DEGREE PROJECT IN THE FIELD OF TECHNOLOGY
INDUSTRIAL ENGINEERING AND MANAGEMENT
AND THE MAIN FIELD OF STUDY
COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
*STOCKHOLM, SWEDEN 2016*

# Evaluation of webscraping tools for creating an embedded webwrapper

**JOACIM OLOFSSON ERIKSSON**

**KTH ROYAL INSTITUTE OF TECHNOLOGY**
**SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION**


KTH Computer Science
and Communication

# Evaluation of webscraping tools for creating an embedded webwrapper

JOACIM OLOFSSON ERIKSSON

Master's Thesis at CSC
Supervisor: Jeanette Hellgren
Examiner: Anders Lansner

# Abstract

This report aims to evaluate three different tools for web data extraction in Java for the company Top of Europe. The tools used in the evaluation was jArvest, Jaunt and Selenium WebDriver. Through a case implementation which wrapped parts of a specific web application, web document data was to be automatically identified, extracted and structured. By using the results of the case implementation, the tools was contrasted and evaluated.

The results discovered jArvest as non-functioning while the other alternatives provided similar performance but also offering somewhat different strengths. Jaunt provides a good interface to the HTTP protocol and has bigger possibilities for wrapping DOM elements while Selenium WebDriver supports JavaScript, AJAX and some graphical interface aspects.

# Referat

## Utvärdering av verktyg ämnade för webscraping vid skapandet av en webwrapper

Denna rapport ämnar att utvärdera tre olika verktyg för att extrahera data från webben i Java åt företaget Top of Europe. Verktygen utvalda för denna utvärdering var jArvest, Jaunt och Selenium WebDriver. Genom en case implementation ämnad att 'web-wrappa' en specifik web applikation skulle web dokument data automatiskt identifieras, extraheras samt struktureras. Resultatet av detta case skapade sedan data för att kontrastera verktygen i en utvärdering.

Resultaten visade att jArvest ej gick att tillämpa i detta fall medan de andra delade många likheter men även hade olika styrkor. Jaunt ger djupare tillgång till HTTP protokollet och har större möjlighet att wrappa DOM element medan Selenium erbjuder JavaScript, AJAX och viss exponering av grafiska attribut.

# Contents

# 1. Introduction

The web has grown immensely during recent years. A substantial part of this growth can be accredited to the introduction of Web 2.0 which has lowered the technical barriers for publishing online material to become reduced to a degree that technical knowledge isn't substantially necessary and publishing can be done by average users [1]. This progress has probably not escaped anyone and is to a large degree a positive one but it does have some implications also. Vast amounts of information are being conceived and published on the web but all too often it is produced solely with human readers as prospective consumers only. There are several reasons why this data is of interest within the fields of business- and competitive intelligence [2], social networking [3], scientific purposes [4], etc.

Web data extraction systems aim to efficiently collect these kinds of web data by the use of automated systems and limited human effort. As opposed to accessing structured data through the technologies proposed by the Semantic Web [5] and Linked Data [6] initiatives such as RDF, OWL, SKOS and SPARQL, semi- and unstructured data is usually requested as plain text, XML, HTML or XHTML which then has to be parsed in order to extract the parts which are of interest. This process is called web scraping or web data extraction. Simple rudimentary methods is commonly used for web scraping while more advanced tools also exist but may require technological knowledge in several areas.

A request or query for web data usually delivers already structured data in templated HTML or similar format for the end-user in a nice readable way which isn't optimal for automated retrieval. In order to convert this data back into a structured format, a web wrapper can be used. A web wrapper is defined as a procedure that seeks and finds data required by a human user, extracts the data from semi- or unstructured web resources and transforms them into structured data by using semi-automatic or fully automatic methods. Thus, a web wrapper must be able to communicate with the `HTTP` protocol using standard verbs in order to request and retrieve web documents. It then has to be able to crawl through web API's and documents in order to find the data of interest which it consumes and outputs the restructured data in a relational form.

In this thesis the main goal is to evaluate three Java tools that provides functionality and aid to accomplish web data extraction. The investigation is based on implementing a particular web wrapper to extract data in the deep web with the aid of the different web scraping tools.

## 1.1. Objectives and aim

The web is comprised by numerous technologies and standards for publishing and delivering content. Most of the content is served for human interaction and consumption without bothering to provide service for automated retrieval and external integration of the data. Interacting with content is therefore straightforward for humans navigating with a web browser, but it becomes a daunting task for automated solutions to reach and retrieve this kind of data.

The purpose of this investigation is to evaluate different tools for implementing a web wrapper within the bounds of a specific application and its requirements. This may provide insights into convenient pragmatic solutions and consequences arising from these. In other words, based on a case study, this thesis aims to investigate, clarify and establish what constraints and strengths three different tools for web scraping provide for implementing a web wrapper.

### 1.1.1. Problem Definition

The main research question in this thesis is defined as "What is the best solution for implementing a embedded web wrapper in the Play framework considering obtained functionality, solution complexity and ease of implementation. ".

Implementing a web wrapper refers to constructing an embedded application for retrieving, extracting and structuring data from web sources which is then made available to the main web application within the Play framework. Obtained functionality, solution complexity and ease of implementation are the main criteria for evaluating the different tools used and provides a basis for contrasting them.

There are a vast plethora of tools that could be used in order to develop working solutions but in order to streamline the thesis work, a predefined selection of tools were chosen by the company.

### 1.1.2. Delimitations

The extraction of data in this thesis relates to finding a specific set of data for using in this case study and the generality of these could be questioned. The data of interest is located within the deep web as opposed to the surface web which can be accessed in a more straightforward manner equivalently to a web spider or indexer.

As this is a case study with the purpose of examining tools for scraping web content which is to be used within the bounds of an existing application, preliminary analysis of a solution which indicates that the underlying structure of the application has to be severly restructured will be deemed unsatisfactory and thus, be discarded.

### 1.1.3. Motivation

Choosing between programming tools and frameworks can in general be very hard because of poorly updated documentation or no documentation at all and the initial lack of understanding ones actual developer needs. When choosing to use a technical

solution it may also lead to a situation where migrating to a different one will be very costly. In this thesis the aim is to address this by differentiating web scraping tools and to illustrate what sets them apart. This could also prove helpful for developers of web scraping frameworks because it raises their understanding of usage patterns and situations.

The process of gathering unstructured data on the web is an interesting area within many contexts whether it be for business, scientific or personal usage. The advertisement business rely on directed advertisement which are distributed throughout several pages, for the service to understand its current context, web data extraction and web wrappers can be used in conjunction with content analysis tools for contextual analyses of the current page. In science, data sets are shared and used by several researchers and often publicly publicized. In some cases the data sets are provided through a structured API, but often data is only accessible through search forms and HTML documents which calls for web wrapping methodologies to be used. Personal use has also grown as services have started to emerge which provides users with tools to mashup components from different web pages into own collection web pages.

## 1.2. General implementation context

To provide contextual information to the thesis investigation, this section provides some brief information on the application in which the test cases were executed. The project was undertaken at a Swedish company in which the final implemented solution will also be used in production.

### 1.2.1. Top of Europe

The company, Top of Europe, in which the thesis work is performed at, deals with managing and administrating horse training data for customers, mainly horse owners and trainers but also other stakeholders.

### 1.2.2. The web application

The application platform context for which the case was subjected is a web application that manages training related data. In this web application, customers create their own hierarchy with users and training subjects in which they manage training related data. The main idea of this application is to provide customers with a system that can be easily accessed in ubiquitous ways for the whole customer company and provide effective management tools while providing automated information for outside stakeholders as well.

### 1.2.3. Application specifications

The application is built on the Play! Framework which is a relatively young and complex web application framework for Java. The application is deployed in the Amazon cloud services and uses relational databases. The application follows the architectural standards for REST web services and is thus, RESTful.

## 1.3. List of abbreviations

- **AJAX** - Asynchronous JavaScript and XML
- **API** - Application programming interface
- **CSS** - Cascading style sheets
- **DOM** - Document object model
- **HTML** - Hypertext markup language
- **HTTP** - Hypertext transfer protocol
- **HTTPS** - Hypertext transfer protocol secure
- **IDE** - Integrated development environment
- **OWL** - Web ontology language
- **RDF** - Resource description framework
- **REST** - Representational state transfer
- **SKOS** - Simple knowledge organization system
- **SPARQL** - Simple protocol and RDF query language
- **SSL** - Secure sockets layer
- **URI** - Uniform resource identifier
- **URL** - Uniform resource locator
- **W3C** - World wide web consortium
- **XHTML** - Extensible hypertext markup language
- **XPath** - XML path language

# 2. Background

## 2.1. Web data extraction and web scraping

### Web scraping

Scraping refers to capturing semi- or unstructured data and parsing it in order to structure it in some way. The most obvious context is screen-scraping done by techniques such as copy-paste keyboard hotkeys which is a powerful tool strengthened by its simplicity. Web scraping, similarly as screen scraping refers to extracting data from a web document and has been used since the dawn of the web in different scenarios and approaches to gather information more or less automatically in order to retrieve and parse interesting data.

A typical scenario is web search engines using scraping robots to index the web sites that will be available for searching. Another context of use is scraping search results or news results from a web document which requires the scraping algorithm to recognize and delimit objects within the DOM of the document. As stated in the article by Lavergne et al. [7], web scraping is also used to provide fake web services with data aimed at biasing search index services or spam mail robots.

The web scraping process generally targets unstructured HTML and XHTML documents which in its simplest form can be retrieved by using a HTTP request for the specific document URL. Simple GET or POST HTTP requests can be easily completed by the use of `HttpURLConnection` in Java base libraries or using the Apache HttpComponents `HttpCore` and `HttpClient`. As the document is retrieved to memory it can be exposed to analysis and data can be identified and extracted. Web documents generally contain markup tags that provide publishing and layout information for web browsers. This information can be more or less complex and prove helpful to the web scraping process in that it may provide attributes that help in identifying interesting parts. Contained within the text nodes of the element nodes are the actual data or text that is to be extracted.

