" use case from the use case diagram.

For authorizing in the system, user will need to fill the login and password. If during check in the database no user with such login is found - an error will be outputted. If during check in the database user with such login is found - his password will be compared with the password received from user input and in case of match - user `session` object will be created, in case the password does not match - an error will be outputted. Authorization process also can be seen on the Authorization Flowchart (Figure 4.2).

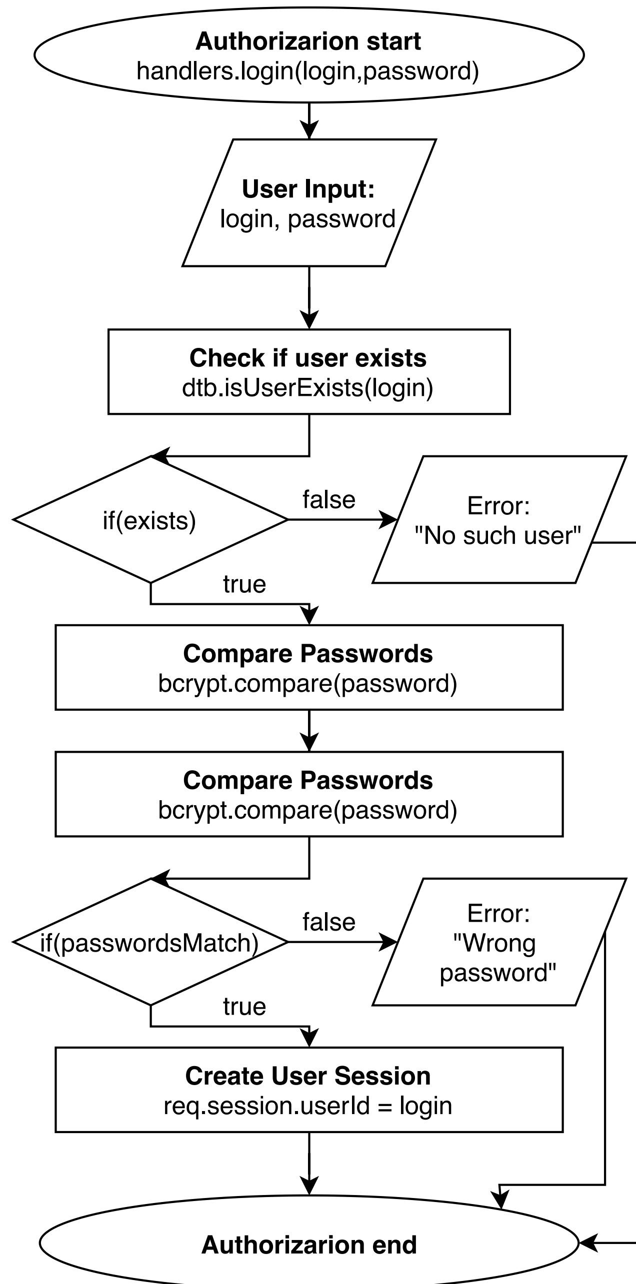

Figure 4.2: Authorization Flowchart

4.3.7 User Accounts Updating and Deletion

This subsection can be logically associated with the "Edit Account" and "Delete Account" use cases from the use case diagram.

For updating the account, user will need to pass values for updating, which could be new email or new password and pass the current password as well, the `userId` value will be taken from user's `session` object. After password comparing, when passwords match - new values will be written to the corresponding user object from the database. Account updating process also can be seen on the Account Updating Flowchart (Figure 4.3).

If user sends a request for account deletion - the `userId` value will be taken from user's `session` object and, after that, a number of deletion operations will be performed in the system:

1. The document in Users collection with a `login` value of `userId` will be deleted;
2. All documents in Datatypes collection with a `user` value of `userId` will be deleted;
3. All documents in Collections collection with a `user` value of `userId` will be deleted;
4. The crawldb of the user, if it was existing, will be deleted;
5. The user's Nutch configuration, if it was existing, will be deleted;
6. All user's Nutch files will be deleted;
7. The user's Solr core will be cleared from data and deleted;
8. The user's Solr configSet folder will be deleted;
9. The user's `session` object will be deleted and user will be logged out of the system.

Account Deletion process also can be seen on the Account Deletion Flowchart (Figure 4.4).

