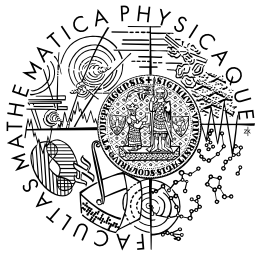


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Petr Fejfar

Interactive crawling and data extraction

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	Mgr. Pavel Ježek, Ph.D.
Study programme:	Computer Science (N1801)
Study branch:	Software Systems (2612T043)

Prague 2018

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

Prague, 20 July, 2018 Petr Fejfar

I would like to thank to Mgr. Pavel Ježek, Ph.D. for his guidance, precision and all the useful advices. Special thanks goes to my girlfriend Markéta for her patience, support and countless weekends she needs to spent without me. Thanks also go to my friend Jakub, for his support and motivation. Last but not least, I feel grateful to my family for supporting me my whole life, especially during the time of writing this thesis, even though I had so little time for them.

Rád bych poděkoval Mgr. Pavlu Ježkovi, Ph.D. za jeho rady a věcné připomínky, které mi pomohly napsat tuto práci. Velmi bych chtěl poděkovat přítelkyni Markétě za její podporu, trpělivost a nespočet víkendů, které musela strávit beze mně. Rád bych také poděkoval mému dlouholetému příteli Jakubovi pro jeho podporu a motivaci, kterou mi dal. V neposlední řadě se citím být vděčen mé rodině za podporu, kterou, kterou mi projevují celý život a obvzlášt v době psaní této práce, přestože jsem se jim tak málo věnoval.

Title: Interactive crawling and data extraction

Author: Bc. Petr Fejfar

Author's e–mail address: pfejfar@gmail.com

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek, Ph.D., Department of Distributed and Dependable Systems

Abstract: The subject of this thesis is Web crawling and data extraction from Rich Internet Applications (RIA).

The thesis starts with analysis of modern Web pages along with techniques used for crawling and data extraction. Based on this analysis, we designed a tool which crawls RIAs according to the instructions defined by the user via graphic interface. In contrast with other currently popular tools for RIAs, our solution is targeted at users with no programming experience, including business and analyst users.

The designed solution itself is implemented in form of RIA, using the WebDriver protocol to automate multiple browsers according to user–defined instructions. Our tool allows the user to inspect browser sessions by displaying pages that are being crawled simultaneously. This feature enables the user to troubleshoot the crawlers.

The outcome of this thesis is a fully design and implemented tool enabling business user to extract data from the RIAs. This opens new opportunities for this type of user to collect data from Web pages for use as primary data, as well data for further analysis.

Keywords: Web crawling, Web data extraction, Web scraping, AJAX, RIA, Rich Internet Application, browser automation

Název práce: Interaktivní procházení webu a extrakce dat

Autor: Bc. Petr Fejfar

E–mailová adresa autora: pfejfar@gmail.com

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí práce: Mgr. Pavel Ježek, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Tato práce se zaměřuje na problematiku automatického procházení stránek a extrakce dat v kontextu moderních webových aplikací, obsahujících vysoké množství aplikační logiky implementované v prohlížeči pomocí JavaScriptu.

V práci je provedena analýza moderních webových stránek, spolu s technikami, které jsou bežně používany k extrakci dat. Na základě této analýzy jsme navrhli nástroj, který moderní webové stránky prochází na základě instrukcí zadaných uživatelem pomocí grafického prostředí. Narozdíl od ostatních nástrojů na procházení a extrakci dat z moderních webových stránek, náš nástroj umožnuje práci uživatelům, kteří nemají zkušenosti s programováním.

Navhrhovaný nástroj je implementován jako webová aplikace a využívá protokolu WebDriver pro automatizaci více prohlžečů pro procházení a extrakci dat z webových stránek pomocí uživatelem definovaných posloupností instrukcí. Náš nástroj umožňuje uživateli prozkoumat aktuální stav prohlížeče extrahujícího data zobrazením aktuálně prochazené stránky. Toto umožní uživatelům vyhledávat a ladit chyby jejich posloupností instrukcí, tak aby extrahovaly data, které mají extrahovat.

Výstupem této práce je návrh a následná implementace nástroje pro extrakci dat z moderních webových stránek pro uživatele bez schopnosti programovat. Tento nástroj umožní sběr dat, který dříve nebyl možný. Tyto data mohou být využity pro další analýzu nebo jako vstupní data do dalších systému.