According to Glez-Pena et al. [8], web scraping approaches can be divided into three different categories which are libraries, frameworks and desktop-based environments. Libraries are general libraries which can be used as building blocks

to assemble all the necessary functions for handling HTTP requests and responses or parsing the documents in different ways. The second category, frameworks provides more complete and focused functionality for scraping which are often based on defining robots and their behaviours. Desktop-based environments aim to attend needs of users with a low technical ability and uses graphical environments in order to simplify the web scraping process which also limits the usage potential. The desktop-based environments cannot be embedded within other applications and are solely used to extract information in a more manual fashion.

#### Web data extraction

Baumgartner et al. [9] defines a web data extraction system as an automated software that repeatedly extracts data from web pages and delivers this data to a database or another application. This implies that an extraction system is automated, transforms data and makes use of the extracted data. Similar to web scraping, web data extraction aims to scrape data but in a more automated fashion. Several approaches exist mainly focused on how to preserve object identification within the DOM as web sites are updated and how to efficiently remove scripted web templates in order to restructure semi-structured data.

However, the field is not limited to these issues which is noted by Baumgartner et al. [10] that states that the actual real-life extraction from a document is only "half the game" since much of the extraction data is buried in the deep web. This implies completely different challenges as a web data extraction system needs to be able to navigate the web as well as identifying and scraping the data. In order to navigate a web site and scrape data in the deep web, it is not feasible to assume the API structure of the web site to be static which means that a web extraction system needs to be able to crawl by using information in documents (such as hyperlinks or forms) rather than accessing just static URL paths. Web 2.0 techniques and dynamic HTML makes this even more complex as just requesting a web document resource is not enough since there may be data that is requested by JavaScript over AJAX which provides dynamic changes in a document. Also, in a RESTful API, URL's commonly address elements while HTTP verbs provide ways to deal with them which also becomes an important aspect of how to access data.

## 2.2. Web wrappers

A wrapper is in general considered to be a procedure that aims to convert semi- or unstructured data into structured data. In a web data extraction context, a web wrapper is a program or procedure that finds and extracts content from web sources and converts it into a relational form [11]. Plenty of web sources keep structured data which are already quantified in relational form but still only publish them in a semi- structured fashion such as HTML markup by the use of dynamic web applications and templates. The aim of the web wrapper is thus, to re-structure this

data to its structured form.

### 2.2.1. Web wrapper life-cycle

Web wrappers are typically constituted by a generation, execution and maintenance life-cycle. In the generation phase, the wrapper is defined and in the execution phase, the wrapper extracts data. The dynamic nature of web applications and sources raises the probability for a wrapper to stop working correctly which induces the need for adjustments and maintenance, in most cases manually but preferably automatic.

#### Wrapper generation and execution

According to the exhaustive survey done by Ferrara et al. [12], four different approaches exist to generating a wrapper - regular expressions-, logic-, tree- and machine learning- based.

Regular expressions technique is the most commonly used. These kinds of wrappers identifies strings or patterns as matching criteria. Using this approach on semi-structured HTML documents often rely on word boundaries, HTML tags, table structure or another criteria based on syntax rules which are well defined. This approach is thus similar to a document text search but regular expressions contain a wider set of rules.

Logioc based approaches are based on web specific wrapper programming languages. These approaches considers the web document not simply as fluid text but rather has the basic assumption that a web document is a semi- structured tree document. As in the case with HTML or XHTML, the DOM represents its structure with comment nodes, element nodes and text nodes which characterizes the document properties and content. This kind of wrapper draws advantage of being able to exploit the combination of semi- structured DOM properties and the document contents while also offering programmatic logic support.

Tree based approaches aims to fragment the document tree and isolate objects for extraction. An example is the partial tree alignment [13] which realizes that data extraction is often carried out on contiguous regions of a document, termed data record regions. The objective is then to identify and extract these regions. The algorithm segments the document based on both the DOM and also graphical features in order to find gaps between data records. The segmentation is followed by a partial tree alignment algorithm which breaks the data record DOM fragment from its sub-tree which becomes the new root. This approach take no regard to data

contents but rather exclusively makes judgements on tree structure and tag content.

Machine learning approach relies on training models to find domain specific data. It is common practice for web sources to utilize templated document publishing models. In some cases these differ or evolve and a machine learning system should be capable of learning how to extract contents in these contexts. The learning phase is often comprised of selecting a multitude of positive and negative examples which will provide data for the system to build a object identification grammar. This implies that some graphical support is necessary for a administrator of such a system.

#### Wrapper maintenance

Regardless of the technique adopted in the generation or execution of a web wrapper, it is highly likely that maintaining the web wrapper is also necessary to secure the robustness of the implementation. Web services and sources tend to rapidly develop and is subjected to constant change. Even though a web wrapper handles dynamically templated documents, the templates themselves might evolve and also the web API of the web application rendering the web wrapper useless. Depending on the wrapper algorithm used and the state of the web data being extracted, it is highly likely that the models or algorithms will require updates. Historically this has been done exclusively by manual labour. Meng et al. [14] was among the first to propose an automated schema based maintenance solution which was based on empirical observations. These empirical observations claimed that hyperlinks are seldom removed, annotations, semantic metadata is often preserved as well as some syntactic features such as characteristics of data items, data patterns and string lengths. Other automated approaches has surfaced in recent years as well, such as proposed by Ferrera et al. [15] which relies on structural similarities between versions of a web document.

### 2.2.2. Crawling for data in the deep web

Retrieving statically designed web pages can be considered trivial since they are served in a static fashion and are simply accessed with one particular URL. However, these kinds of web designs are becoming more uncommon and web designs have been moving towards a more templated structure driven by a more dynamic and complex logic such as returning dynamic templated web pages which are assembled from a relational database depending on what is requested and where it is requested from.

To find the data in the deep web[1], different approaches can be used. If an analysis of the web site API provides a simple enough way to access the document directly

[1]The deep web is further defined in section 2.8

and data is not requested by schemes which are specific for the web sites database or back end. In these cases the data might be accessed by just requesting the item URL for instance `http://www.server.com/path/to/query` but in most cases elements are requested by its id `http://www.server.com/path/to/id`. Provided that id's are just the identifier scheme used in the server database, a extraction system would not know these. In most cases, web extraction systems have to query a form for the element which implies the need of supporting browser like features such as forms or hyperlinks and also more elaborated HTTP requests.

### 2.2.3. Wrapper robustness

The fact that many web sites are script generated using templates, their documents will share a high degree of common HTML DOM tree structures. This provides wrappers with a increased possibility to extract information effectively. However, most web sites constantly evolve implying that the document templates change which renders the wrapper out of order. Dalvi et al.[16] defines wrapper robustness as the probability that wrapper will work on a future snapshot and bases this on evaluating the probability for a wrappers historic success to persevere changes in a web site.

## 2.3. Document object model

The document object model (DOM) is a platform and language independent convention to represent a document with node objects in a tree [17]. As stated in the earlier section, example formats realizing the DOM model is HTML, XHTML and XML. A document becomes a rooted tree and tags are `document nodes` and `element nodes` which contains `text nodes` embedding the data. To examplify the DOM, a HTML web document could be viewed as a tree built from all the embedded markup tags. The nested nodes will then become the trees branches.

## 2.4. Xpath

XML path language provides a syntax to address specific elements of an XML document. The definition specifications have been developed by the W3C consortium [18]. XPath can be used to identify a single element in a document tree or multiple occurrences of the same element. A XPath expression is strictly related to a specific document tree which is a major weakness and reduces its flexibility. XPath 2.0 [19] aims to address this issue by introducing relative path expressions. Minor changes to a DOM tree might render XPath expressions useless.

## 2.5. CSS selectors

Cascading style sheet (CSS) selectors [20] are patterns that match against the DOM tree. CSS selectors are similar to XPath expressions but they are more widely adopted by browsers and provide a simpler syntax.

## 2.6. Structured, unstructured and semi-structured data on the web

Structured data on the web is data that is immediately accessible through a clearly defined API. The distributional properties of this data is described by the Semantic Web and Linked Data initiatives which main objective is to provide methodology for the sole purpose of standardizing data. Protocols for distribution of structured data has been available for some time and several standards exist such as RDF, OWS, SKOS, etc. Structured data is quantified and tagged in order to simplify the API interface between applications.

Semi-structured data does contain some partitioning in that it may be a table or in some other way partitioned in the DOM. This thesis makes a distinction between unstructured and semi-structured data because it may be relevant for the discussion and also help the reader for clarity. These kinds of data types are the main data components that form HTML and XHTML documents throughout the web and in the case of these formats, the main target is providing publishing information for browsers to enhance the readability of the embedded data.

Unstructured data is comprised of fluid text and is not structured in any way. It does not contain any tags or delimiters of any sort.

## 2.7. The semantic web

The semantic web [5] initiative aims to standardize data formats and documents in order to make it available for linkages on internet to form a web of data that can be automatically served between computers. Plenty of standardized formats exist for this cause such as RDF, Micro, JSON, linked data etc. These object formats offer possibilities to package data in a structured fashion.

## 2.8. Deep web

The part of the web that is indexed by search engines is termed the surface web, as opposed to the directly available surface web there is the deep web. The deep web is the underlying resources which are very hard for search engines to index because it

requires some further critera to be accessed. The deep web is estimated to be sized hundreds of times greater than the surface web and also provide the most quality content [21]. To index deep web data - two different approaches can be taken, the first is called federated search [22] which aims to use user-queries to gather deep web data and the second, surfacing which aims to exhaustively produce submissions for deep web forms.

## 2.9. Frameworks vs. Libraries

Using a programmatic library or a framework does imply some differences within this thesis. A library requires the host application to be in control and to call the necessary methods throughout the progression of the process. In a framework, the process is different in that the process has to be predefined and the main application signals a start and the framework calls the hosting application when it is done instead, a design pattern which is called the inversion of control.

## 2.10. Motivation of web-scraping technology choice

There a are several web scraping tools available throughout the web. The reason for choosing the specific ones selected for this evaluation is based on choices of the constituent company and the availability of the tools. Many of the frameworks and libraries available are outdated and have not been updated for several years which helped in narrowing down the constituent selection.