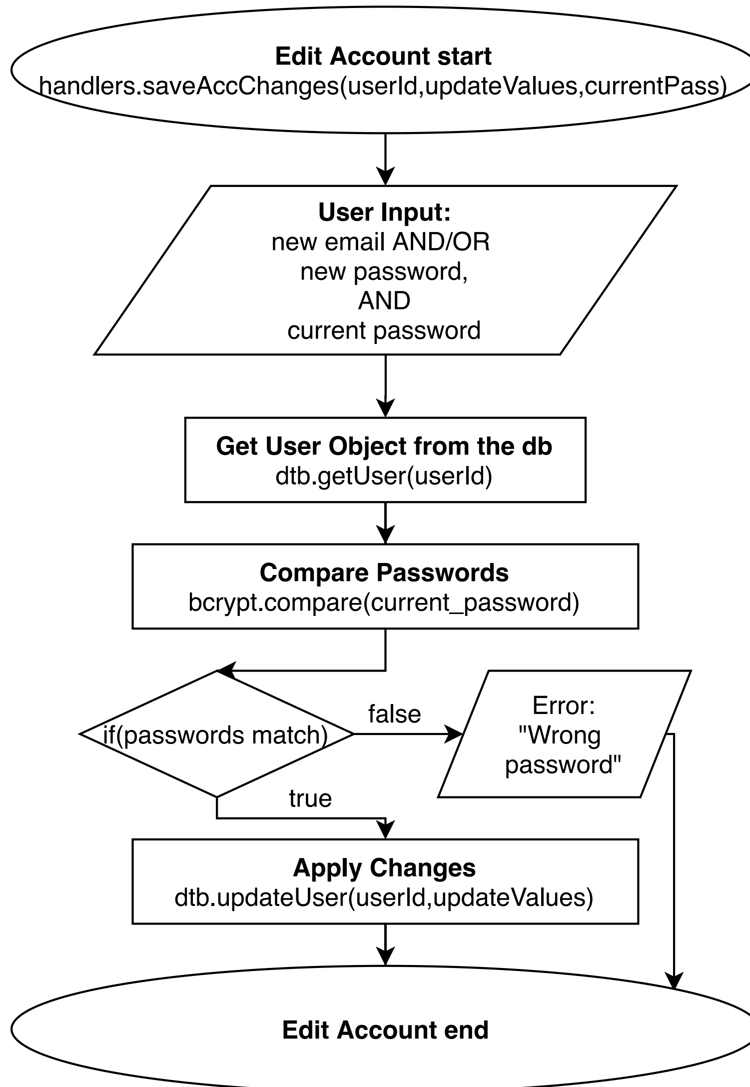

Figure 4.3: Account Updating Flowchart

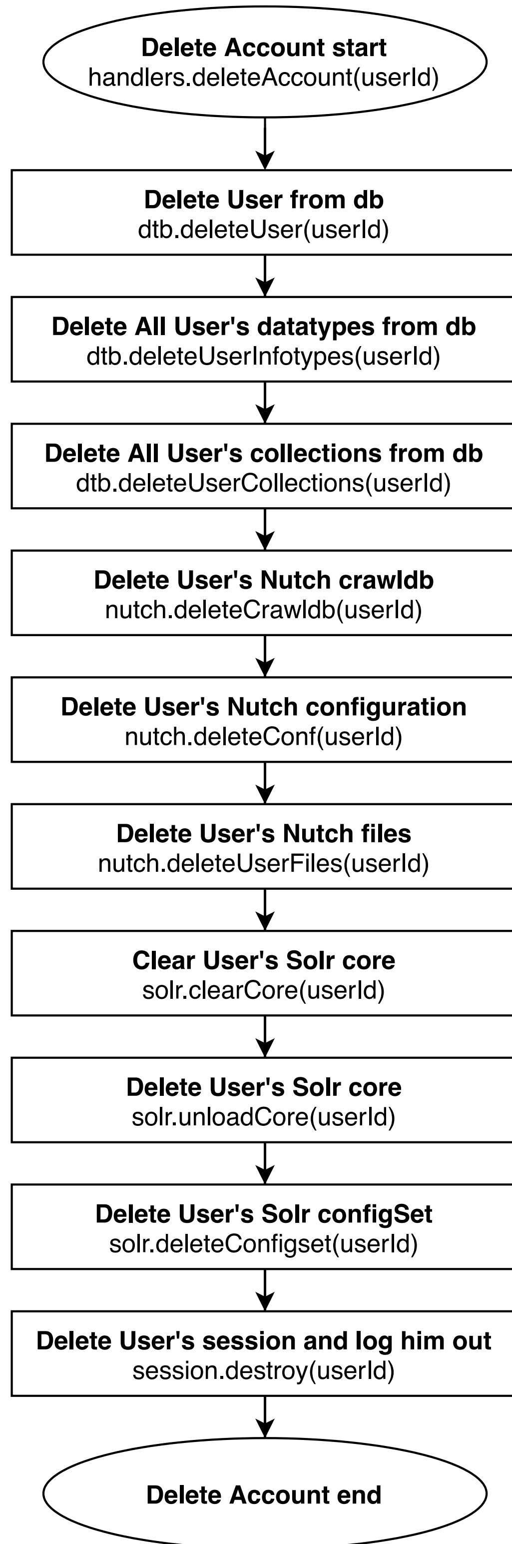

Figure 4.4: Account Deletion Flowchart

4.3.8 User Datatypes

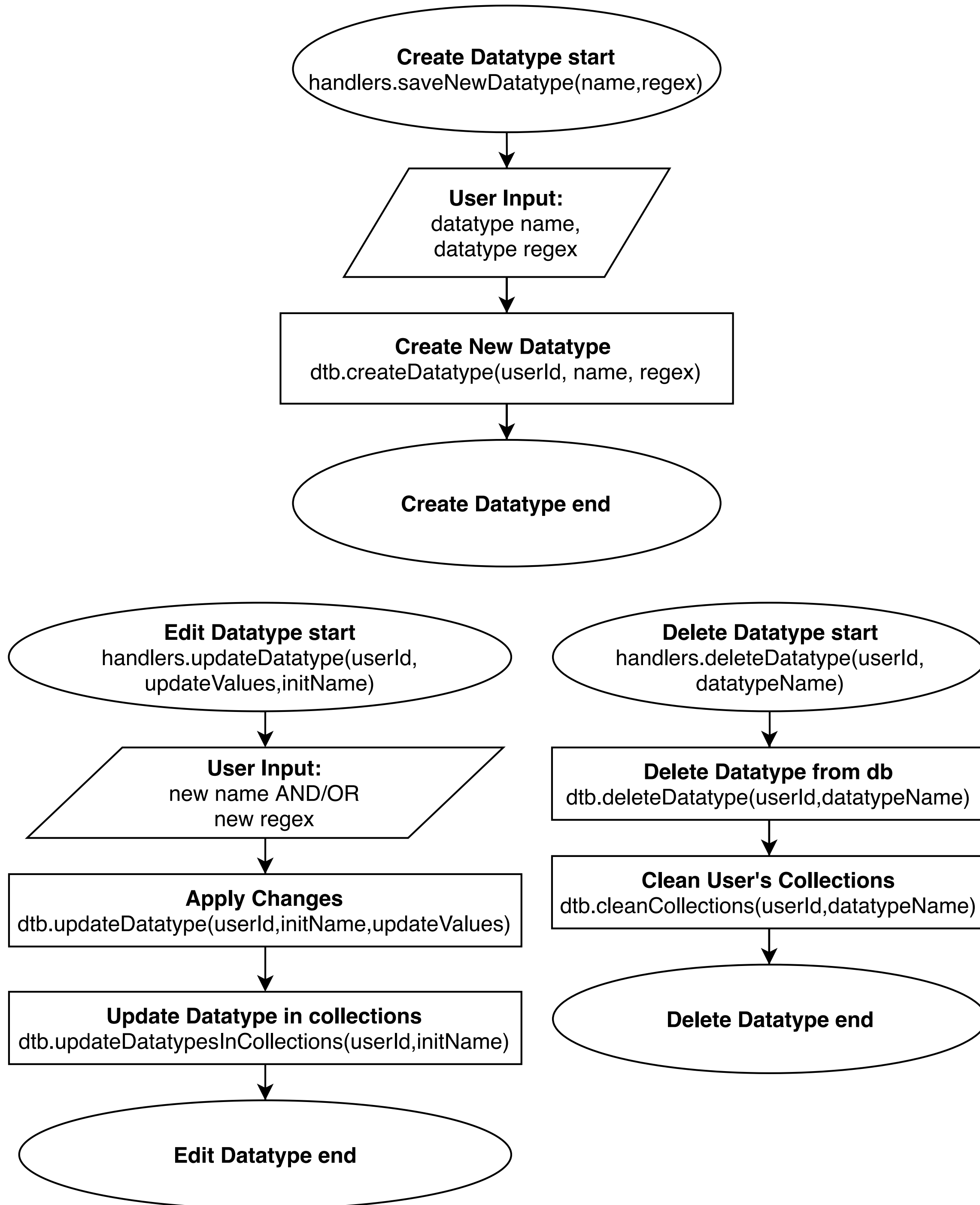

Figure 4.5: Datatypes Creation, Updating and Deletion Flowchart

In our system, "datatype" is a descriptive entity that can be used for detection of certain types of critical data among crawled Web sites' data. Each such entity will contain the name and a standard regular expression pattern for matching the desirable type of critical data.

A user's datatype can be created by a user of the system and can be included in collections of only that user.

This subsection can be logically associated with the "Manage Datatypes", "Create Datatypes", "Edit Datatypes" and "Delete Datatypes" use cases from the use case diagram.

1. Datatype creation: For creating a datatype - user must provide the datatype's name and datatype's regular expression values, after that, the document in Datatypes collection with those values and the `user` value of `userId` will be created. Datatype creation process can be also seen on the Datatypes Creation, Updating and Deletion Flowchart (Figure 4.5);

2. Datatype updating: For updating a datatype - user must provide the datatype's new name and/or datatype's new regular expression values, the value of initial datatype's name will be saved previously in the `initName` variable for purposes of searching of the datatype inside the db, after that, the corresponding fields of the document inside Datatypes collection with `name` value of `initName` and `user` value of `userId` will be updated with new, received values. Also all documents from Collections collection containing the updated datatype will be updated with new values too. Datatype updating process can be also seen on the Datatypes Creation, Updating and Deletion Flowchart (Figure 4.5);

3. Datatype deletion: If user will send a request for a particular datatype deletion - the `userId` and datatype's `datatypeName` values will be extracted from the request, after that, the document inside Datatypes collection with the `name` value of `datatypeName` and the `user` value of `userId` will be deleted. Also from all the documents from Collections collection with `user` value of `userId` - datatypes with value of `datatypeName` will be deleted too, this is required for keeping user's datatype

collections clear from deleted datatypes. If it will appear, that after collection's clearing, there are no datatypes left in it - the collection will be deleted. Datatype deletion process can be also seen on the Datatypes Creation, Updating and Deletion Flowchart (Figure 4.5).

4.3.9 User Collections

This subsection can be logically associated with the "Manage Collections", "Create Collections", "Edit Collections" and "Delete Collections" use cases from the use case diagram.

1. Collection creation: For creating a collection - user must provide the collection's name and collection's datatypes values. After that, the document in "Collections" collection with those values and the `user` value of `userId` will be created. Collection creation process can be also seen on the Collections Creation, Updating and Deletion Flowchart (Figure 4.6);

2. Collection updating: For updating a collection - user must provide the collection's new name and/or collection's new datatypes values, the value of initial collection's name will be saved previously in the `initName` variable for purposes of searching of the collection inside the db, after that, the corresponding fields of the document inside Collections collection with `name` value of `initName` and `user` value of `userId` will be updated with new, received values. Collection updating process can be also seen on the Collections Creation, Updating and Deletion Flowchart (Figure 4.6);

3. Collection deletion: If user will send a request for a particular collection deletion - the `userId` and collection's `collectionName` values will be extracted from the request, after that, the document inside Collections collection with the `name` value of `collectionName` and the `user` value of `userId` will be deleted. Collection deletion process can be also seen on the Collections Creation, Updating and Deletion Flowchart (Figure 4.6).

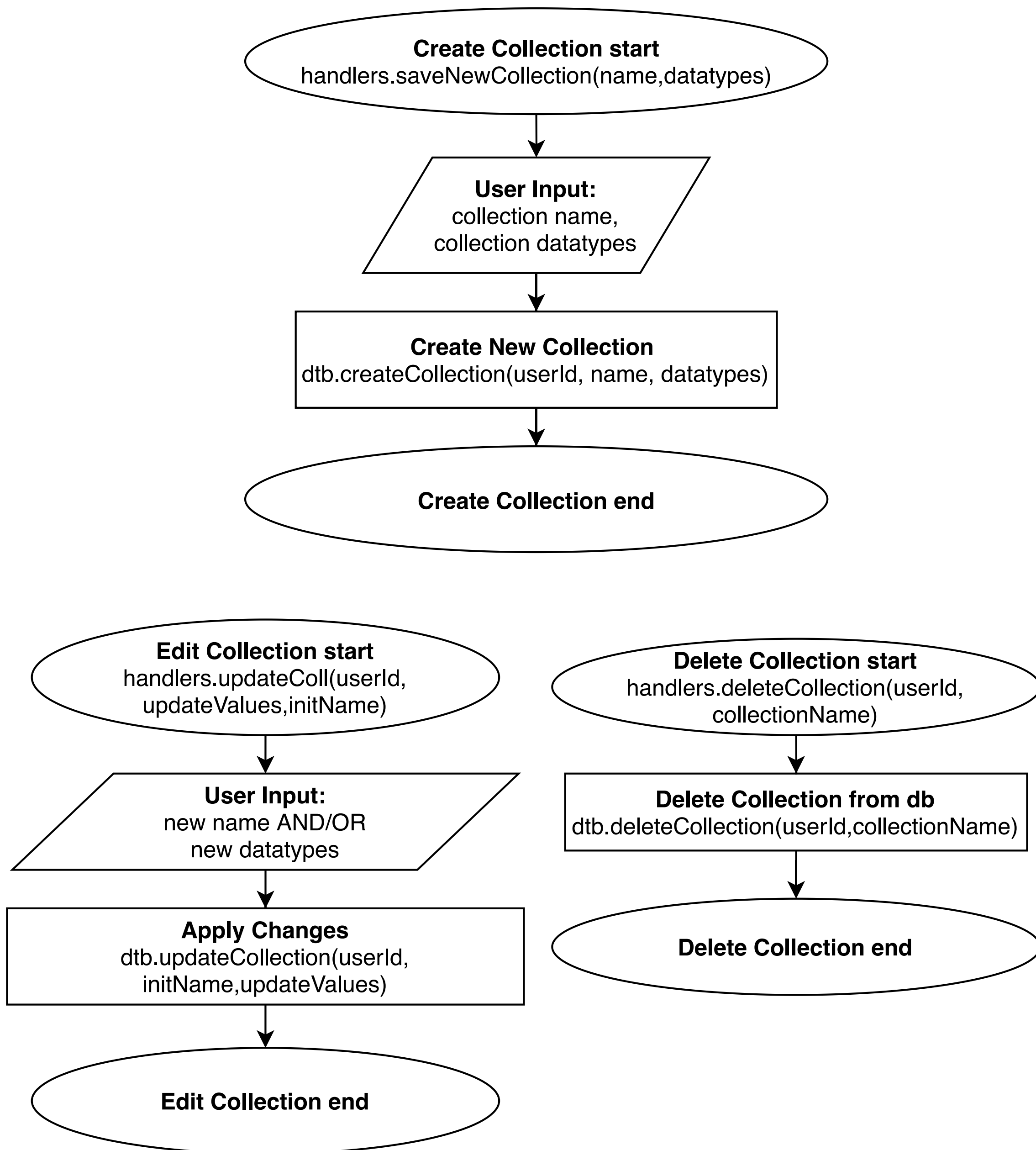

Figure 4.6: Collections Creation, Updating and Deletion Flowchart

4.3.10 Crawling

This subsection can be logically associated with the "Crawl Web sites" and "Receive Crawl end Email Notification" use cases from the use case diagram.

We divided the crawling process into two phases: the preparation phase and the crawling cycle.

Before starting an actual crawling cycle a set of functions must be invoked and finished in order to capture some of the errors before the beginning of the crawling cycle and not after the end of its first iteration.

First, user will pass the necessary values: a number of rounds for the crawling cycle to run, a list of urls to add to the crawldb, a list of regular expression-based url filters which will be used during Nutch Generate job for filtering list of urls to be fetched during Nutch Fetch job and a `userId` which will be extracted from user's `session` object.

Then a Nutch configuration for the user will be created, Nutch files will be filled with corresponding content received from the input and Nutch Inject job will be invoked by sending a POST request with the following JSON body:

```
{
    "type":"INJECT",
    "confId":"conf_of_user",
    "crawlId":"crawl_of_user",
    "args": {"url_dir":"users_files/user/urls"}
}
```

In the actual request bodies it is expected that all `"user"` substrings will be replaced with actual user's `userId` value.

If Nutch Inject job will fail - user will receive a notification about an occurred error almost immediately, but if we will put Nutch Inject job as a part of crawling cycle - the user will know about the error only after the first iteration of the cycle.

If Nutch Inject job will finish successfully - the crawl cycle will be started next and the fact that user has started the crawling cycle will be written to the database.