Klíčové slova: Web crawling, Web data extraction, Web scraping, AJAX, RIA, Rich Internet Application, browser automation

Contents

Part I

Introduction

1. Introduction

Over the past years, World Wide Web has rapidly evolved in many respects. One of the elements which have had a significant impact on transformation of the Web is larger usage of client–side JavaScript, caused by the need of Web page creators to provide richer user experience. These heavy client side JavaScript pages are called Rich Internet Applications (RIA)[1].

The number of RIAs – sources of significant portion of Web data – is continuously growing, however we can find the lack of crawling and extraction tools, which would provide the way to obtain information from Web pages automatically. We believe, that this is caused by new technical challenges that go hand by hand with RIAs. Although, the topic of crawling and data extraction of non–RIAs (static Web pages) has been covered almost entirely and state of the art open source implementation can extract data from pages efficiently (for example Apache Nutch [1]), we consider solutions for crawling and data extracting from RIAs very immature. Crawlers and data extraction tools need to evolve to adapt RIAs.

We believe that this gap can be filled by creating an interactive data extraction tool for RIAs. From the perspective of a user, this new tool would let him or her visually[2] define all the information for extraction from the Web page (we call these information *page model*). Such tool would provide the user with an opportunity to crawl Web pages without knowledge of underlying Web technologies and technical details would stay hidden. Therefore new type of user – the one without deeper technical knowledge or business user – can manage this tool. The user would have a browsing–like experience and the difference between static Web pages and RIAs would remain hidden. At the moment, mainly programmers are able to crawl and extract data from RIAs by creating custom scripts. This tool would be a significant improvement leading to the spread of Web crawling and data extraction tools for RIAs, to help in various scenarios.

The idea for this large-scale solution arose from author's own experience with Web crawling and data extraction tools at IBM, where he experienced a rising demand for a solution allowing to extract data from thousands of websites, as well for one that remains on premise to avoid breach of customer data.

Data obtained from the Web page by this proposed tool can be used in countless use cases. The major ones are data and opinion mining, business and competitive intelligence and extracting data from social networks to

[1] They are also called Web application, single page applications or dynamic pages.

[2] Visually means, that user sees rendered Web page and he/she selects elements on page using mouse pointer.

further analysis, search engines, etc. Use cases remain the same for RIAs as well as for static Web pages. We advise reader to explore other resources to find more use cases. An example would be a survey about applications of crawling and Web data extraction techniques, made by Ferrara et al. [2].

Considering the fact that RIAs are on the rise and there is currently no efficient way to crawl them, it is worth focusing on this topic and creating such tool, which lets the user define what data can be extracted from RIA and then crawls this page according his/her definition. This tool would enable to obtain data in structured format for specific Web sites. The user would interact with crawler in following steps (step's number corresponds with numbers in Figure 1.1):

1. The user opens the tool and specifies page, which he/she wants to crawl.
2. Crawler tool shows a preview of Web page to the user.
3. The user selects, which elements he/she wants to extract and how to access these elements on different pages of Web site. We call these information *page model*.
4. The user starts crawling based on *page model*.
5. The crawler uses the *page model* as source of information on what part of Web page to extract.
6. During extraction step, crawler is accessing Web page and extract data according to the *page model*.
7. The user can use extracted data to process it in other system.

This thesis goal is to create such tool. Tool will addresses challenges related to RIAs and create a platform for future development of universal Web crawler.

1.1 Challenges

Rich Internet Applications present new challenges compared to static Web pages. In this section, we will look into some of these challenges to present an overview of problems, which is necessary to be solved when crawling RIAs.