## 2.11. jARVEST Framework

jARVEST is a web harvesting and scraping framework development at the University of Vigo [23]. jArvest uses its own jRuby based domain specific language (DSL) to define harvesting robots which are independent and external artefacts. The robots can then be executed onto a web site to extract information.

The robots are assembled through a set of transformers. There are several different transformers available for construction of robots. For instance, `wget` and `post` is used to do a HTTP requests for a predefined URL and `merge` collapses input from several transformers. A transformer takes an input string and in some cases certain criterias and then outputs a resulting strings which may be passed on to another transformer.

Transformers can be executed in a pipelined serial or parallel fashion and there is some support for procedural instructions such as loops and variables. jArvest also implements requests for AJAX servers. The input strings are in our case HTML data in which the transformers can be instructed to extract or act upon certain

parts defined through the DOM tree with the aid of XPath or CSS selectors.

## 2.12. Jaunt

Jaunt is developed for web application unit testing, web scraping and automation. The main feature in Jaunt is that it is a headless browser which provides possibilities to perform browser-level, document-level and DOM-level operations and has a very small footprint. [24]

The Jaunt library acts like a browser and provides functionality which is similar to a browser but through API calls contrary to human interaction. The feature set is comprehensive and the library parses HTML, XHTML and XML even if broken into DOM trees. Jaunt is programmatically controlled for performing several browser related tasks such as posting forms and following links as well as parsing retrieved documents into a DOM tree for traversal. There is no support for JavaScript or Ajax.

## 2.13. Selenium WebDriver

Selenium WebDriver is a comprehensive solution developed since 2004 mainly for unit testing web applications [25]. Selenium acts through automating almost any web browser such as Internet Explorer, Firefox etc. or work by itself as a headless browser by implementing HtmlUnit [26]. Ajax and Javascript are fully supported when automating web browsers which relies on the browsers implementation of these. The headless browser uses its own implementation of JavaScript which is an emulation of the implementation from FireFox 3.6.

Controlling Selenium can be done either by recording usage patterns within a browser or programmatically driving it by using its library methods. There is a wide feature set for requesting documents over HTTP/HTTPS protocol. Selenium has support for all common document operations such as DOM, XPath, CSS selectors. Forms can be filled out and processed in a web browser related fashion.

## 2.14. The case

The case deals with gathering two different data sets within the deep web from the organization Svensk Travsports web application located at `www.travsport.se`. Svensk Travsport is a Swedish organization that works as a service provider for trotting-enthusiasts, practitioners and other stakeholders. They manage a large database containing various information connected to trotting such as trainers, horses and race tracks. The data to be extracted have several different parts and are located in different web resources. The main idea is to produce a web wrapper that structures the information automatically. A logic based approach will be used for building the web wrapper in each of the different web scraping tools and the approach will be kept as similar as possible with the exception of trying to create the best possible solution based on what the tool has to offer. The reason for not using a tree based- or machine learning approach is that the tools themselves does not hold any logic to support these kind of designs and also that it was rejected by the company as these implementations would be harder to maintain and develop further.

### 2.14.1. Web wrapper steps

The approach can be divided into 5 steps corresponding to each of the web documents that has to be retrieved from the web site service. Since the API of the web application uses id's to retrieve documents about entities, the web wrapper has to use the same methodology as a real human user and navigate through the website contrary to retrieving the specific information instantly.

#### Step 1

The session is initiated by retrieving the main index document which is retrievable through the base URL. The main search function form is highlighted in figure 2.1 with a red box. The first step is to fill this form with the trainers name and submit. In the document, this form is always located at the top and the input carries a `searchString` meta tag. The result from posting this form is a redirect with status code 302 for another web page. The search query posted by a user will be stored server-side and the session is prevailed by using session cookies.

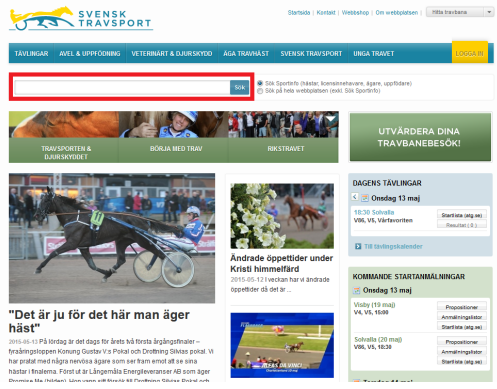


Figure 2.1: Initial document.

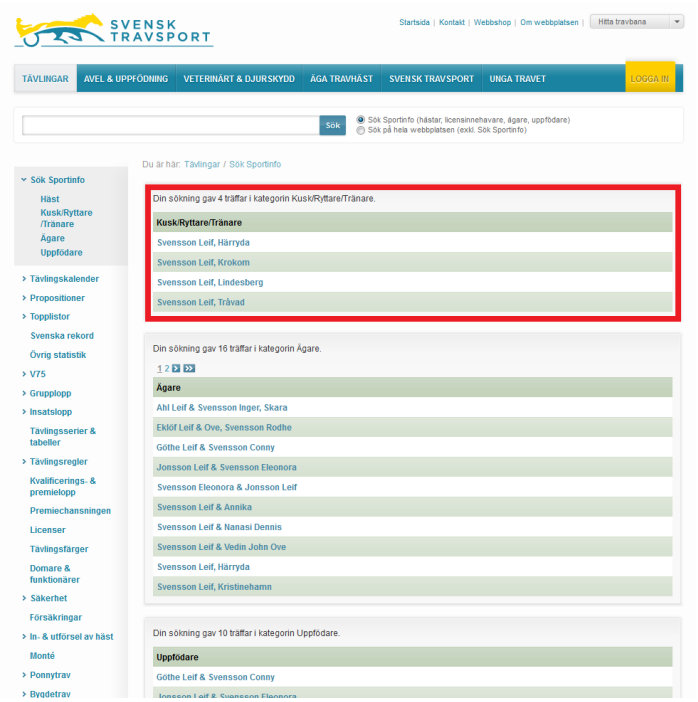


Figure 2.2: Search results.

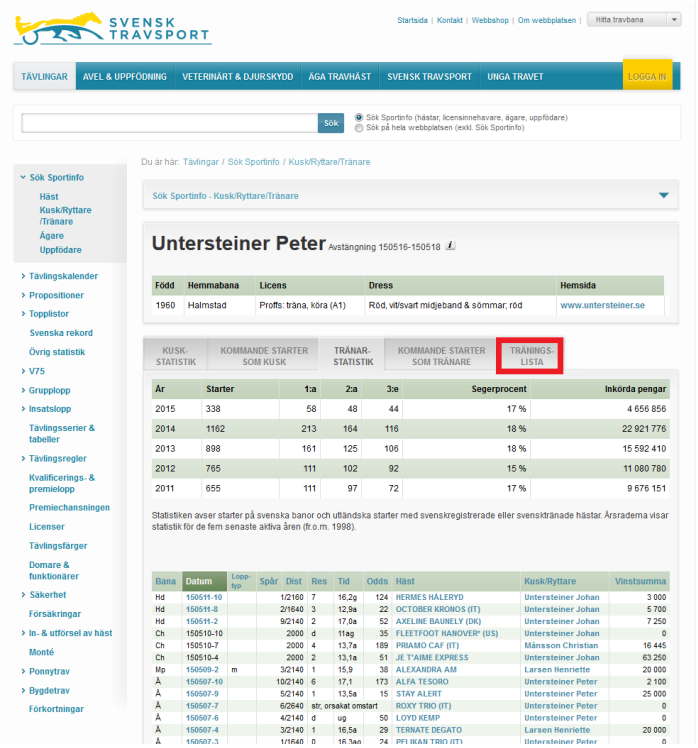


Figure 2.3: Trainer page.

### Step 2

In the second step the correct table containing the hyperlink for the sought trainer has to be identified. This document might contain several tables and is dynamic since it is based on the search query posted in step one. The tables does not contain any relevant meta data to help in separating the correct one for identification which entails that identification has to be done by examining the table headings. Visual structure of the website is depicted in figure 2.2 with the correct table marked in red.

### Step 3

Highlighted in figure 2.3 is the button that links to a trainers list of active horses currently in training. This is a hyperlink leading to a new document rather than being a tab for a tabbed table which it appears to be visually. Following this hyperlink leads to the view in the next step.

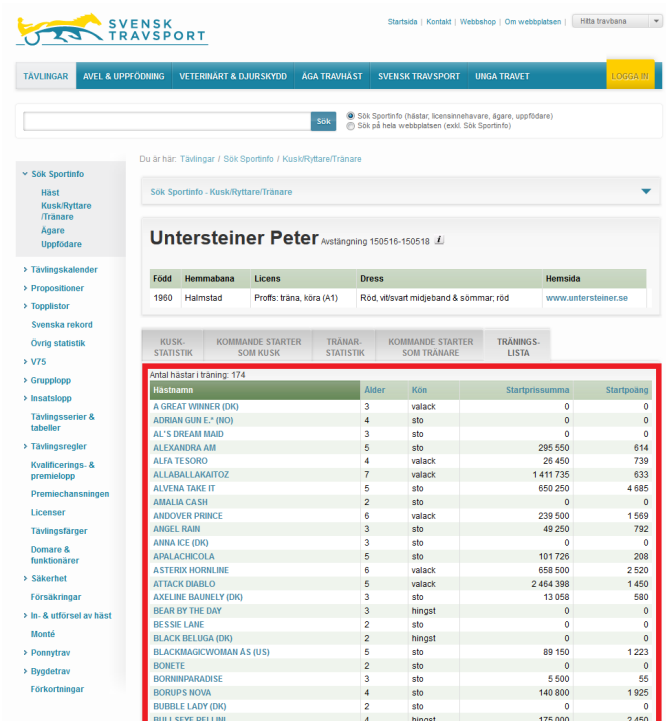


Figure 2.4: Trainer horses.