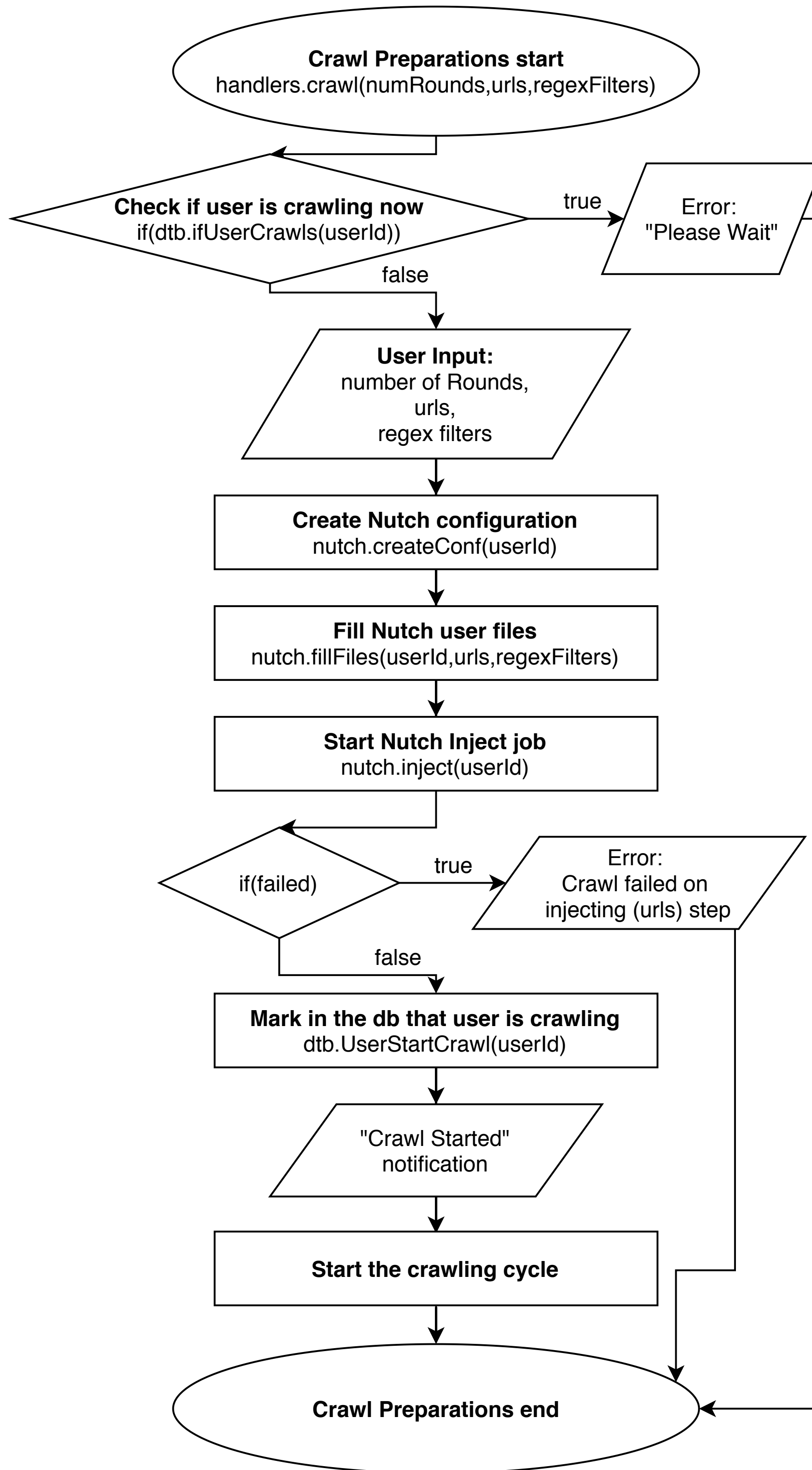

Figure 4.7: Crawl Preparations Flowchart

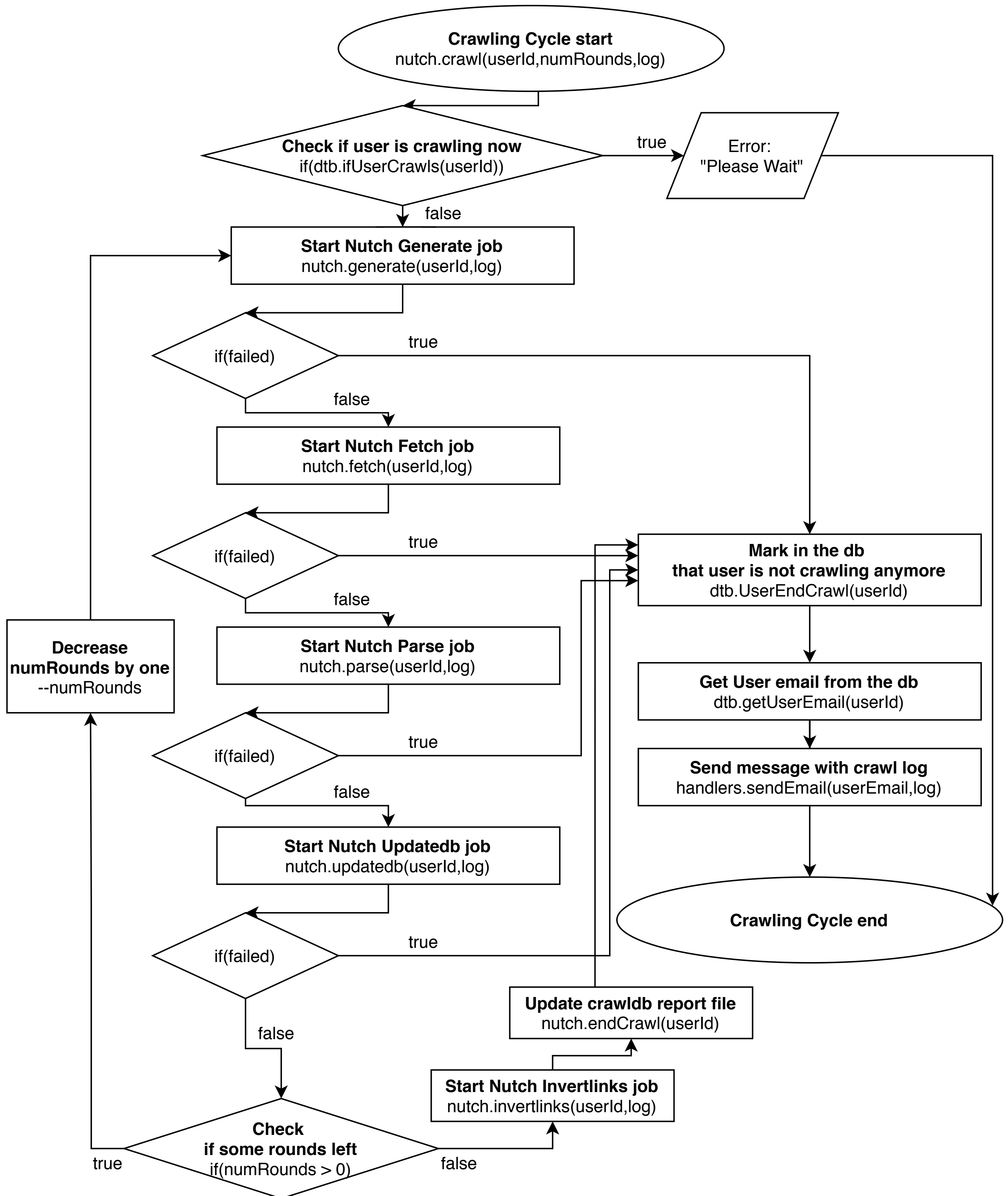

Figure 4.8: Crawling Cycle Flowchart

Crawling cycle will use the `numRounds` variable defined previously during preparation phase. Also before first iteration an empty `log` object will be created for recording statuses of the finished Nutch jobs and passed to the cycle.

The iteration of the crawl cycle consists of invocations of Generate, Fetch, Parse and Updatedb Nutch jobs one by one and tracking their completion statuses. If the job was completed successfully - the job's finishing status will be written to the `log` object and the next Nutch job will be invoked, on the contrary, if the job was not completed successfully - the job's finishing status will still be written to the `log` object, but then next jobs invocations will be cancelled and an email with `log` object as attachment will be sent to user's email, after that, further cycle iterations will be aborted.

If all four jobs finished successfully - the `numRounds` variable will be decreased by one and if, after that, it will not be equal to zero - a new cycle iteration will start, if it will be equal to zero - Nutch Inverlinks job will be invoked, after job's finish, Nutch Readdb operation will be invoked and will create a file with crawldb stats, if the old file existed - it will be replaced by the new one, then, an email with log object as attachment will be sent to user's email and crawling cycle will end.

Both Crawl Preparations phase and Crawling Cycle can be also seen on the corresponding Crawl Preparations (Figure 4.7) and Crawling Cycle (Figure 4.8) Flowcharts.

4.3.11 Crawldb Updating and Deletion

This subsection can be logically associated with the "Crawl Web sites", "Receive Crawl end Email Notification", "Observe the results of previously crawled data" and "Delete Previously Crawled Data" use cases from the use case diagram.

Obviously the user's crawldb will be created after the end of the first Nutch crawling cycle's iteration, but if user would like to update his crawldb, he will have to wait until the entire crawling process will end and an email, notifying about this will be send to him.

After the end of crawling process - user will have the ability to observe the stats of

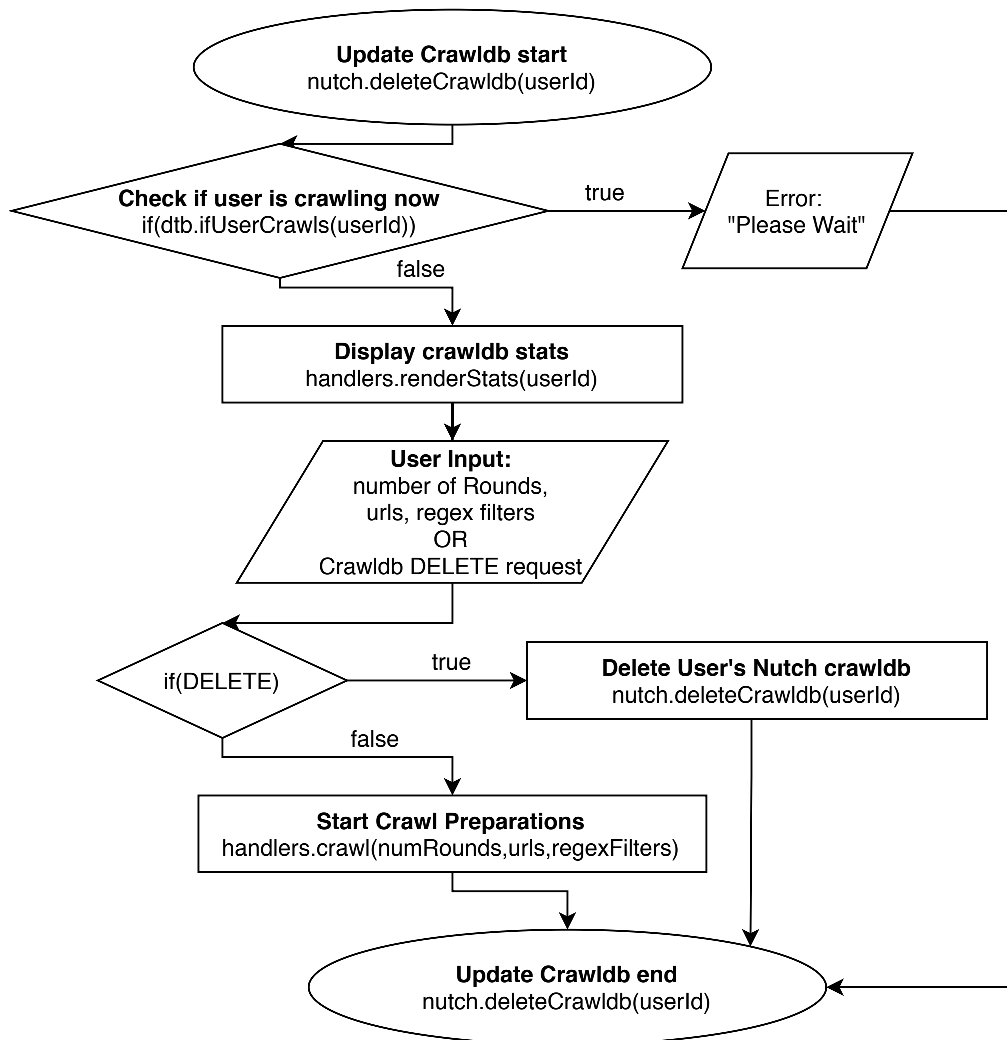

Figure 4.9: Update Crawldb Flowchart

the crawldb and continue to crawl by further invocations of another number of crawling rounds with or without passing new urls for injection into crawldb and with or without passing any updates for regular expression-based filters.

Also user will have an ability to delete his crawldb. After such request will be received - the crawldb folder assosiated with the user, located at `apache-nutch-1.16` folder and named "crawl_of_`user`" (where `user` substring must be replaced with the `userId` value extracted from the received request) will be deleted from the file system.

Crawldb updating process can be also seen on the Update Crawldb Flowchart (Figure 4.9).