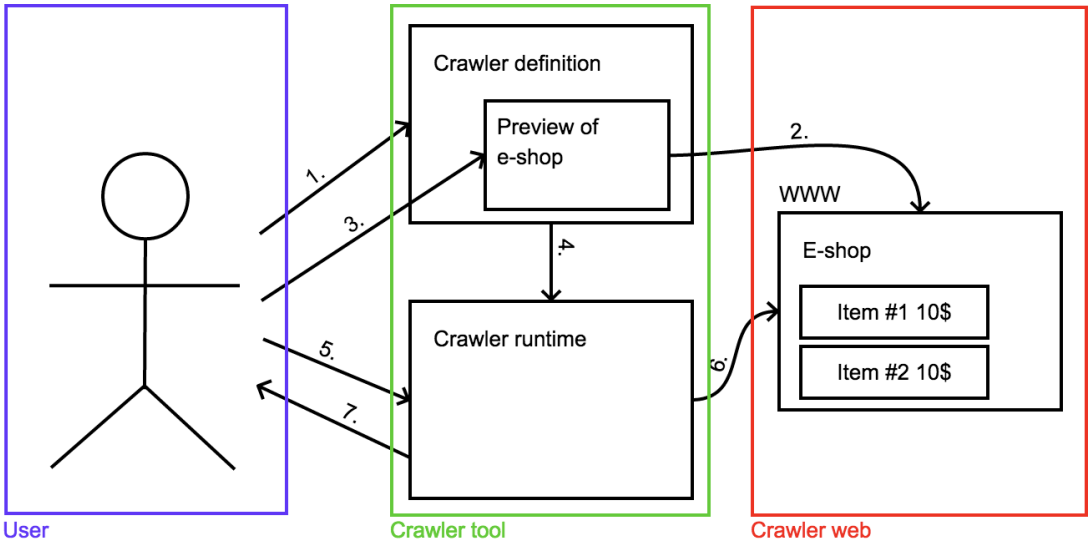


Figure 1.1: *Crawler–user interaction – high–level view.* This diagram shows user's interaction with crawler tool.

1.1.1 Search state space explosion

The characteristic of static Web pages is, that they are identifiable by URL. Therefore crawler accesses page's URL to obtain its content. State of crawling static pages can be represented as set of visited URLs and set of URLs to visit. Whole process of crawling can be expressed as state space search, where state is represented by URL and actions are represented by hyperlink between two Web pages. This fact simplifies the problem of crawling static pages.

However RIAs are not identifiable by URL. Web page can be shaped using client–side JavaScript and content of the page can vary in time. JavaScript code can fetch new content using HTTP calls any time. Actions caused by the user (for example clicking on element or page scrolling) or actions generated by JavaScript (see Figure 1.2 for illustration) can alter the page content as well. URL does not need to change during these changes, therefore URL does not uniquely identify the content.

Crawling of RIA can be expressed as state space search[3]. Static page crawling can be expressed as state space search, however representation of state space is more complicated. Following hyperlinks are no longer only circumstance to transition between states, additionally all JavaScript actions need to be reflected. Actions in this state space search are any Javascript actions, which can occur on the Web page. State is no longer identifiable by URL, but all possible variations of Web page changed by JavaScript. Considering

[3]Formal definition of crawling can in found in article from Mesbah et al. [3]. It is defined as search state space on graph. In this thesis there is no need to formally define crawling of RIA, therefore we leave this definition for simplicity.

all these variation of single page caused by JavaScript results in state space explosion.