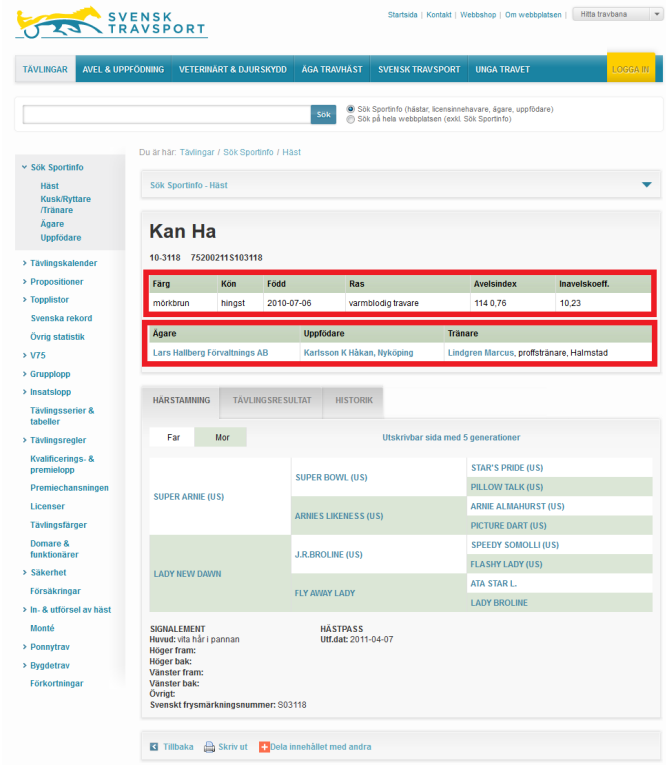


Figure 2.5: Horse individual page.

### Step 4

Figure 2.4 shows the first semi-structured table that is to be converted into structured data. It contains the links to each of the individual horse documents which has to be extracted.

### Step 5

The final step is to scrape the two tables highlighted in figure 2.5. In a similar fashion as earlier tables, these tables do not make use of any metadata to aid identification.

The procedure for gathering data is illustrated in the flowchart figure 2.6 below.

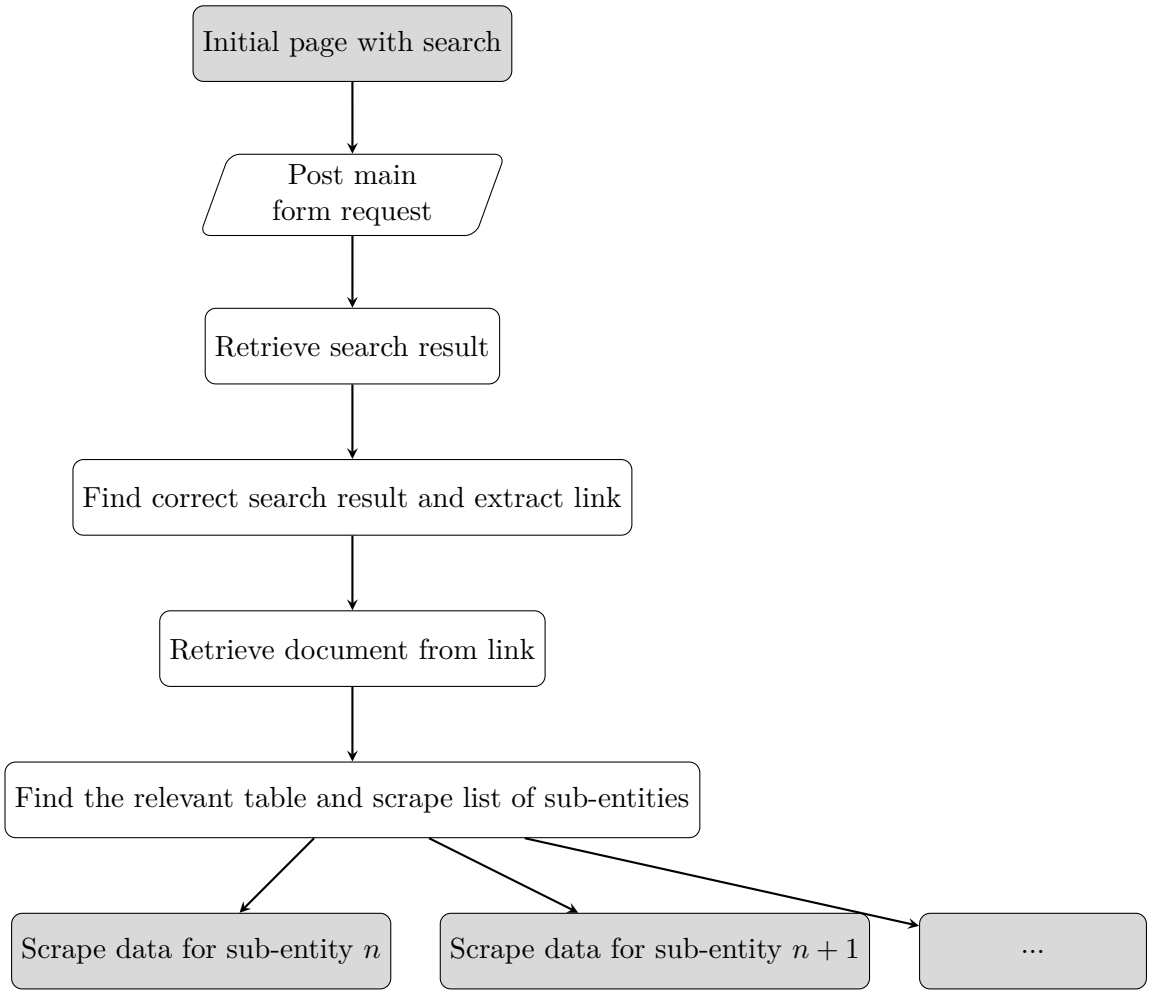


Figure 2.6: Flowchart diagram of the web wrapper

# 3. Method

The investigation in this report is based on a particular case solution implemented in three different web scraping tools. The context of the implementation is the Java language and the Play Framework which is a web application framework aimed at building complex RESTful web applications. By analyzing the different tools for the automated extraction of data from web services through the implementation of this particular case, the report aims to evaluate the data extraction possibilities available within the different tools. The case is further elaborated in the background chapter.

## 3.1. Selection

The selection of development tools were predefined by Top of Europe. Some of the proposed tools were discarded as being outdated and poorly updated which left the three tools discussed in the background chapter. The remaining three tools is regarded as having a good availability, updating policy and compability with Java and the Play framework. Availability is judged from a point of view that the tools has to be either free or provide a fully functional demonstration version. What is implied by updating policy is that the tool still has to be developed and not be obviously abandoned. These selection criteria fits the three selected tools but still, many other tools are available even though they might not be developed specifically for web scraping.

## 3.2. Evaluation

The selected web scraping tools are to be evaluated, as specified in the problem definition "In terms of obtained functionality, solution complexity and ease of implementation". The evaluation will provide the study with qualitative data in the predefined criteria. Quantitative data such as benchmarking is in this case hard to establish and is considered rather pointless. Partly because the used libraries and frameworks differ too vastly in their implementation methods that they can not be considered to be equal in how they approach the case implementation, but also due to the developers environment and subjectivity.

The first and most important factor is examining obtained functionality and how well the tools can handle the vast plethora of technology available on the web for retrieving and restructuring published web data. Taking into account that much of the data available is buried in the deep web and that this kind of data will require both fundamental HTTP support as well as HTML functionality but also might need more complex functionality such as support for asynchronous updates through AJAX.

The second criteria is solution complexity and ease of implementation. In the particular context of this implementation it is discussed with aspects depending on developer environments and developer preferences which may be subjective to some degree. Solution complexity is a loosely defined criteria which is harder to establish but could for instance include logic issues in a solution.

As mentioned earlier, some of these criteria could be quantified and measured simply in for instance number of functions and implementation time but that does not provide the reader with any real understanding on how these tools really compare. Quantified measurements can prove misleading and misses the complexity of the issues facing web data extraction. In order to address these issues, qualitative reasoning and discussion provide better insight.

The evaluation criteria is described and organized as:

- Obtained functionality
  - Basic HTTP protocol verb request API
  - HTTP header exposure
  - HTTPS support
  - General DOM identification
  - Other DOM identification methods
  - DOM search traversal methods
  - Wrappers for DOM elements
  - Form handling
  - Relative URL path support
  - DOM HTML exposure
  - Parse HTML special characters
  - Broken HTML support
  - Graphical UI data
- Solution complexity and ease of implementation
  - IDE support
  - Additional dependencies
  - Compability issues with Play Framework
  - Stability issues
  - General ease of implementation
  - Solution complexity

## 3.3. Method motivation

As mentioned earlier in the introductory method part - this study will provide qualitative data with the intent to clarify differences and ease of implementation regarding the selected web-scraping tools for java. Quantitative measurements could have been used to provide solid data for comparison of the different tools. However, using quantitative measures like for instance time in minutes it takes to create a solution misses the complexities in working with the different tools.

To do this kind of evaluation based on a case implementation has drawbacks and benefits. The main benefits are that the case is implemented into a live online application which sets the boundaries for what is possible to implement and stability is important which is an indication that cutting edge technology may not

be a relevant choice which is also the case for strongly customized solutions that may require heavy changes in the web application codebase. Drawbacks are the generality of the results, while being able to provide relevant results for the case implementation, the results can not be interpreted as general.

# 4. Result

This chapter presents the implementation results and the evaluation results. The criteria for the evaluation is described in the method chapter.

## 4.1. Implementation results

This section contains parts of the web wrapper implementations for each of the web scraping tools.

### 4.1.1. Approach

In this case the target is to scrape background information about horses which are available on the web application at `https://www.travsport.se`. The background information will be used to provide structured trainer data in an automated fashion. This implies that as a trainer information is requested, the scraping solution has to do a thorough lookup into the information required to fill all attributes within each of the contained objects. In order to be successful, it is not enough to do an analysis of the REST API of the site because references are done by id's rather than names which requires a solution to work its way through web page forms and hyperlinks in order to search for the relevant data.