4.3.12 Critical Data Detection

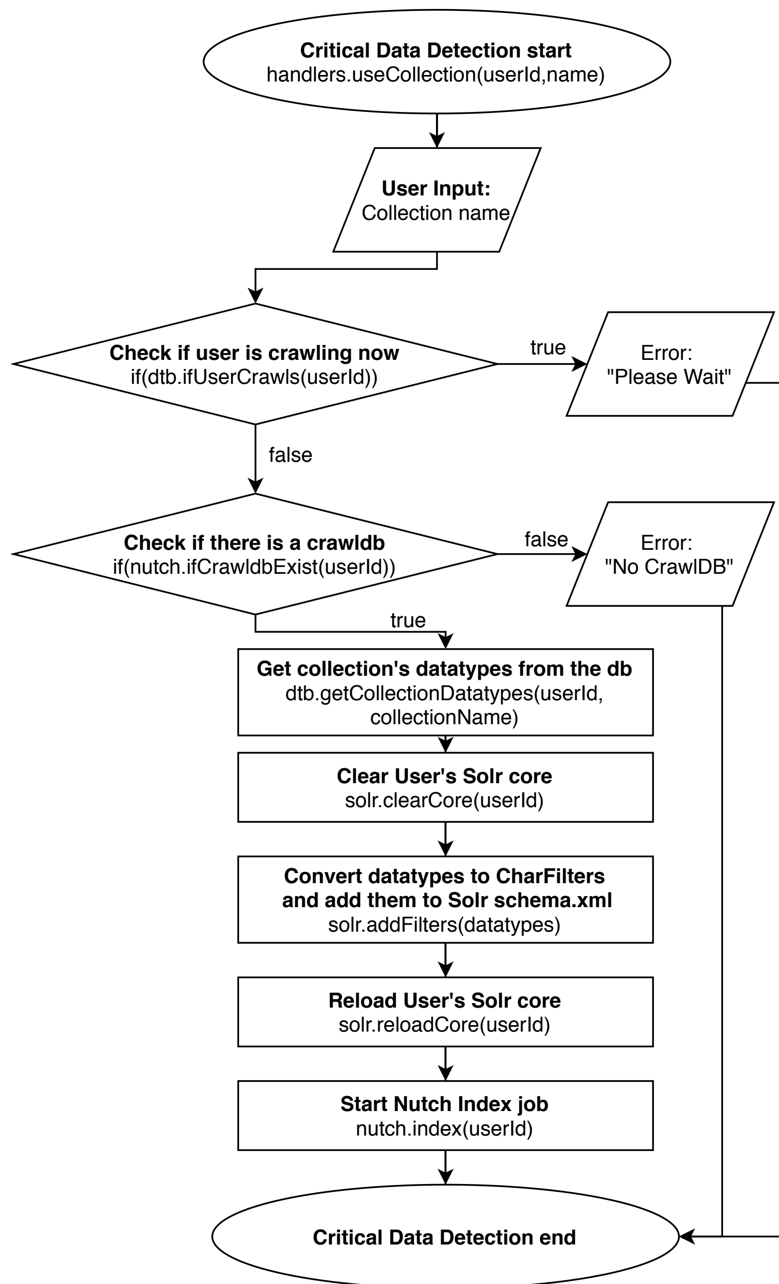

Figure 4.10: Critical Data Detection Flowchart

This subsection can be logically associated with the "Use a Collection for data detection" use case from the use case diagram.

First, user will need to pass the necessary values: the name of the collection of datatypes that will be used and a `userId` which will be extracted from user's `session` object. Then two checks will be performed: first will check if crawling cycle is running at the moment of invoking the detection process and second will check if there is a crawldb associated with the user. In case if crawling cycle associated with the user is still running or in case if user does not have a crawldb - corresponding error messages will be outputted.

If there is no running crawling cycle associated with the user and user has a crawldb - user's Solr core will be cleared, datatypes will be extracted from the collection and converted to Solr CharFilters, after that, those Solr CharFilters will be placed into Solr schema.xml file replacing the `"<!--{{charFilters}}-->"` substring.

Process of conversion of a datatype into a Solr CharFilter can be explained on the example. Consider the following datatype:

```
{
    "_id":"0123456789",
    "name":"IPB",
    "regex":"Instituto\sPolitecnico\sde\sBraganca",
    "user":"user123"
}
```

The conversion algorithm is to take the following CharFilter template and replace some values with corresponding values from the datatype object:

```
<charFilter class="solr.PatternReplaceCharFilterFactory"
pattern="(regex)"
replacement="user_name"/>
```

A Charfilter created from this datatype:

```
<charFilter class="solr.PatternReplaceCharFilterFactory"
pattern="(Instituto\sPolitecnico\sde\sBraganca)"
```

```
replacement="user123_IPB"/>
```

After modification of the schema.xml file - user's Solr core will be reloaded and Nutch Index job invoked, upon the end of the indexing all datatypes are already marked in all the indexed content.

Critical Data Detection process can be also seen on the Critical Data Detection Flowchart (Figure 4.10).

4.3.13 Observing Detection Results

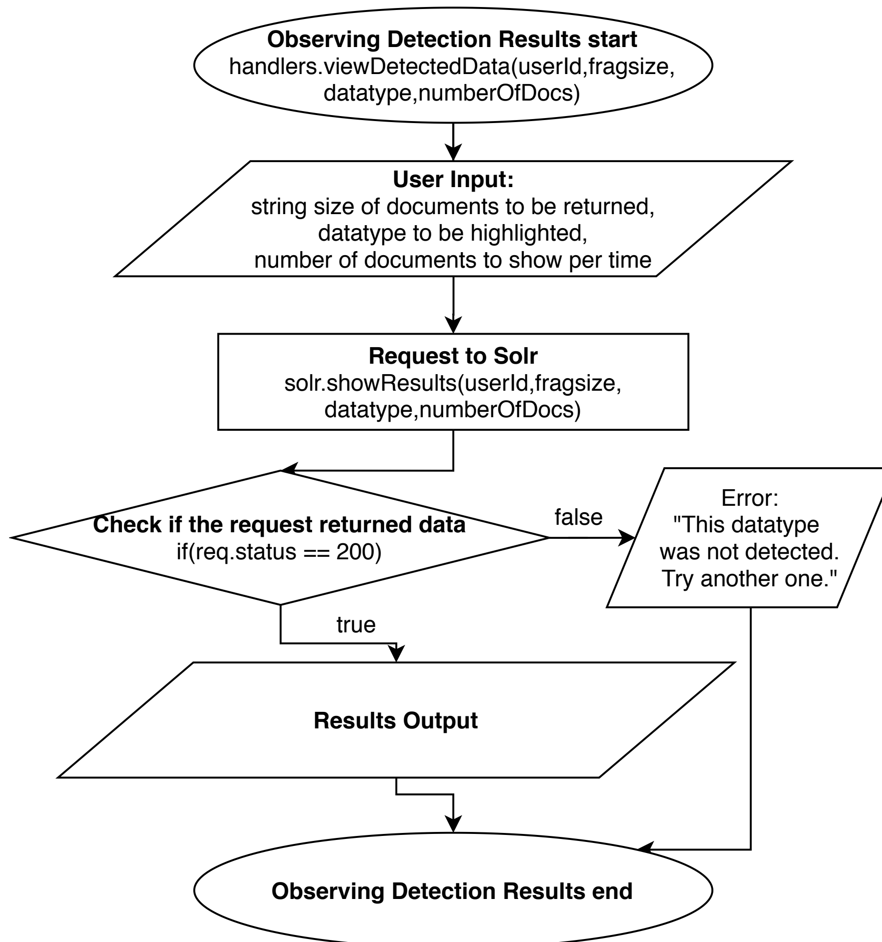

Figure 4.11: Observing Detection Results Flowchart

This subsection can be logically associated with the "Observe Detection Results" use

case from the use case diagram.

After data has been indexed and the datatypes detected and tagged, user will be able to look at the results of the detection. For this user must pass the following values which will define displaying rules for the results:

1. The `userId` value will be taken from user's `session` object;

2. The datatype which will be highlighted in returned content strings;

3. The number of documents to display per page. This is important, because directly affects the performance;

4. The `fragsize` number - this variable will define the amount of the content chars to be shown excluding the datatype occurrences. In case if the value of the `fragsize` is less or equal to the length value of the detected datatype - only exact datatype occurrences will be shown, in case if `fragsize` is a larger value than datatype occurrence length - datatypes occurrences will be shown together with some additional chars. This may be useful if user do not want to see only exact occurrences of the detected datatypes and do want to see some context in which those datatypes were detected. Also if user will set `fragsize` value to zero - the datatypes occurences will be shown in full content text.

Then, using values received from user, a GET request with Solr query will be generated and send to Solr `host:8983/solr/core_of_user/select` path in order to retrieve the data. The Solr query parameter (q) will be set to the following string, where `user` and `datatype` will be replaced with `userId` and `datatype` values correspondingly:

q=content:"**user_datatype**"

In that Solr query we state that we want to search for the occurrences of "user_datatype" string in the documents' "content" field. Other query parameters would be: the `fragsize` parameter, which will be set to user's `fragsize` value and the `row` parameter, which will be set to user's `numberOfDocs` value.

For highlighting the detected occurrences of the datatype inside data that will be received after the request - we will use Solr highlighting feature, which could be enabled by adding the following query parameters to the request:

```
hl=true&
hl.fl=content&
hl.fragsize=fragsize&
hl.q=content:user_datatype&
hl.snippets=21474836&
hl.simple.pre=<mark>&
hl.simple.post=</mark>&
```

The `hl=true` parameter will enable the Solr highlighting, `hl.fl` will state that the highlighting must be enabled only for documents' "content" field, `hl.fragsize` will be set to user's `fragsize` value, `hl.q` will contain the query for searching the substrings to be highlighted, `hl.snippets` is responsible for the setting up the limit of the substrings that could be highlighted inside one document, we will set it to the maximum value that Solr allows. The `hl.simple.pre` and `hl.simple.post` parameters define optional substrings which will be placed at the beginning and at the end of the highlighted substrings, we will use html marking tags for that.

Other parameters are:

```
rows=numberOfDocs&
wt=json
```

The `rows` parameter will be set to the value of user's `numberOfDocs` variable and the `wt` parameter will be set to "json" just for convenience of parsing the Solr response that wil be received into javascript object.

The user values will be included as request parameters. After that, a status of the sent request will be checked and in case if will be equal to 200, which means that the data was successfully received from Solr - an output will be generated and send to the user. In case if the status will not be equal to 200 - this will mean that the datatype passed by

user was not detected in the indexed data and the corresponding message will be shown.

Observing Detection Results process can be also seen on the Observing Detection Results Flowchart (Figure 4.11).

Chapter 5

Tests and Results

This chapter presents and describes the tests that were performed in order to check if the implemented system is able to detect datatypes in the data crawled previously by Nutch.

5.1 Testing of Preparatory Operations

Before testing the main functionality of the system - critical data detection, we perform and test several preparatory operations: authorize a user, crawl a number of Web sites, prepare a collection of datatypes. Each of these operations is having its own subsection.

5.1.1 Authorization

We created a user with `login` value of "user123", the `password` value of "123" and with `email` value of "user123@grr.la" (Figure 5.1, Figure 5.2).

INSTITUTO POLITÉCNICO DE BRAGANÇA

Sign In

Login

user123

Password

E-mail

user123@grr.la

Sign Up

Figure 5.1: User Creation Screenshot

INSTITUTO POLITÉCNICO DE BRAGANÇA

Sign Up

Your account created successfully

Login

login

Password

password

Sign In

Figure 5.2: User Creation Result Screenshot

We were expecting the new document with corresponding values to be appeared inside the database. We used the official GUI for MongoDB - MongoDB Compass for checking if the document for the new user was created in Users collection:

```
_id: ObjectId("5ee79b5f9d19273f9fd99ecb")
login: "user123"
password: "$2b$10$ve57IYYtkMZ8non/pOJsXOMAn2.QXLhuDqsAR/MLkHdEKM28sv2Oa"
email: "user123@grr.la"
crawls: false
last_detection_at: false
last_detected_datatypes: false
```

Figure 5.3: Screenshot of user123 in the MongoDB

From the screenshot above (Figure 5.3) we can see that the document was created. Then, we signed into the system using newly-created user's credentials (Figure 5.4, Figure 5.5).

Sign Up

Your account created successfully

Login

user123

Password

•••

Sign In

Figure 5.4: Login Screenshot

INSTITUTO POLITÉCNICO DE BRAGANÇA

Manage Custom Datatypes Crawl Account Sign Out

You don't have any collections

Create New Collection

Figure 5.5: Login Result Screenshot

We were expecting cookies to be appeared in the browser. We used the Google Chrome DevTools for checking if the cookies for the authorized user were created.

Name	Value	Dom...	Path	Expi...	Size	Http...	Secure	Sam...	Prior...
user_sid	s%3AlJ8u3MLa3Q6dfbhHMKkgdKXgN...	local...	/	202...	94	✓			Medi...
JSESSIONID	1anugxbsrolw5xzmj5z158h0l	local...	/	Sessi...	35				Medi...
io	lHDZ-2nj9ZiWc3moAAAE	local...	/	Sessi...	22	✓		Strict	Medi...
connect.sid	s%3Af3BeCplkRD63Pdxl7WjOMYGgOi...	local...	/	Sessi...	93	✓			Medi...
_ga	[illegible]186.1584018169	.ipb.pt	/	202...	30				Medi...
23d241263f8d921b9f25...	khgha86kr2sa3n9pf77irqh506	port...	/	Sessi...	58	✓			Medi...
cookieaccept	yes	port...	/	202...	15				Medi...

Figure 5.6: Cookies Screenshot

From the screenshot above (Figure 5.6) we can see that the cookies were created.

5.1.2 Crawling

Then we started a crawl cycle for 10 rounds by injecting the "http://ipb.pt/" url and "+." regular expression filter (this filter will accept all incoming urls) (Figure 5.7, Figure 5.8).

INSTITUTO POLITÉCNICO DE BRAGANÇA

Sign Out

You have no Crawldb yet. Start crawling to create it.

Urls:

http://ipb.pt/

Url regex filters:

accept anything else
+.

Number of rounds

10

Launch Crawl Process

Cancel

Figure 5.7: Crawl Setup Screenshot

INSTITUTO POLITÉCNICO DE BRAGANÇA

Manage Custom Datatypes Dashboard Sign Out

Please wait.
Crawling started.
Await for finish notification by email.
Do not perform Crawldb-related operations until the end of the crawl process.

OK, I understood

Figure 5.8: Crawl Invocation Result Screenshot

We were expecting Nutch Generate job to be invoked in the Nutch server, because invocation of this job is the first step of the crawling cycle iteration. We used the Nutch server logging terminal for checking if the job was started:

```
File  Edit  View  Search  Terminal  Help
Injector: urlDir: users_files/user123/urls
Injector: Converting injected urls to crawl db entries.
Injecting seed URL file file:/home/tucker/Documents/P/The Mawpath/apache-nutch-1
.16/users_files/user123/urls/seed.txt
Injector: overwrite: false
Injector: update: false
Injector: Total urls rejected by filters: 0
Injector: Total urls injected after normalization and filtering: 1
Injector: Total urls injected but already in CrawlDb: 0
Injector: Total new urls injected: 1
Injector: finished at 2020-06-15 23:48:32, elapsed: 00:00:01
Generator: starting at 2020-06-15 23:48:34
Generator: Selecting best-scoring urls due for fetch.
Generator: filtering: true
Generator: normalizing: true
Generator: topN: 1000
Generator: running in local mode, generating exactly one partition.
Generator: number of items rejected during selection:
Generator: Partitioning selected urls for politeness.
Generator: segment: crawl_of_user123/segments/20200615234836
Generator: finished at 2020-06-15 23:48:37, elapsed: 00:00:03
Fetcher: starting at 2020-06-15 23:48:40
Fetcher: segment: crawl_of_user123/segments/20200615234836
Fetcher: threads: 10
```

Figure 5.9: Nutch Terminal Screenshot

From the screenshot above (Figure 5.9) we can see that the job was successfully started.

Upon the finish of the crawling cycle - we checked the notification message and the attached log object that should have come to our email immediately after the end of crawling cycle.

From the screenshot below (Figure 5.10) we can see that the notification was successfully delivered to user's email.

Also, after each of crawling rounds, we were checking the amount of crawled Web pages (urls). We put those results in the chart below (Figure 5.11). In the chart, the X axis represents the number of crawling rounds, and the Y axis represents the number of crawled Web pages.

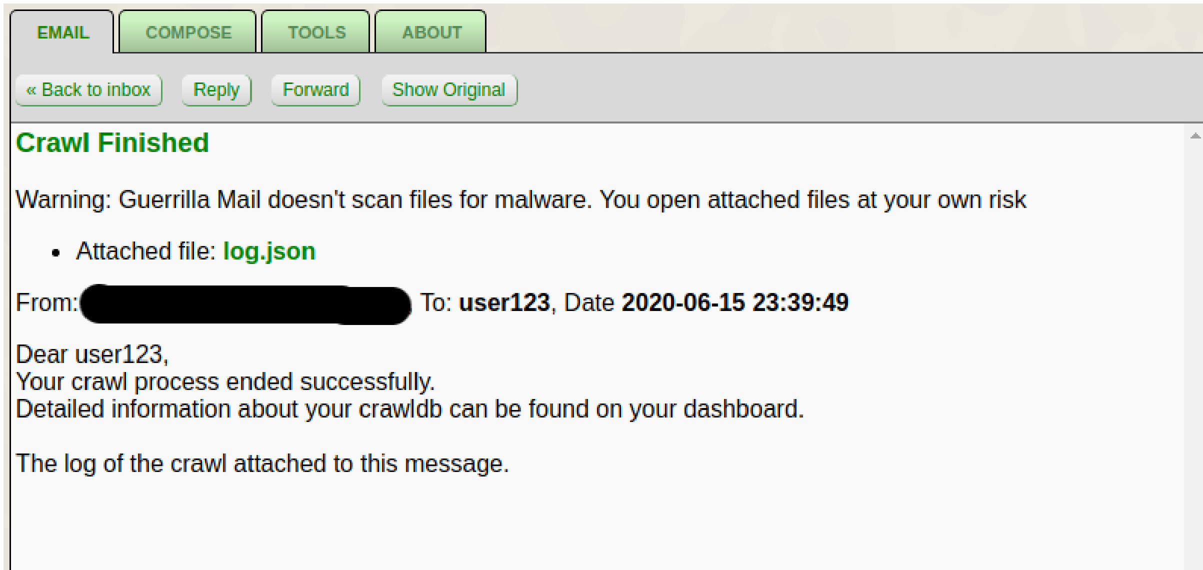

Figure 5.10: Received Email Screenshot

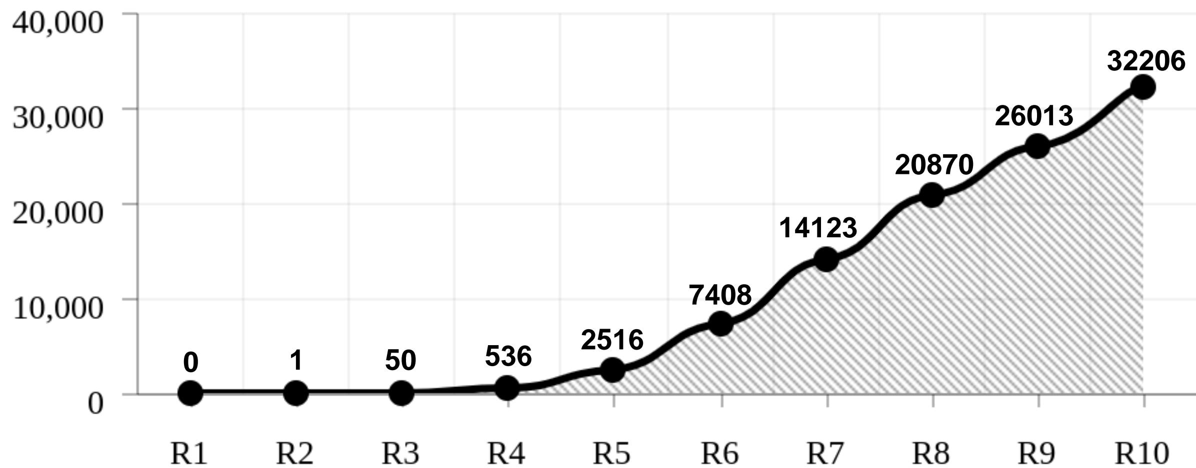

Figure 5.11: Crawling rounds chart

5.1.3 Creating a Collection

Then, we created a datatype (Figure 5.12, Figure 5.13), for detecting portuguese phone numbers, with the `name` value of "portuguese_phones" and the `regex` value of:

```
((\(?\+?351\)?)(\s)?(\d\d\d)(\s)?(\d\d\d)(\s)?(\d\d\d))|((\d\d\d
)\s?(\d\d\d)\s?(\d\d\d))
```

Figure 5.12: Datatype Creation Screenshot

Figure 5.13: Datatype Creation Result Screenshot

We were expecting the new document with corresponding values to be appeared inside the database. We used the official GUI for MongoDB - MongoDB Compass for checking if the document for new datatype was created in Datatypes collection:

```
_id: ObjectId("5eeeb5f9caf10709f1986f52")
name: "portuguese_phones"
regex: "((\(?\+?351\)?)(\s)?(\d\d\d)(\s)?(\d\d\d)(\s)?(\d\d\d))|((\d\d\d)\s?(\..."
user: "user123"
```

Figure 5.14: Datatype in the DB Screenshot

From the screenshot above (Figure 5.14) we can see that the document was created.

The same way, we created one more datatype, for detecting strings of time periods written in Portuguese, with the `name` value of "portuguese_time_periods" and the `regex` value of:

```
((([0-3]\d)(-|\.|\/)[0-1]\d(-|\.|\/)[1-2]\d\d\d|([0-3]?\d)\sde\s
((J|j)aneiro|(F|f)evereiro|(M|m)arço|(A|a)bril|(M|m)aio|(J|j)un
ho|(J|j)ulho|(F|f)agosto|(S|s)etembro|(O|o)utubro|(N|n)ovembro|(
D|d)ezembro)((\s|\sde\s)[0-2]\d\d\d)?|([0-3]?\d(-|\.|.\/)[0-1][1
-2])|\d)\sa\s(([0-3]\d)(-|\.|\/)[0-1]\d(-|\.|\/)[1-2]\d\d\d|([0-
3]?\d)\sde\s((J|j)aneiro|(F|f)evereiro|(M|m)arço|(A|a)bril|(M|m
)aio|(J|j)unho|(J|j)ulho|(F|f)agosto|(S|s)etembro|(O|o)utubro|(N
|n)ovembro|(D|d)ezembro)((\s|\sde\s)[0-2]\d\d\d)?|([0-3]?\d(-|\.
|.\/)[0-1][1-2])))|((uma?\s|dois\s|duas\s|tres\s|quatro\s|cinco\
s|seis\s|sete\s|oito\s|nove\s|dez\s)?(\ssegundos?|\sminutos?|\sh
oras?|\sdias?|\ssemanas?\smeses?|\smêses?|\sanos?))
```

It is expected that `portuguese_time_periods` datatype will be able to detect substrings which are describing time periods like, for example: "24 de janeiro a 08 de fevereiro de 2019", " 22/06/2020 a 04/07/2020" or just the simplest time periods that can consist of only one word like "ano", "dia", "hora" etc.