### 4.1.2. jArvest Framework

jArvest is controlled by defining robots in a JRuby based DSL. An implication from this is that a hosting application cannot directly call jArvest methods but rather has to define a complete external robot with the dedicated DSL. jArvest runs on its own thread and executes the defined robot autonomously and returns the result to the host.

The methods provided are termed transformers, they receive streams of strings and output resulting streams of strings. In addition, several of the transformers have more parameters as well. The input strings are the HTML documents retrieved at a given source. The transformers can be cascaded as well as set to branch in parallell.

The session is initiated by creating a jArvest thread which is handed the robot definition as a string object.

**Listing 1: Initiating the jArvest framework**

```
String sSite = "https://www.travsport.se/";
String sResource =
    "?12-1.IFormSubmitListener-searchForm-searchForm";
String sPostQuery=  "sportinfo-search-form_hf_0="+
                    "&searchString="+sTrainer+
                    "&radioGroup=radio5";
Jarvest jarvest = new Jarvest();
String[] sResultArray =
    jarvest.exec("post(:URL=>'"+sSite+sResource+"',
    :queryString=>'"+sPostQuery+"',
    :outputHTTPOutputs=>'true') | wget()");
```

The POST request is sent to the server and the server responds with a 302 redirect to another resource. jArvest was unable to comply with the redirect, and so jArvest was manually directed towards the resource from the server response. However, the response was still not correct for the requested search critera. The case requires the harvester (or scraper) to successfully implement session cookies in order for the server to identify the user throughout the session.

This is not an unusual scenario for web based services to handle session identification. In the jArvest documentation it is clearly stated that jArvest indeed handles cookies and supports cookie based sessions. However, this implementation did not succeed in utilizing that. jArvest did also not automatically redirect upon server request and even though explicitly programmatically redirecting the jArvester agent to load the end-search resource did not prevail the session cookie. This resulted in not being able to continue the investigation further with the jArvest framework.

### 4.1.3. Jaunt

Jaunt provides a rudimentary headless browser. The browser can crawl web-sites much like a regular browser and incorporates many browsing like features which resemble the usage patterns of a human user such as form handling and following hyperlinks. As the browser is directed to a URL, it will fetch the page and automatically parse the DOM tree which is made available for searching and editing through the `Document` class. Wrappers for general DOM objects are provided by the `Element` and `Elements` types which features a set of methods to further search the encapsulated object or expose HTML and text nodes.

A session starts with the creation of a `UserAgent()` which is the headless browser. The UserAgent is then pointed to a web document and in this case we know from analyzing the actual website that the initial search is to be done from the first form in the page.

**Listing 2: Initializing and posting a form in a headless browsing session**

```
1 UserAgent userAgent=new UserAgent();
2 userAgent.settings.autoRedirect=true;
3 userAgent.visit("https://www.travsport.se");
4 Form form = userAgent.doc.getForm(0);
5 form.setTextField("searchString",trainer);
6 form.submit();
```

Selecting a form object in a document is not limited to form numbering used in the example and Jaunt provides several other specific helpers for this task. `Document.getForm(Element element)` casts a selected `Element` into a `Form` object and `Document.getForm(String elementQuery)` identifies a form through a Jaunt feature called elementQueries. A element query provides an interface to use regular expressions to search for object text, attributes and values. Further, `Document.getFormByButton(String)` and `Document.getFormByButtons(String[])` selects a form based on a regular expression search for contained button or buttons while `Document.getForms()` returns a `List` object with all the forms contained in the retrieved document.

A regular HTTP request using the GET verb is sent when using the `visit()` function. Jaunt provides a simple API for using other verbs as well as overriding the standard request HTTP headers. Examples of this is `sendGET()`, `sendPOST()`, `sendHEAD()`, `sendPUT()` and `sendDELETE()`, each which can be completed with remapping of request header values such as `sendGET(URL, "user-agent:Mozilla/4.0")` in which the server will interpret the client as using Firefox instead of Jaunt.

In the retrieved document, the form submission will not lead the client to the result page but rather needs to be redirected from a mid-resource which saves the search string on the server. The headless browser will then follow the `HTTP 302` redirect response from the server in a similar fashion as a regular web browser. In order to preserve the search criteria between the client and server, session cookies needs to be implemented in the client browser and in Jaunt, cookies are handled automatically in order to preserve the session. When the next command is executed, the redirected page will have been loaded into the headless browser. This can be confirmed by the following code:

**Listing 3: Examine current web location**

```
1 String browserLocation = userAgent.getLocation();
```

Provided that the location of the browser is the correct one and that a HTTP status code 200 was delivered which can be caught by a `ResponseException` followed by investigating the exception response `getResponse().getStatus()` method, the browser will have retrieved the result page for the query successfully.

At this point we need to identify our search criteria from the newly retrieved page in order to find the hyperlink that provides the path to the trainers HTML document. The problem is that the web application might provide us with several tables containing hits from different areas which are not only trainers but also owners and others. Visually, each of the tables are named in their headings but structurally these tables do not contain any kind of metadata for identifying them in the DOM. The only way to identify them is to analyze each table heading row followed by searching the rows or to extract all the document hyperlinks globally and match them to our search criteria. In the latter case, we will not get any clue as to what table the hyperlink was retrieved from which may lead the browser astray. In the first case we have to extract all the tables whereby we identify the correct one and search the rows for the correct hyperlink. In order to simplify extraction of tables, a general helper function was introduced. Tables contain semi-structured information and the function is designed to map headings as instructed by the caller of the function.

**Listing 4: Identifying a table given some headers**

```
1  public static Table findTable(UserAgent ua,List<String>
       headerNames) throws JauntException {
2      //Uses Java regex
3      String pattern="(?s)^";
4      for (String s: headerNames)
       pattern+="(?=.*"+escapeHtml4(s)+")";
5      pattern+=".*$";
6      Pattern p=Pattern.compile(pattern);
7
8      Elements tables=ua.doc.findEvery("<table>");
9
10     for (Element t: tables) {
11         Table table=ua.doc.getTable(t);
12         Matcher m= p.matcher(table.getRow(0).innerText());
13         if (m.matches()) return table;
14     }
15     throw new NotFound("Cannot identify a table by the given
       header names.");
16 }
```

The above code snippet illustrates one way of identifying a table in a document by examining its headers. Each table is cast into a Jaunt table object and the header row is extracted by using `Table.getRow(0)`. In order to match the search criteria against each of the header columns, Jaunt provides a regular expression search in the `findEach(String query)` method that can be executed against the text nodes simultaneously as matching the element nodes which embeds them. Unfortunately this does not work when searching for several child elements simultaneously because the only supported logic is the `OR` operator which means that even though a correct number of header elements is identified, they have to be validated again in order to secure that they are not multiples of one of the criterias. This was solved by using the traditional Java implementation of regular expressions which provides ways of achieving `AND` logic operations. As `Element.innerText()` is used to extract the text nodes of the whole header row, Jaunt concatenates them automatically which provides a text string to match the criteria against. The same methodology can thus be used for other encapsulating elements as well because Jaunt concatenates all sub-text nodes present within an element when using the `Element.innerText()`.

**Listing 5: Extract data columns from a Table object**

```
1  public static List<Map<String, Object>> getTableData(Table
       table,List<String> headerNames) throws JauntException {
2      if (table==null) throw new NotFound("Missing table
       object.");
3      List<Map<String,Object>> resultList=new
       ArrayList<Map<String,Object>>();
4      for (Element e:
       table.getCol(0).toList().subList(1,table.getCol(0).size()))
       resultList.add(new HashMap<String,Object>());
5      for (String header: headerNames) {
6          Iterator<Map<String,Object>>
       resListIterator=resultList.iterator();
7          for (Element e:
       table.getColBelow("(?s).*"+escapeHtml4(header)+".*")) {
8              Map<String, Object> m=resListIterator.next();
9              m.put(header,e);
10         }
11     }
12     return resultList;
13 }
```

Using the previously identified table object, the above code will extract all `Element` objects for the columns selected by header names. The elements will be wrapped into a map that uses the header names as keys. The task of extracting becomes very simplified through the use of the `Table.getColBelow(String headerRegex)` method which allows the use of regular expressions to select a column and returns all the column cells below that specific header. The `Table` object also features methods for selecting by numbering. Further, rows and cells can be addressed with both numbering and regular expressions.

The algorithm stores the elements for each of the table cells because there might be more relevant information in a cell than just its text node, for instance a `HREF` or similar attributes. It is important to note that the element objects stored will become stale(rendered invalid) if the headless browser retrieves a new web document and data has to be extracted and moved into other data structures.

With the table extraction helper, the task of retrieving the correct table from the DOM and then identifying the correct row is simplified:

**Listing 6: Retrieve a correct table row and follow its hyperlink**

```
1  List<Map<String, Object>>
       tableRows=getAllTableRowsDefinedByHeaderNames(userAgent,
       Arrays.asList("Kusk/Ryttare/Tränare"));
2  Element trainerElement=findTrainer(trainer, trainerRows);
3  if (trainerElement!=null) (new
       Hyperlink(trainerElement.findFirst("<a href>"),
       userAgent)).follow();
4
5  public static List<Map<String, Object>>
       getAllTableRowsDefinedByHeaderNames(UserAgent userAgent,
       List<String> headerNames) throws JauntException {
6      Table table = findTable(userAgent, headerNames);
7      List<Map<String,Object>>
       resultListMap=getTableData(table, headerNames);
8      return resultListMap;
9  }
10
11 public static Element findTrainer(String
       targetName,List<Map<String, Object>> table) {
12     for (Map<String, Object> row: table) {
13         for (Map.Entry<String, Object> col: row.entrySet()) {
14             String
       trainerName=(cleanUp(((Element)col.getValue())
       .innerText("",true,true))).split(",")[0];
15             if (trainerName.equals(targetName)) return
       (Element)col.getValue();
16         }
17     }
18     return null;
19 }
```

Finding the search criteria becomes trivial and is for simpleness and enhanced readability put into a new function. If the correct `Element` is found, a Jaunt hyperlink object is instantiated by the `Element`. Finally, the `follow()` method of the hyperlink instance is used which represents a user mouse click in a browser and redirects the headless browser to the linked document.