Then we created a collection (Figure 5.15, Figure 5.16) with the `name` value of "1st collection" and to the collection's `datatypes` value we added both datatypes - "portuguese_phones" and "portuguese_time_periods" we created previously:

Figure 5.15: Collection Creation Screenshot

Figure 5.16: Collection Creation Result Screenshot

We were expecting the new document with corresponding values to be appeared inside the database. We used the official GUI for MongoDB - MongoDB Compass for checking

if the document for new collection was created in "Collections" collection. From the screenshot below (Figure 5.17) we can see that the document was created.

```
_id: ObjectId("5eef7fb163820e7b974fee1b")
name: "1st collection"
datatypes: Object
   portuguese_phones: "((\(?\+?351\)?)(\s)?(\d\d\d)(\s)?(\d\d\d)(\s)?(\d\
   portuguese_time_periods: "((([0-3]\d)(-|\.|\/)[0-1]\d(-|\.|\/)[1-2]\d\
user: "user123"
```

Figure 5.17: Collection in the DB Screenshot

5.2 Main operations testing

After the preparatory operations were tested and demonstrated, we can start to test the core operations of the system - the operations that are letting the system to detect a particular user's collection of datatypes inside the crawled data.

5.2.1 Indexing and Detecting

As it was written before - processes of indexing and data detection are performed in the system simultaneously. We can start the data detection process by clicking on "use" (collection) button (Figure 5.18, Figure 5.19). It is expected that this collection which is consisting from only one "ipb_emails" datatype will be able to detect absolutely all IPB emails in crawled data:

Manage Custom Datatypes Crawl Account Sign Out

Your Collections:

1st collection

portuguese_phones
portuguese_time_periods

Edit Use

Figure 5.18: Collection Use Screenshot

Sign Out

Returned strings size (set to zero if you want only full text to return):

1

Choose a datatype:

portuguese_phones

Number of urls per time:

10

View detected pieces

Cancel

Figure 5.19: Collection Use Result Screenshot

We were expecting Nutch Index job to be invoked in the Nutch server, because invocation of this job is the first step of the detection process. We used the Nutch server logging terminal for checking if the job was started:

```
File  Edit  View  Search  Terminal  Help
Indexing 1000/48915 documents
Deleting 0 documents
Indexing 1000/49915 documents
Deleting 0 documents
Indexing 1000/50915 documents
Deleting 0 documents
Indexing 1000/51915 documents
Deleting 0 documents
Indexing 1000/52915 documents
Deleting 0 documents
Indexing 1000/53915 documents
Deleting 0 documents
Indexing 1000/54915 documents
Deleting 0 documents
Indexing 1000/55915 documents
Deleting 0 documents
Indexing 361/56276 documents
Deleting 0 documents
Indexer: number of documents indexed, deleted, or skipped:
Indexer:   2618  deleted (gone)
Indexer:   9045  deleted (redirects)
Indexer:  30361  indexed (add/update)
Indexer: finished at 2020-06-29 22:07:06, elapsed: 00:03:38
```

Figure 5.20: Nutch Indexing Log Screenshot

From the screenshot above (Figure 5.20) we can see that the job was successfully finished and in total 30361 Web Pages were converted to documents and indexed.

5.2.2 Observing Detection Results

Then we set the `Returned strings size` value to "0", so we could see entire content strings along with datatypes marked in them, the `Number of urls per time:` value to "10" and the `datatype` value to "portuguese_phones", after that, we submitted the form and received the following output (Figure 5.21).

The same way we looked on detection results of "portuguese_time_periods" datatype by changing the `datatype` value to "portuguese_time_periods" (Figure 5.22).

INSTITUTO POLITÉCNICO DE BRAGANÇA

Sign Out

Total urls with occurences of portuguese_phones: 10608

1. **http://www.cics2004.ipb.pt/alojaes.html**

Alojamientos Alojamientos NOTA: Conviene realizar la reserva rápidamente pues las fechas coinciden con el inicio de la EURO 2004 de fútbol, algunos alojamientos ya tienen reservas. Establecimiento Estrellas H. Sencilla H. Doble H. Triple Contactos (+351...) Teléfono Fax Pousada S. Bartolomeu Pousada - - - 273 331 493 273 323 453 Estalagem Turismo (1) **** - - - 273 310 700 273 310 701 Hotel S.Lázaro **** 100,00 € 120,00 € - 273 302 700 273 302 701 Residencial Classis **** 27,50 € 40,00 € 47,50 € 273 331 631 273 323 458 Albergaria Shalom *** 30,00 € 40,00 € - 273 331 667 273 331 628 Residencial Santa Apolónia *** 22,50 € 35,00 € 45,00 € 273 312 073 273 313 219 Residencial S.Roque *** 25,00 € 35,00 € - 273 381 481 273 326 927 Residencial Tulipa *** 30,00 € 38,00 € 50,00 € 273 331 675 273 327 814 Residencial Tic-Tac *** 30,00 € 35,00 € 45,00 € 273 331 373 273 331 673 Hotel S.José (2) *** 39,00 € 50,00 € - 273 331 578 273 331 242 Hotel IBIS (3) ** 33,00 € - - 273 302 520 273 302 569 Residencial Meirinhos ** 25,00 € 30,00 € 45,00 € 273 323 421 Pousada da Juventude 12,50 € (4) 35,00 € - 707 203 030 217 232 102 (1) 20,00 € por persona en habitación doble. (2) Existe la

Figure 5.21: portuguese_phones Datatype Detection Results Screenshot

INSTITUTO POLITÉCNICO DE BRAGANÇA

Sign Out

Total urls with occurences of portuguese_time_periods: 1579

1. **http://essa.ipb.pt/index.php/essa/alunos/horarios?format=feed&type=atom**

Escola Superior de Saúde - Horários Escola Superior de Saúde - Horários Escola Superior de Saúde de Bragança - ESSa Alunos - Horários - Título Horários (Início de Aulas a 17/02/2020) Alunos - Horários - Horários de Atendimento Horários de Atendimento Essa 1º Semestre 2019-2020Localização de salas de Aulas de Cursos da Escola Superior de Saúde (ESSa, ESA e ESTiG)Calendário Académico 2019/2020 Alunos - Horários - Mestrados Mestrados - Início do Semestre a 17/02/2020 Enfermagem Médico-Cirúrgica [1º Ano] Enfermagem de Saúde Materna e Obstetrícia [1º Ano] [Semana - 08/06/2020 a 12/06/2020] Enfermagem de Saúde Familiar [1º Ano] Enfermagem Comunitária [Estágio] Ciências Aplicadas da Saúde [Biotecnologia] [Intervenção Comunitária] [Proj/Est/Dis] Farmácia e Química de Productos Naturais [-] [2º Ano 1º Semestre] Alunos - Horários - Pós-Graduações Pós-Licenciaturas/Pós-Graduações Pós-Graduação em Gestão em Enfermagem [1ºAno] Alunos - Horários - Licenciaturas Licenciaturas - Início do Semestre a 17/02/2020 Dietética e Nutrição [1º Ano][Semana - 08/06/2020 a 12/06/2020][Bioestatistica I -12/06/2020 a 18/06/2020] [2º Ano][Semana - 01/06/2020 a 05/06/2020][Semana - 15/06/2020 a 19/06/2020] [3º Ano] [Estágio]

Figure 5.22: "portuguese_time_periods" Datatype Detection Results Screenshot

We can see that out of total number of 30361 indexed Web pages' content:

1. The results of the performed detection process of the "portuguese_phones" datatype are 10608 urls with highlighted occurrences of this datatype, inside all the pages on which they were found.

2. The results of the performed detection process of the "portuguese_time_periods" datatype are 1579 urls with highlighted occurrences of this datatype.

5.3 Comparison with Search Engines

This section describing the process of comparing the system's detection results compared to the similar detection results that could be obtained by using Google and Yandex search engines. User, datatypes and the collection from the previous sections are referenced throughout this section.

Some regular expressions are not so complicated and could be converted to a non-identical, but similar wildcard search pattern for further search queries to Google or Yandex.

For example, "portuguese_phones" datatype is not very complicated and can be converted to the "+351*" wildcard, and although such wildcard will not detect phone numbers without "+351" substring - users will still be able find very similar lists of results using, for example, Google or Yandex.

The search query for searching phone numbers by "+351*" wildcard will be:

```
site:ipb.pt +351*
```

The results received after requests with such search query are 7,290 results from Google and only 762 results from Yandex (Figure 5.23, Figure 5.24).

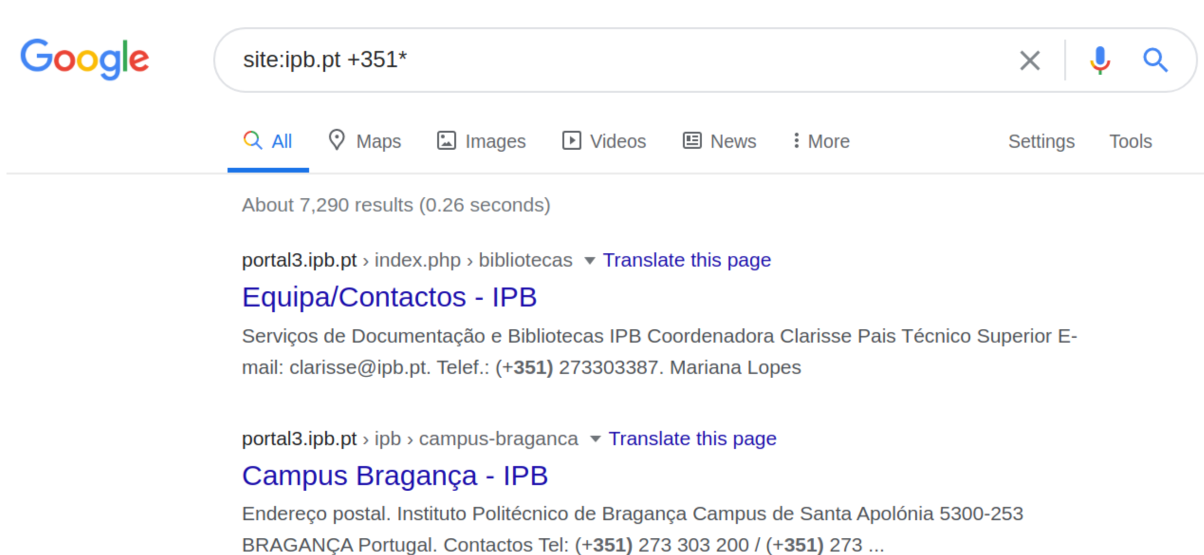

Figure 5.23: Google Search Results

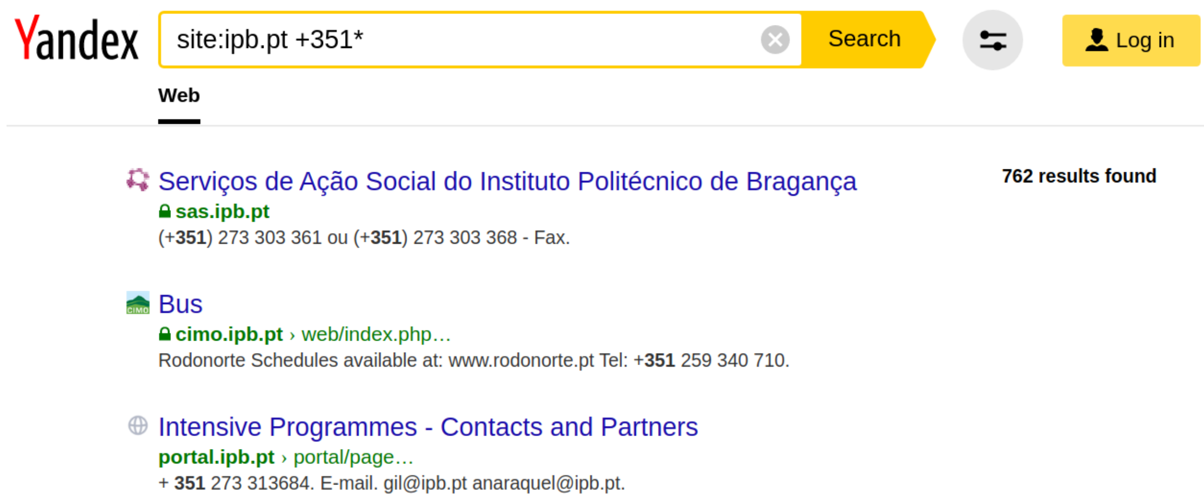

Figure 5.24: Yandex Search Results

According to the results, our system is already detected more Portuguese phone numbers than Yandex or Google did, despite of the fact that we indexed only 30361 Web pages, while Yandex or Google, surely, indexed more.

In order to obtain the number of ipb.pt domain Web pages indexed by Google and Yandex we performed the following search query to each search engine:

```
site:ipb.pt
```

The results were "About 95,200" - from Google, and "7 thousand results found" - from Yandex. Basing on these numbers we created the following chart (Figure 5.25) for displaying the ratio of the number of indexed pages to the number of found pages with occurences of Portuguese phone numbers.

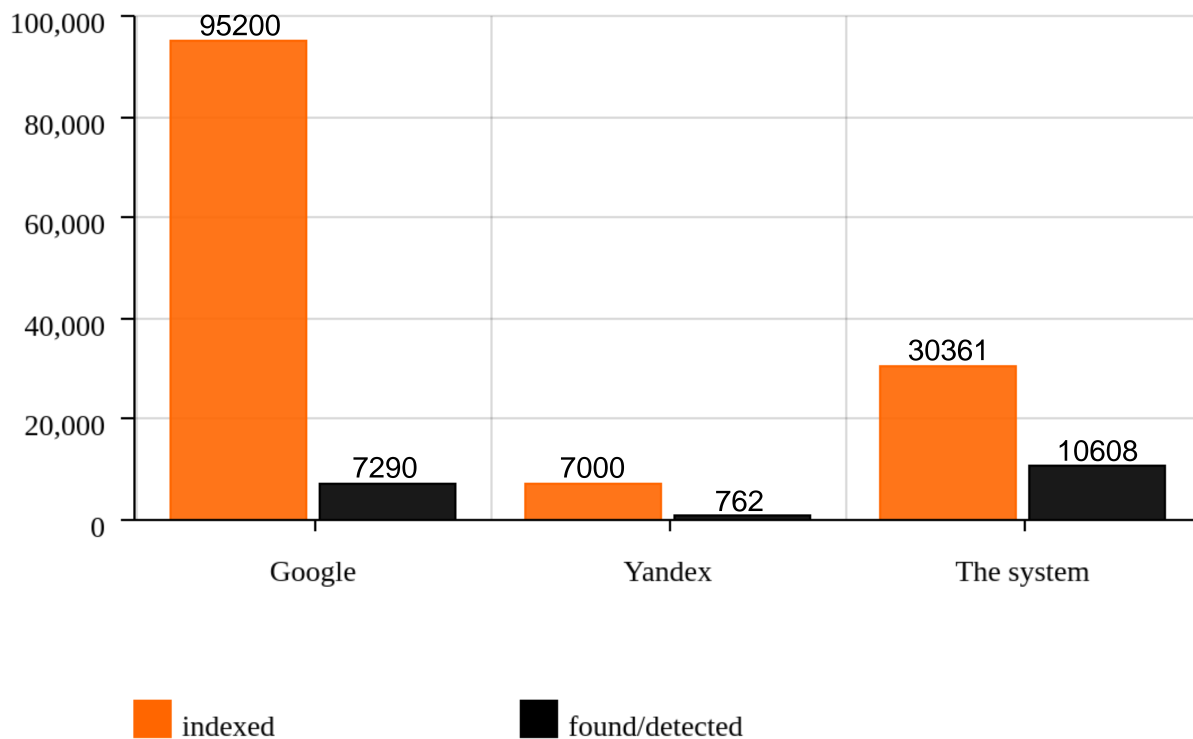

Figure 5.25: Indexed pages and pages with detected Portuguese phone numbers

Such detection of desirable datatypes by using search engines are also could be accepted by users, although the final results, obviously, lack the accuracy and the context with adjustable length.

But in case if a datatype's regular expression is much more complicated like, for example, in the "portuguese_time_periods" datatype - conversion of such datatype to a wildcard will take more time and most likely will be less efficient.

Also, in case if a regular expression contains almost no text characters and most of the other characters are related only with regular expressions and do not have their wildcard analogues - it could be hard or even impossible to convert such regular expression to a

wildcard search string which would be able to get similar search results.

For example, if the user with `login` value of "user123" would like to detect absolutely all email addresses inside the crawled data, including email addresses with the second domain, he would have to create the following datatype:

```
"name":"emails",
"regex":
"\w+@\w+\.\w+(\.\w+)?",
"user":"user123"
```

Then, he would have to attach that datatype to our "1st collection" collection and use it for data detection again. The detection results would be 9362 urls with occurences of email addresses (Figure 5.26).

Sign Out

Total urls with occurences of emails: 9362

1. **http://portal3.ipb.pt/images/ipb/imagem/lista.pdf**

Lista interna do IPB EDIÇÃO DE JANEIRO DE 2018 Um espaço internacional 2 Nome abreviado Área/Departamento Porta Extensão E-mail Presidência Sobrinho Teixeira Presidente 3204 sobrinho@ipb.pt Luís Pais Vice-Presidente 3203 pais@ipb.pt Orlando Rodrigues Vice-Presidente 3208 orlando@ipb.pt Elisabete Vicente Madeira Administradora 3202 elisabete@ipb.pt Pró-Presidentes Albano Alves Sistemas de Informação 3001 albano@ipb.pt Anabela Martins Imagem e Apoio ao Estudante 3390 giape@ipb.pt Dina Jerónimo Macias Assuntos Académicos 3831 dmacias@ipb.pt José Adriano Empreendedorismo 87 3077 adriano@ipb.pt Receção Fátima Vaqueiro Telefonista 3200 vaqueiro@ipb.pt Maria Rosário Alves Telefonista 3000 mariarosario@ipb.pt Segurança 3290 Gabinete Apoio ao Presidente e Relações Públicas ipb@ipb.pt Sandra Maria Cascais Madeira 3222 sandra@ipb.pt Olga Padrão 3305 olga@ipb.pt Gabinete de Imagem e Apoio ao Estudante giape@ipb.pt Ana Fernandes Ribeiro Azevedo Coordenação ESSa 3962 anitaazevedo@ipb.pt Carla Sofia Veiga Fernandes Coordenação ESTiG 3127 cveiga@ipb.pt Cristina Mesquita Coordenação ESE 3630

Figure 5.26: "emails" Datatype Detection Screenshot

The result is showing all the urls with occurrences of any email addresses. Datatypes like the one described above ("emails" datatype) are filled with mostly generic symbols and will not be completely convertible into wildcard search pattern.

If we will try to search in Google or Yandex with the following search queries

```
site:ipb.pt *@*
site:ipb.pt *@*.*
```

we will get no results from Google and will get 670 chaotic results from Yandex which are not related with email addresses (Figure 5.27, Figure 5.28).

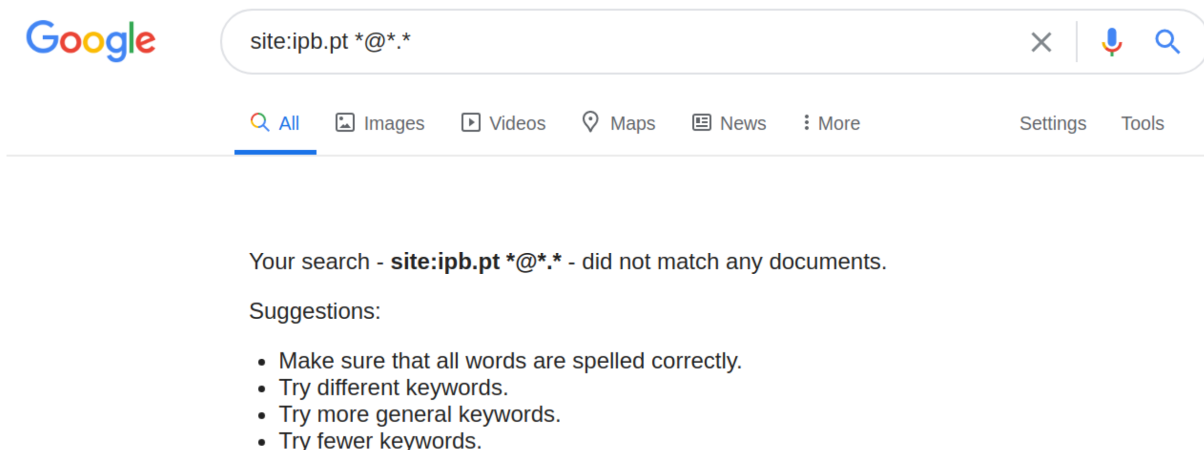

Figure 5.27: Google "emails" Datatype Detection Screenshot

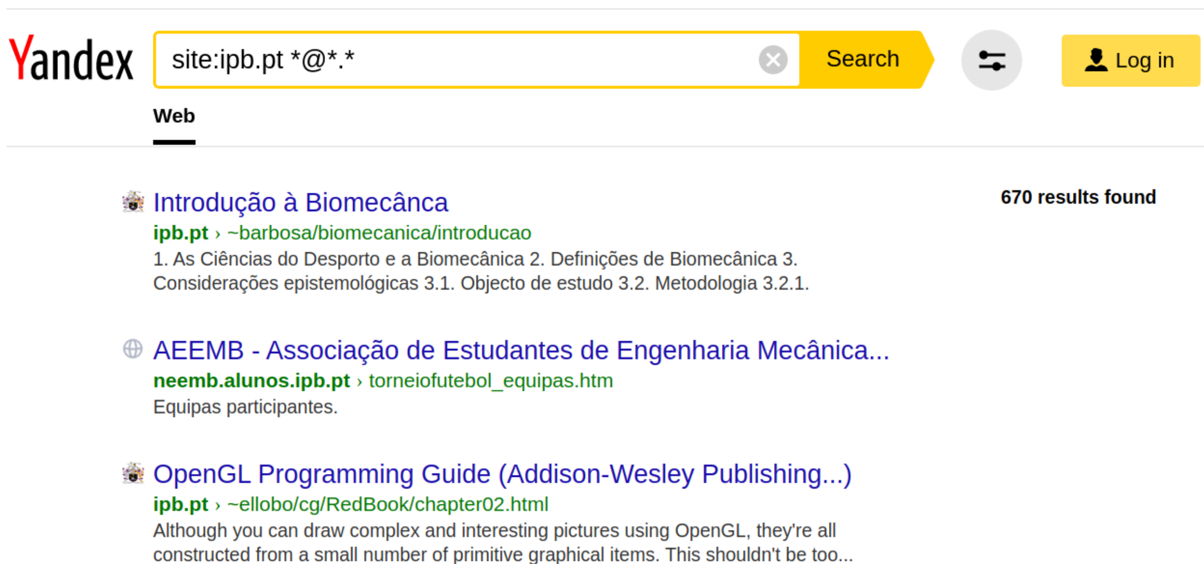

Figure 5.28: Yandex "emails" Datatype Detection Screenshot

Based on results from search engines and the system, we created the following chart