In the document retrieved at this point, we need to switch the view for a list of actual horses active in training within the trainers stable. In a regular browser, this is done by clicking a button called "TRÄNINGS-LISTA". The only support we get from Jaunt for this is to search the DOM globally for this button and retrieve

its <HREF> tag attribute.

**Listing 7:**

```
1 Elements  tLink=userAgent.doc.findEvery("<a href>");
2 Element res=null;
3 for (Element tl: tLink) {
4     if
      (cleanUp(tl.innerText("",true,true)).equals("TRÄNINGS-LISTA"))
      res=tl;
5 }
6 userAgent.visit(normalizeUrl(res.getAt("href")));
```

The `cleanUp()` function utilized for each of the `String` objects extracted from the <a> text nodes is simply removing erroneous characters induced by the Jaunt document de-encoding. The reason for not being able to simply create a hyperlink instance and using the earlier mentioned `follow()` method is because the web application uses relative URL paths at this point. This can not be normalized by Jaunt which means that a request containing relative paths can not successfully be retrieved and standard Java library URI had to be used in order to normalize the paths in a trivial helper function.

At this point there is actual data which are interesting to scrape. This page contains some specific information on each of the horses and also provides hyperlinks to each of the horses individual pages. The data needed in this case is available in a single semi-structured table that we can use our previously defined table scraping function to structure.

**Listing 8:**

```
1 List<Map<String, Object>> horseTableRows =
      getAllTableRowsDefinedByHeaderNames(userAgent,
      Arrays.asList("Hästnamn", "Ålder", "Kön",
      "Startprissumma", "Startpoäng"));
```

Contained within this table in each row of the column "Hästnamn" is the `Element` which holds the name and link references for each of the individual horses. The last step is to retrieve each of these documents and scrape data from them. As before mentioned step, relative paths are being used which is necessary to normalize before they can be requested.

**Listing 9:**

```
1 for (Map<String,Object> o: horseTableRows) {
2     String
      url=normalizeUrl(((Element)o.get("Hästnamn")).findFirst("a")
      .getAt("href"));
3     userAgent.sendGET(url);
4     List<Map<String, Object>> horsePageData =
      getAllTableRowsDefinedByHeaderNames(userAgent,
      Arrays.asList("Färg", "Född", "Kön", "Ras", "Avelsindex",
      "Inavelskoeff."));
5     List<Map<String, Object>> horsePageMiscData =
      getAllTableRowsDefinedByHeaderNames(userAgent,
      Arrays.asList("Ägare", "Uppfödare", "Tränare"));
6 }
```

The last part of this case could also be implemented by querying for each horse through the main page document form. Jaunt features a form permutation function in which HTTP request headers can be generated into `HTTPRequest` objects based on a form setting. The enables Jaunt to generate lists of different interesting form setting headers which can be executed in batch instead of using the `Form.submit()` function for each single form setting.

**Listing 10: Form permutations**

```
1  userAgent.visit("https://www.travsport.se");
2  Form form = Form form = userAgent.doc.getForm(0);
3  for (int i=0; i<horseNames.size(); i++) {
4      form.addPermutationTarget("searchString", horseNames[i]);
5  }
6  List<HttpRequest> requests = form.getRequestPermutations();
7
8  for(HttpRequest request : requests){
9    userAgent.send(request);
10   // Crawl to horse page
11   // Scrape data
12   ..
13 }
```

### 4.1.4. Selenium

Selenium provides several advanced features for remote controlling browsers but also incorporates its own headless browser. All the functionality can be controlled programmatically from a hosting application or from within a browser as a plugin feature. Being developed mainly for unit testing of web applications, Selenium provides powerful functionality for acting in a browser environment. To traverse the DOM, several methodologies are incorporated other than tag finders such as Xpath and CSS selectors which are also supported.

A session is initiated by instantiating one of the browser drivers or the embedded headless browser (`HtmlUnitDriver()`) which uses the `HtmlUnit` headless browser. A GET request can then be executed simply by using the method `get()` or `navigate().to()`.

**Listing 11: Instantiating a Selenium headless browser and posting a form**

```
1  WebDriver driver = new HtmlUnitDriver() {
2      @Override protected WebClient modifyWebClient(WebClient
       webClient){
3          WebClient answer=super.modifyWebClient(webClient);
4          answer.getCookieManager().setCookiesEnabled(true);
5          webClient.getOptions().setUseInsecureSSL(true);
6          webClient.getOptions().setSSLClientProtocols(new
       String [] {"TLSv1"});
7          return answer;
8      }
9  };
10 driver.get("https://www.travsport.se");
11 WebElement queryField=
       webDriver.findElement(By.name("searchString"));
12 queryField.sendKeys(trainer);
13 queryField.submit();
```

In order to correctly handle the servers SSL security connection, the web driver has to be instructed explicitly about this. This is only necessary for some servers and will in most cases be handled automatically which unfortunately was not the case here, this is a problem when constructing automated solutions because it causes uncertainty. When the page is retrieved, the DOM of the document can immediately be accessed and processed. The form field `searchString` is filled out by using the `sendKeys()` function and then the queryfield itself can be used as a reference point for submitting that particular form. Selenium will automatically traverse the DOM in order to find out what form this element is a part of for the submit action to be correctly handled in the case of multiple forms. Submitting a form is the only way to do `POST` requests. Selenium lacks the possibility to use any other HTTP

verbs which means that `GET`. `PUT`, `POST`, `DELETE`, etc. requests cannot be explicitly generated and sent.

The headless browser of Selenium then follows the redirects and reacts the same way as a regular web browser as the form is posted which leads the browser to the resulting page. In order to verify the current location, Selenium provides access for the location.

**Listing 12: Get the current web location**

```
1 String browserLocation=webDriver.getCurrentUrl();
```

Unfortunately, Selenium provides no insight into status codes and response headers resulting from a request. When developing automated solutions without that functionality, validation becomes very problematic. This means that `getCurrentUrl()` in conjunction with whatever document was passed has to be analyzed in order to figure out what actually happened during the request.

Assuming the request is successful, several hits are possible at this stage and the sought after link will be in a specific table which lacks meta-identifier. Identification has to be done by extracting the heading of the tables to ensure the links are extracted from the correct table.

In order to extract table rows separately and identify the tables by their headings, a general helper function was implemented.

**Listing 13: A general helper function for extracting table data**

```
1  public static List<Map<String, Object>>
       getAllTableRowsDefinedByHeaderNames(WebDriver wd,
       List<String> headerNames) {
2      List<Map<String,Object>> resultList=null;
3      List<WebElement> allTables =
       wd.findElements(By.tagName("table"));
4      Map<String, Object> result=new HashMap<String, Object>();
5
6      for (WebElement table: allTables) {
7          List<WebElement>
       rows=table.findElements(By.tagName("tr"));
8          List<WebElement>
       headerRow=rows.get(0).findElements(By.tagName("th"));
9          ArrayList<String> hds=new
       ArrayList<String>(headerNames);
10         for (WebElement h_c: headerRow) {
11             String hText=h_c.getText();
12             if (hds.contains(hText)) hds.remove(hText);
13         }
14         if (hds.size()!=0) continue;
15         resultList=new ArrayList<Map<String, Object>>();
16         for (WebElement row: rows.subList(1,rows.size())) {
17             Map<String, Object> rowMap=new HashMap<String,
       Object>();
18             Iterator<WebElement> headerIterator=
       headerRow.iterator();
19             for (WebElement c :
       row.findElements(By.tagName("td"))) {
20                 String hText=headerIterator.next().getText();
21                 if (!headerNames.contains(hText)) continue;
22                 rowMap.put(hText,c);
23             }
24             resultList.add(rowMap);
25         }
26     }
27     return resultList;
28 }
```

The above table extraction algorithm is implemented by exploiting the DOM tree. The main function is `findElement()` and `findElements()` which returns the element or elements that have been found using a locator or query object. A query

object can be constructed using the `By` package which implements several strategies such as `id()`, `className()`, `tagName()`, `name`, `linkText()`, `partialLinkText()` and `cssSelector()`. The usage of these query objects are quite trivial and will not be elaborated on at this point.

With the aid of the table extraction algorithm, retrieving the correct table and extracting a hyperlink in order to find the document needed for the next step is trivial.

**Listing 14: Retrieving the correct table and finding the correct hyperlink**

```
1  List<Map<String, Object>>
       tableRows=getAllTableRowsDefinedByHeaderNames(webDriver,
       Arrays.asList("Kusk/Ryttare/Tränare"));
2  WebElement trainerElement=findTrainer(trainer, tableRows);
3  if (trainerElement!=null)
       trainerElement.findElement(By.partialLinkText(trainer)).click();
4  public static WebElement findTrainer(String
       targetName,List<Map<String, Object>> table) {
5      for (Map<String, Object> row: table) {
6          for (Map.Entry<String, Object> col: row.entrySet()) {
7              String
       trainerName=(((WebElement)col.getValue()).getText()).split(",")[0];
8              if (trainerName.equals(targetName)) return
       (WebElement)col.getValue();
9          }
10     }
11     return null;
12 }
```

Using the `partialLinkText()` query object makes it very straightforward to identify the correct hyperlink since we are only traversing the DOM branch starting with the table cell identified to hold the relevant hyperlink. Selenium implements a `click()` method which can be used to simulate a mouse click on any kind of element. Many elements do not respond to this kind of user input but several do, and since Selenium supports JavaScript any kind of tag can be used for manipulative or dynamic events. In this case, when the hyperlink is clicked with the `click()` method, the headless browser follows the URL and loads the new document.

A new web document is retrieved and in this document we need to switch the view for a list of horses active in training. This is done by clicking a link called `"TRÃNINGS-LISTA"`.

**Listing 15: Finding a specific link and following it.**

```
1 webDriver.findElement(By.partialLinkText("TRÄNINGS-")).click();
```

In the DOM inner text node, this button is divided by a line break, "TRÃNINGS-<br>LISTA". However, there are no other buttons in this document that contains a hyperlink named similarly and a global partial search can be used for "TRÃNINGS-" to identify the button instantly and follow the URL link.

Reaching a trainers individual page containing its horses in training provides us with interesting data to scrape. The information is contained within a table lacking any identification other than the table headings in which the earlier table extraction algorithm can be used.

**Listing 16: Retrieve the list with a trainers horse information and references to horse documents**

```
1 List<Map<String, Object>>
      horseTableRows=getAllTableRowsDefinedByHeaderNames(webDriver,
      Arrays.asList("Hästnamn","Ålder",
      "Kön","Startprissumma","Startpoäng"));
2 if (horseTableRows==null) return;
```

The next step is to retrieve all the URLs for each of the horses in order to visit each of the documents and extract more semi-structured data. It is necessary to store all these URL's separately because as we retrieve a new page, all the `WebElements` currently stored are lost. `WebElement` only store pointers to the branches in the currently loaded DOM tree and if a new document is retrieved it will make previously stored `WebElements` stale, meaning that they are invalidated. By extracting the `HREF` attribute it can be stored in a `String`.

**Listing 17:**

```
1 List<String> horseUrl=new ArrayList<String>;
2 for (Map<String,Object> o: horseTableRows)
3     horseUrl.add(((WebElement)
      o.get("Hästnamn")).findElement(By.tagName("a")).
      getAttribute("href"));
```

Finally it is possible to visit each of the pages containing the specific horse information and extract it.

**Listing 18:**

```
1  for (String sUrl: horseUrl) {
2     webDriver.get(sUrl);
3     List<Map<String, Object>> horseInf =
      getAllTableRowsDefinedByHeaderNames(webDriver,
      Arrays.asList("Färg", "Född", "Kön", "Ras", "Avelsindex",
      "Inavelskoeff."));
4     List<Map<String, Object>> horseAd=
      getAllTableRowsDefinedByHeaderNames(webDriver,
      Arrays.asList("Ägarpseudonym", "Uppfödare", "Tränare"));
5  }
```

#### 4.1.5. Play framework implications

As far as this case goes, none of the tested solutions had any issues working in conjunction with the Play framework.

### 4.2. Evaluation results

#### 4.2.1. jArvest Framework

The approach to harvesting or scraping with the jArvest framework has its largest benefits when static content is to be collected and forwarded into another solution for further processing. A big drawback is the unability to successfully handle session cookies, which contrary to what the documentation states, did not work. This issue brought the investigation into a halt because it was simply not possible to move forward without this basic functionality that maintains a session between a server and a client browser.

Using jArvest for scraping in the deep web would probably have been problematic caused by the lack of logic in its DSL. There are simply too few transformer options to successfully search, identify and manipulate retrieved document data which will force the developer to compromise and dynamically create robots to execute in order to use programmatic Java logic instead.

The case required the initial condition to be dynamic and there is a lack of predefined meta data such as id's and attributes in the retrieved documents, searching through them to find the relevant parts was necessary as well as being able to assess the next step to be taken or abort the whole process.

jArvest contains a lot of functionality which should make it a stellar choice for web scraping in a general sense because it handles many of the usual tasks such as connecting, retrieving, posting and DOM traversal. The DOM functionality is

quite comprehensive as there is support for XPath expressions, CSS selectors and tag identification. There is also support for AJAX.

The robots must be predefined in a JRuby based DSL which makes it hard to use Java native functions and logic for assessing structure of documents. The DSL is not comprehensive enough for searching and manipulating the retrieved documents. For instance, retrieving a document and following a hyperlink that is identified in the document is not possible and another robot has to be defined for retrieving that specific document. This makes it very unfitting for crawling and dynamic content handling and raises its implementation complexity.

### 4.2.2. Jaunt vs Selenium

There is big similarity among the two tools. They both employ a headless browser which share many similarities to how a regular browser works and contain methods that resemble how users approach internet browsing. However, Selenium is a large suite for unit testing web applications and supports XPath expressions, CSS selectors and its own native DOM traversal methodologies while Jaunt only has its own DOM tree methods.

#### Obtained functionality

Functionality for handling the HTTP protocol on a basic level is good for both Jaunt and Selenium Webdriver, both of the solutions provide a simple and intuitive API to get a web document. However, the lack of support in Selenium for making requests with the standard HTTP verbs other than `GET` is a very weak spot. Jaunt provides a simple interface for requesting with any of the standard verbs while Selenium only provides this kind of interface for the `GET` verb.

When greater detail is needed for editing request headers or analyzing response headers, Jaunt provides several ways. Complete request headers can be created in advance or request mappings can be changed as the request is executed. Response headers and status codes can be accessed in full to be examined when a HTTP response has been received. Selenium WebDriver on the other hand lacks this functionality completely and cannot even provide a http response status code.

As a document is retrieved, both of the tools parse its DOM tag tree automatically and provides a set of complete DOM handling functions. Tag elements can be selected as singles or as lists of all the matching elements. In Jaunt, queries can also be set to look for nested DOM elements or non-nested which can be a powerful feature. However, Selenium WebDriver also supports XPath expressions which can be very handy for pinpointing a specific element within the DOM. Another great feature of Selenium WebDriver is that it also implements CSS Selectors.

Jaunt has other tools for navigating the DOM, the API features native methods for extracting forms, tables and hyperlinks. Forms and tables can be selected by number, element or query. Hyperlinks can be iterated over a whole document. Jaunt also supports regex strings for searching through the DOM which makes different kinds of advanced wildcard searches possible.

Crawling web sites is comprehensively supported by both of the tools as identified hyperlinks can be followed in one single command. Posting forms is also very easy to do by simply identifying the correct form, defining the field data to be entered and submit it. Even though Selenium WebDriver does not support the POST verb in its API, it does seamlessly post forms defined in a web document.

Another aspect of crawling web sites is the fact that many web site use relative URL path references. Jaunt cannot handle relative URL paths, and trying to request resources using them fails. Selenium WebDriver delivers full support for these.

Jaunt offers even deeper functionality for forms which simplify the process of handling different elements within the form such as radiobuttons, textfields, checkboxes, multiple submit buttons etc. The most prominent feature in Jaunt form handling is the form permutation, this can be used to fill a form with all kinds of different data and store the request headers for each of the form permutations and when all is collected, send the requests in batches contrary to posting one form at a time.

Selenium WebDriver does not expose the DOM element/branch HTML code while Jaunt does this in several ways. In Jaunt a element can be selected and the the outer HTML, inner HTML and text nodes is exposed. However, when extracting the text nodes, they often have to be cleaned up from junk special characters appearing and the translation of HTML special characters is far from perfect. Selenium delivers a better performance when extracting text nodes and no junk characters appear while the translation of HTML special characters into Java strings is flawless.

Elaborated scraping features for semi-structured data is scarcely provided by the two tools. Jaunt provides a table object wrapper which aids the extraction of tables. It does not make it alot easier than using the DOM, but it provides mappings between positions and header names for tables. Selenium has no such features at all and structure has to be relied upon the DOM.

In Selenium WebDriver there are possibilities to gather data relating to the graphical representation of a web document such as attaining positions and sizes of elements while Jaunt has no such feature. However, using the HtmlUnit headless browser of Selenium these features was not able to provide any such data while running Selenium with a different driver will. Further, Selenium WebDriver supports JavaScript and AJAX which enables scripts to be run in the same manner as in a regular browser and actions to be triggered programmatically as well.

The results are comprised in figure 4.1.

**Solution complexity and ease of implementation**

Case solutions for Jaunt and Selenium can be created almost identically because they feature similar DOM handling and overall there is a high degree of ease of implementation because many of the methods provided through the API's are self descriptive and easy to implement.

Both of the tools are Java compliant and there is no reason why they wouldn't be properly supported in any of the available Java IDE's, Intellij IDEA and Eclipse was used during this investigation. There was no issues whatsoever in conjunction with the Play Framework as well. However, scraping huge amounts of data in conjunction with having to crawl through web sites may consume alot of time caused by latencies between requests and responses. This can be problematic in a web application in which the user will have to wait a long time for the answer.

Restructuring the main application was not necessary to any larger extent and embedding the dependencies and main libraries was not problematic either.

As mentioned above, Jaunt has poor capabilities for handling HTML special characters and induces junk characters into extracted text strings. This raises the need for debugging and calls for confusion when using regular expressions for matching or structuring the data which sometimes needs to be cleaned up.

Selenium WebDriver SSL encryption control is buggy and works for some sites while misinterpreting others, resulting in having to manually configure what protocols to be used while Jaunt did not expose any of these kinds of weaknesses. The exceptions raised from Selenium can be hard to interpret and documentation covering these aspects are non existant making Selenium more complex to work with.

However, in a general sense the ease of implementation is high and both tools provide methods and an API that is easy to work with and effective to develop in while complexity is low compared to implementing logic in jArvest which has shortcomings in its DSL.

The results is compiled in figure 4.2.

| | jArvest | Selenium WebDriver | Jaunt |
|---|---|---|---|
| **Obtained functionality** | | | |
| Basic HTTP protocol verb request API | Partial(GET and POST) | Partly(GET supported and POST only through forms) | Yes(GET, POST, PUT, DELETE and HEAD) |
| HTTP header exposure | None | None | Request and response |
| HTTPS support | Yes | Yes, but buggy | Yes |
| General DOM identification | Xpath, CSS Selectors | DOM queries for tags and attributes, XPath specifiers and CSS selectors | DOM queries for tags and attributes |
| Other DOM identification methods | None | Hyperlink- , partial hyperlink- | Table-, form- and hyperlink name-search |
| Searching traversal methods | None | Find first element or find all elements. | Find first, find all, nested and non-nested elements. |
| Wrappers for DOM elements | None | None | Table, form and hyperlink. |
| Form handling | None | Handles multiple forms, fills textfields and other elements. Submits branched elements even if form is unknown. | Multiple forms per page, wraps to own object, several different ways of identification, fills all elements. Form permutations. |
| Relative URL paths | Unknown | Yes | No |
| DOM HTML exposure | None | Only the full document HTML | A element or a elements inner or outer HTML is available |
| Parse HTML special characters | Unknown | Fully | Partial |
| Broken HTML | Yes | Yes | Yes |
| Graphical UI data | No | Yes, but not for HtmlUnitDriver | No |

Table 4.1: Result table.

| | jArvest | Selenium WebDriver | Jaunt |
|---|---|---|---|
| **Solution complexity and ease of implementation** | | | |
| IDE support | Partial, but not for DSL | Yes | Yes |
| Additional dependencies | Several | Several | None |
| Compability issues with Play Framework | None | None | None |
| Stability issues | None | None | None |
| General ease of implementation | Low | High | High |
| Solution Complexity | High | Low | Low |

Table 4.2: Result table 2.