(Figure 5.29) for displaying the ratio of the number of indexed pages to the number of found pages with occurences of email addresses:

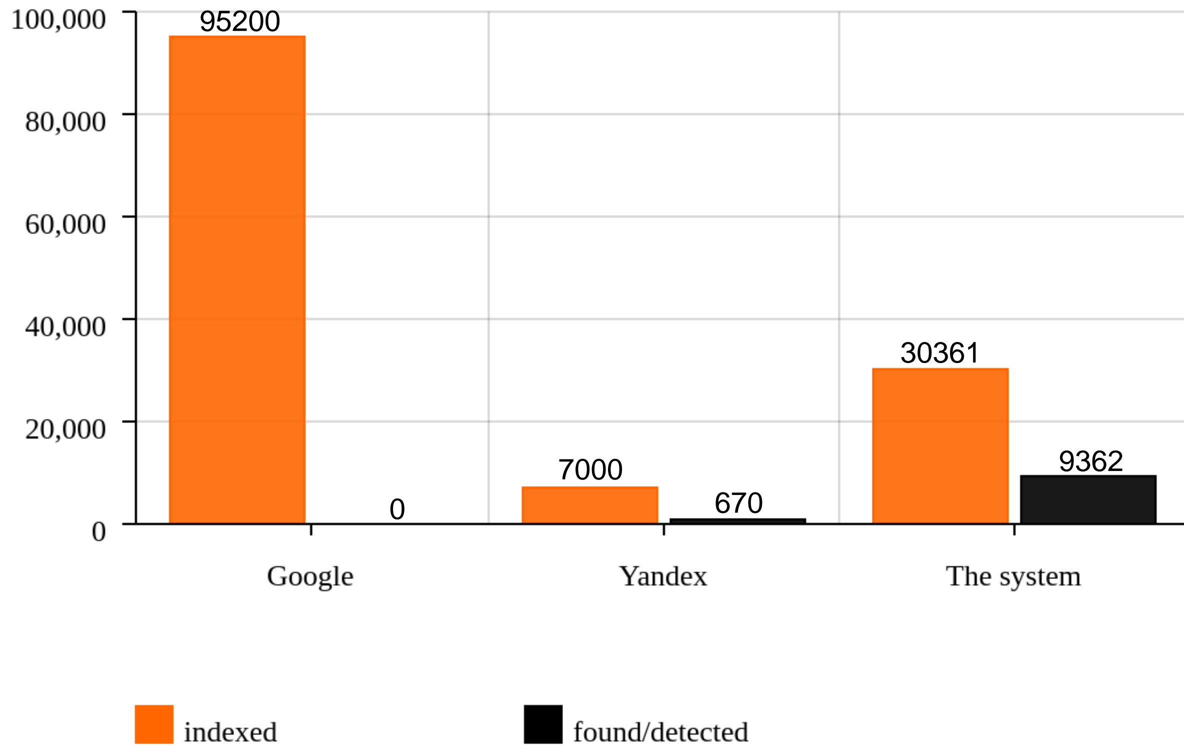

Figure 5.29: Indexed pages and pages with detected email addresses

So, as it was written above - in case if a regular expression contains almost no text characters and most of the other characters are related only with regular expressions and do not have their wildcard analogues - it could be hard or even impossible to convert such regular expression to a wildcard search string which would be able to get similar search results.

Chapter 6

Conclusions and Future Work

In this work, in order to solve the problem of critical data leaks detection in institutions' public Web sites, it was implemented a multiuser system, which is allowing its users to crawl data from public Web sites of different institutions by using the functionality of Apache Nutch, and detect several critical types of data simultaneously by using the functionality of Apache Solr, all integrated into a developed system.

Further sections of this chapter describe conclusions that were made after the end of the work and a number of suggestions for future system's improvement.

6.1 Conclusions

The implemented system allows users to detect and overview a set of desirable critical types of data within these types of data context with adjustable length. In such way, the system may help users to detect probable data leaks, while whether to consider and delete the detected data on the corresponding Web pages as data leaks or not - is a particular user's decision.

System's abilities let users to create their own, desirable datatypes and organize them into collections for detecting multiple datatypes simultaneously. In such way users can combine human mind capabilities and trustworthiness of the system as an algorithm in order to achieve the flexibility during data identifying processes and reliability during

data detection ones.

In comparison with some of the most common search engines - Google and Yandex, the system's data detection processes require much more time, but unlike search engines - the system is performing data detection basing on regular expressions and displaying results inside context with adjustable length. Modern search engines are incapable of performing regular expression-based search. Even if a particular datatype is already inside content of pages indexed by search engines and such datatype could be found by the wildcard search - most likely such datatype could be considered as a leaked one automatically and an organization will need to put a lot of effort for stopping further copying and spreading of the datatype in the Web. Not to mention how difficult it is to delete information once indexed by Google or Yandex also how unsafe it is to search for critical datatype using them, because in case if user's searching requests will be compromised and decrypted - the datatype could be considered leaked as well.

Because of Apache Nutch usage, the implemented system can be used for detecting the desirable critical data regardless of the fact if a particular Web page or an entire Web site was already indexed by search engines or not.

6.2 Future work

The basic functional parts of the system could be considered finished, but the system can be improved greatly by expanding already implemented crawling and data detection functions.

Crawling functions of the system could be improved by adding possibilities that will let users:

1. to schedule crawl processes;

2. to have and manage several Nutch crawl databases;

3. to launch several crawling processes simultaneously;

4. to have and manage several Solr cores;

5. to have an ability to configure the structure of Solr documents to be indexed.

Datatype detection functions of the system could be also improved by studying the existing works about modern machine learning algorithms in data security and by further creating and adding some machine learning algorithms for detecting critical types of data without user's interference, using such algorithms as alternative or additional measures for detection of critical data. Other existing Data Leak Detection systems should be checked for a possibility of intergration with our system for improving the results of data detection processes.

And in general, the system could be improved in order to make it more user-friendly by creating a better user interface and giving users an ability to search for desirable datatypes not only by using regular expressions but also by using wildcard patterns, because, obviously, this will decrease the requirements for the system's users and will make the system available to more of them.

Bibliography

[1] A. Skrop, “Dataleak: Data leakage detection system”, *MACRo 2015*, vol. 1, Jan. 2015. DOI: 10.1515/macro-2015-0011.

[2] P. Raman, H. G. Kayacık, and A. Somayaji, “Understanding data leak prevention”, in *6th Annual Symposium on Information Assurance (ASIA’11)*, Citeseer, 2011, p. 27.

[3] H. Balinsky, S. J. Simske, and D. S. Perez, *Data leak prevention systems and methods*, US Patent 9,219,752, Dec. 2015.

[4] *Hivecode DLD*, https://hivecode.io/data-leak-detection.html, Accessed: 2020-01-20.

[5] *Google Cloud DLP*, https://cloud.google.com/dlp, Accessed: 2020-01-20.

[6] R. Accorsi, A. Lehmann, and N. Lohmann, “Information leak detection in business process models: Theory, application, and tool support”, *Information Systems*, vol. 47, pp. 244–257, 2015.

[7] C. Corrodi, T. Spring, M. Ghafari, and O. Nierstrasz, “Idea: Benchmarking android data leak detection tools”, in *International Symposium on Engineering Secure Software and Systems*, Springer, 2018, pp. 116–123.

[8] *Node.js "About"*, https://nodejs.org/en/about/, Accessed: 2020-01-20.

[9] *NPM "About"*, https://docs.npmjs.com/about-npm/, Accessed: 2020-01-20.

[10] *What is MongoDB?*, https://www.mongodb.com/what-is-mongodb, Accessed: 2020-01-20.

[11] S. Khalil and M. Fakir, "Rcrawler: An r package for parallel web crawling and scraping", *SoftwareX*, vol. 6, pp. 98–106, 2017.

[12] *Scrapy 1.8 documentation*, https://docs.scrapy.org/en/1.8/, Accessed: 2020-01-20.

[13] *ScrapeStorm Document Center*, https://www.scrapestorm.com/?type=tutorial&cat_id=44, Accessed: 2020-01-20.

[14] *Octoparse resources*, https://www.octoparse.com/help, Accessed: 2020-01-20.

[15] *The 80legs Developer Hub*, https://developer.80legs.com/, Accessed: 2020-01-20.

[16] *PySpider documentation*, http://docs.pyspider.org/en/latest/, Accessed: 2020-01-20.

[17] *Apify documentation*, https://docs.apify.com/, Accessed: 2020-01-20.

[18] *Apache Nutch wiki*, https://cwiki.apache.org/confluence/display/NUTCH/Home, Accessed: 2020-01-20.

[19] *Hadoop MapReduce Documentation*, https://cwiki.apache.org/confluence/display/NUTCH/MapReduce, Accessed: 2020-01-20.

[20] *Apache Nutch release announcements*, http://nutch.apache.org/, Accessed: 2020-01-20.

[21] J. Beall, "The weaknesses of full-text searching", *The Journal of Academic Librarianship*, vol. 34, no. 5, pp. 438–444, 2008.

[22] *Apache Solr Documentation*, https://lucene.apache.org/solr/resources.html, Accessed: 2020-01-20.

[23] *Elastic Stack Documentation*, https://www.elastic.co/guide/index.html, Accessed: 2020-01-20.

[24] *Art. 4 GDPR Definitions*, https://gdpr-info.eu/art-4-gdpr/, Accessed: 2020-01-20.

[25] *NutchTutorial-Step-by-Step:Concepts*, https://cwiki.apache.org/confluence/display/nutch/NutchTutorial#NutchTutorial-Step-by-Step:Concepts, Accessed: 2020-01-20.

[26] *Nutch Inject*, https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=122916815, Accessed: 2020-01-20.

[27] *Nutch Generate*, https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=122916817, Accessed: 2020-01-20.

[28] *Nutch Fetch*, https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=122916820, Accessed: 2020-01-20.

[29] *Nutch Parse*, https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=122916830, Accessed: 2020-01-20.

[30] *Nutch Updatedb*, https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=122916836, Accessed: 2020-01-20.

[31] *Nutch Invertlinks*, https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=122916838, Accessed: 2020-01-20.

[32] *Nutch Index*, https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=122916842, Accessed: 2020-01-20.

[33] *NPM fs-extra package*, https://www.npmjs.com/package/fs-extra, Accessed: 2020-01-20.

[34] *NPM express package*, https://www.npmjs.com/package/express, Accessed: 2020-01-20.

[35] *NPM body-parser package*, https://www.npmjs.com/package/body-parser, Accessed: 2020-01-20.

[36] *NPM cookie-parser package*, https://www.npmjs.com/package/cookie-parser, Accessed: 2020-01-20.

[37] *NPM express-session package*, https://www.npmjs.com/package/express-session, Accessed: 2020-01-20.

[38] *NPM bcrypt package*, https://www.npmjs.com/package/bcrypt, Accessed: 2020-01-20.

[39] *NPM nodemailer package*, https://www.npmjs.com/package/nodemailer, Accessed: 2020-01-20.

[40] *NPM mongodb package*, https://www.npmjs.com/package/mongodb, Accessed: 2020-01-20.

[41] *Nodejs http*, https://nodejs.org/api/http.html, Accessed: 2020-01-20.

[42] *Nodejs File System*, https://nodejs.org/api/fs.html, Accessed: 2020-01-20.