# 5. Conclusion

In this case, the best solution for implementing a web wrapper for Top of Europe was the Jaunt library. Both Selenium and Jaunt provided solutions sufficient for engaging in many web data extracting tasks while jArvest failed to deliver in this case implementation. The reason jArvest could not complete the case implementation was because of its lack to prevail a basic session by using session cookies.

The obtained functionality in Jaunt and Selenium is very similar but each do have its own strengths and weaknesses. Jaunt has many superior features related to effectively wrapping HTML DOM objects and exposing HTTP headers for editing or examination while Selenium WebDriver provides JavaScript, AJAX and some graphical support. Using a headless browser as platform in a web data extraction system provides tools to quickly access web resources and hides much of the technological complexities because they already contain many of the basic handling routines necessary for interaction with web resources. The literature in the field of web data extraction is very focused on how to handle a semi-structured document and preserve pointers to objects within these documents as the site evolves but this task becomes pointless if there isn't a solid construction to find and navigate to the data also. Browsers have been developed for several years in a ever co-evolving pattern along with web communication and architectural technology which provides new ways to access data through new structure of URL's and dynamic web documents. For extraction methods to stay relevant, they need to adhere to these kinds of technologies as well as successfully identify parts in documents for scraping.

Using these solutions in a live situation where a user requests the data to be extracted through a web user interface, time and efficiency also becomes important because it will directly affect usability of the web user interface. If several queries are to be posted, Jaunt has the form permutation feature which is handy in these situations. Jaunt also provides API to produce header requests which might aid in requesting specific data in batches while Selenium WebDriver has to do it in a manual browser related fashion by requesting a document, filling the form and posting it for each of the batch items which implies more requests and increased time windows spent.

The case solution implemented in this report does not use XPath because it

is too connected to a specific documents tree structure. The case did not use complex tree or machine learning algorithms either because these are not supported in any specific way in the reviewed tools and also, they were not interesting for the company to implement caused by the complexities of maintaining them. Both Jaunt and Selenium implements powerful features to locate certain elements within the DOM that provide a more loose connection between elements DOM tree placement and the elements features. This makes solutions easier to maintain and change but also provide better performance compared to tree edit algorithms or machine learning algorithms.

Jaunt struggles with character handling both for translating HTML special characters but also includes arbitrary junk characters which may induces problems if they are not catched. This raises the need to clean up scraped data which means that a automated solution has to be overlooked at given points in order to not corrupt database items stored. Debugging these kinds of problems can be hard and confusing when the source data extracted does not contain these junk characters. Manual supervision of the extracted data is thus neccessary.

Jaunt is also only available with time-limited licensing, 1 month for the free version and up to 12 months for commercial. This is unfortunate in the case when automated solutions are produced in a "set and forget" way because when the time limitation runs out, it just stops working and a new library has to be imported.

Several web sites utilize JavaScript and AJAX which is not supported in Jaunt contrary to Selenium WebDriver. These features enable dynamic content to be retrieved and published in the DOM of a web document and thus, can also be extracted. In Selenium WebDriver, these technologies are incorporated seamlessly in the same manner as a regular browser upon activation. As a document is retrieved, any script will be executed.

Finally, some caution should be taken when wrapping web sites and automating retrieval of large amounts of documents. If a client requests a very large amount of documents within a small timespan, there is a chance that a server will ban the client and break the wrapper.

## 5.1. Recommendations

When web scraping for data in the Play framework or in the Java language in general, the preferred route according to this evaluation is to use Jaunt. However, if JavaScript and AJAX is needed to handle the dynamic nature of many web sites and applications, it will not suffice and Selenium WebDriver is a stronger alternative.

## 5.2. Future work

A step forward in this area is to investigate into more elaborate methodologies for web scraping that could be implemented into headless browsers. Headless browsers provide a very elaborated interface towards the web and also may provide remote

control functionality for real browsers as in the case of Selenium, this makes for excellent platforms for accessing and interacting with web resources. However, as concluded in this report, available initiatives can be regarded as rather primitive tools in that they mainly provide functionality that is based completely on human web browser usage patterns (pointing, clicking and filling forms) and maybe automated interactions could be explored in a different fashion than the traditional.

# References


[1] C. A. Iglesias J. I. Fernandez-Villamor and M. Garijo. A framework for goal-oriented discovery of resources in the restful architecture. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 44(6):796–803, 2014. doi: 10.1109/tsmcc.2013.2259231.

[2] G. Gottlob M. Herzog R. Baumgartner, O. Frölich and P. Lehmann. Integrating semi-structured data into business applications: A web intelligence example. *Lecture Notes in Computer Science*, pages 469–482, 2005. doi: 10.1007/11590019_54.

[3] M. Bhat N. S. Purohit, A. B. Angadi and K. C. Gull. Crawling through web to extract the data from social networking site-twitter. In *Parallel Computing Technologies (PARCOMPTECH), 2015 National Conference on*, pages 1–6. IEEE, 2015.

[4] Aldo Geuna, Rodrigo Kataishi, Manuel Toselli, Eduardo GuzmÃ¡n, Cornelia Lawson, Ana Fernandez-Zubieta, and Beatriz Barros. Sisob data extraction and codification: A tool to analyze scientific careers. *Research Policy*, 44(9): 1645–1658, 2015. doi: 10.1016/j.respol.2015.01.017.

[5] J. Hendler T. Berners-Lee and O. Lassila. The semantic web. *Sci. Am.*, 284 (5):28–37, 2001.

[6] K. Idehen C. Bizer, T. Health and T. Berners-Lee. Linked data on the web. *Proc. 17th Int. Conf. WWW*, pages 1265–1266, 2008.

[7] T. Urvoy T. Lavergne and F. Yvon. Filtering artificial texts with statistical machine learning techniques. *Lang Resources & Evaluation*, 45(1):25–43, 2010. doi: 10.1007/s10579-009-9113-0.

[8] H. Lopez-Fernandez M. Reboiro-Jato D. Glez-Pena, A. Lourenco and F. Fdez-Riverola. Web scraping technologies in an api world. *Briefings in Bioinformatics*, 15(5):788–797, 2013. doi: 10.1093/bib/bbt026.

[9] M. Garofalakis-A. Fekete C. S. Jensen R. T. Snodgrass A. Wun V. Josifovski A. Broder S. Mankovskii, M. van Steen and D. Fetterly et al. Web data extraction system. *Encyclopedia of Database Systems*, pages 3465–3471, 2009. doi: 10.1007/978-0-387-39940-9_1154.

[10] M. Ceresna R. Baumgartner and G. Ledermuller. Deepweb navigation in web data extraction. *International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06)*, 2005. doi: 10.1109/cimca.2005.1631550.

[11] N. Kushmerick. *Wrapper induction for information extraction.* PhD thesis, University of Washington, 1997.

[12] G. Fiumara E. Ferrara, P. De Meo and R. Baumgartner. Web data extraction, applications and techniques: A survey. *Knowledge-Based Systems*, 70:301–323, 2014. doi: 10.1016/j.knosys.2014.07.007.

[13] Z. Yanhong and L. Bing. Structured data extraction from the web based on partial tree alignment. *IEEE Transactions on Knowledge and Data Engineering*, 18(12):1614–1628, 2006. doi: 10.1109/tkde.2006.197.

[14] D. Hu X. Meng and C. Li. Schema-guided wrapper maintenance for web-data extraction. *Proceedings of the fifth ACM international workshop on Web information and data management - WIDM '03*, 2003. doi: 10.1145/956699.956701.

[15] E. Ferrara and R. Baumgartner. Intelligent self-repairable web wrappers. *AI*IA 2011: Artificial Intelligence Around Man and Beyond*, pages 274–285, 2011. doi: 10.1007/978-3-642-23954-0_26.

[16] P. Bohannon N. Dalvi and F. Sha. Robust web extraction. *Proceedings of the 35th SIGMOD international conference on Management of data - SIGMOD '09*, 2009. doi: 10.1145/1559845.1559882.

[17] W3.org. W3c document object model 4, 2015. URL `http://www.w3.org/TR/dom/`.

[18] W3.org. Xml path language (xpath), 1999. URL `http://www.w3.org/TR/xpath/`.

[19] W3.org. Xml path language (xpath) 2.0 (second edition), 2011. URL `http://www.w3.org/TR/xpath20/`.

[20] W3.org. Selectors level 3, 2015. URL `http://www.w3.org/TR/css3-selectors/`.

[21] X. Cheng L. Jiang Q. Zheng, Z. Wu and J. Liu. Learning to crawl deep web. *Information Systems*, 38(6):801–819, 2013. doi: 10.1016/j.is.2013.02.001.

[22] M. Shokouhi. Federated search. *Foundations and Trends in Information Retrieval*, 5(1):1–102, 2011. doi: 10.1561/1500000010.

[23] D. Glez-Pentilde. jarvest, 2015. URL `http://sing.ei.uvigo.es/jarvest/`.

[24] Jaunt-api.com. Jaunt - java web scraping & automation, 2015. URL http://jaunt-api.com/.

[25] Seleniumhq.org. Selenium - web browser automation, 2015. URL http://www.seleniumhq.org/.

[26] M. Guillemot M. Bowler and A. Ashour. Htmlunit â welcome to htmlunit, 2015. URL http://htmlunit.sourceforge.net/.