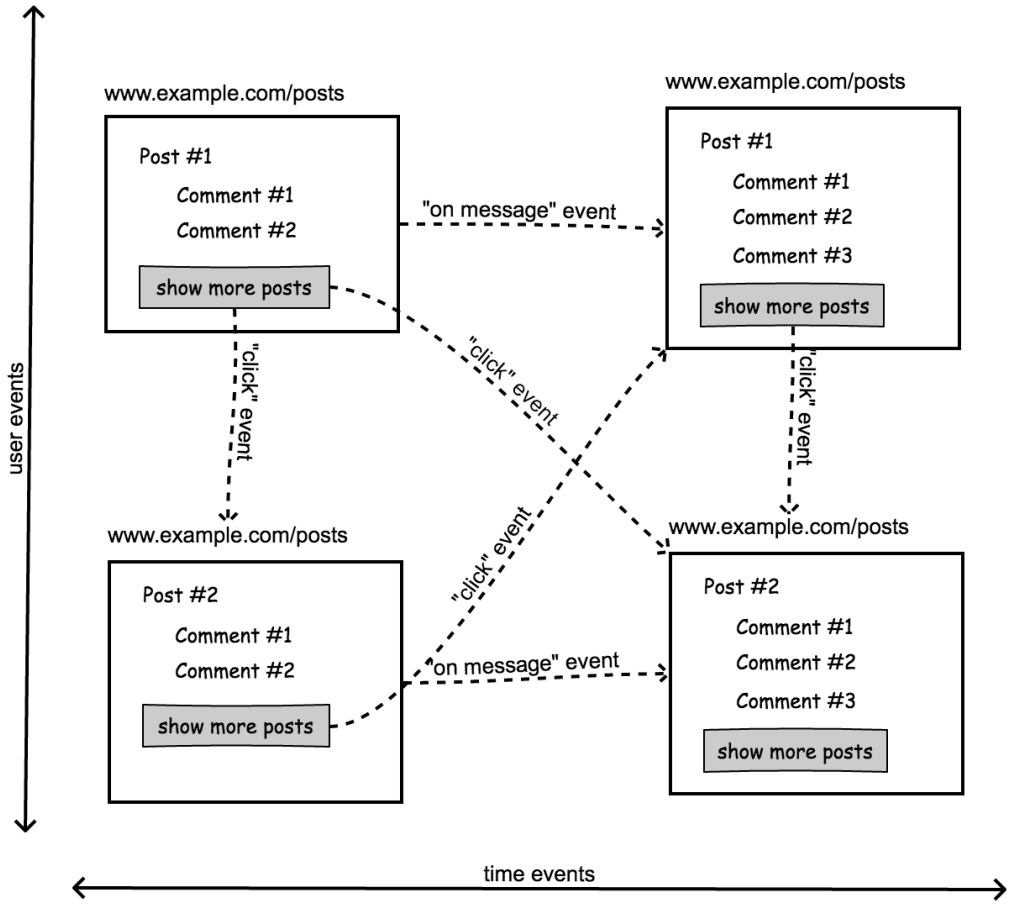


Figure 1.2: *Rich Internet Application crawling state space search.* Diagram represents example of Web page displaying posts and comments. Transition between states of this Web page are illustrated by dashed lines. One possible transition is caused by executing *click* action when the user clicks on the button to load next post. Another one is caused by received message (for example notification from server via WebSocket) with new comment. During the transition between these states, Web page URL remains the same.

Execution of all possible JavaScript actions is not possible, because of the state space size. However, majority of these possible actions will not obtain new content of the Web page. These actions could also change the server state. An example of such actions is sending an mail or adding new comment to discussion. This can be even considered as malicious behavior and it will not obtain new data when crawling. Therefore crawlers need to carefully pick actions to dispatch. This is not a problem while crawling static page, because pages are designed by Web developers to show information,

not to change server state. State of the server is changed by submitting Web forms in case of static pages.

1.1.2 Page dynamism

When dealing with RIAs, simple task such as load page becomes challenging as crawler cannot determine when page is fully loaded. JavaScript actions triggered by Web page JavaScript can fetch data from server via HTTP protocol and these actions can be trigger by any time. This results in challenge that crawlers cannot recognize when page is fully loaded with data or if any new event will generate new content.

The enormous search state space and a need to structured data output are reasons why the system cannot be automated simply by executing all JavaScript action on page and retrieve all content from page. Our proposed tool exploits the domain knowledge of users, who will define which parts of the Web page should be extracted to reduce this state space search size. Additionally the user specifies which interaction with the browser should be made to access more Web pages with content. We see limitation in fact, that this user, who has domain knowledge about the Web page and knows which parts of the page should be extracted, is not necessary a Web developer. This user could have limited knowledge about Web technologies. Therefore RIA crawler needs to reflect this fact and provide the user an experience, which would hide technical details from him/her.

Web browser needs to be used to process crawled page correctly. Usage of Web browsers will process JavaScript on page and handle all other resources (for example data fetched via HTTP). We will elaborate more on this in chapter 4.

Crawler of RIAs cannot rely on storing Web pages as Web Archive (WARC) [4], which is well defined format for storing crawled pages. WARC stores only HTML and other HTTP resources (images, CSS, etc.), because Web ARChive is assuming that one URL identifies one page. Therefore new format needs to be developed.

1.1.3 Captcha

Some Web pages are protected by Turing test to differentiate between regular user and crawler. These tests prevent crawler to retrieve data automatically. Very popular solution is to let the user read a character on image. There are usually transformations applied to this image, so it is difficult

to make to use optical character recognition systems. These solutions are usually called Captcha [5]. With rise of the deep learning techniques, these image recognition systems have became more fragile to automated solving by computers [6]. Also services providing manual Captcha solving became generally available. Response time and accuracy is high when automated Captcha solvers are combined with manual labor work. Authors of such service, called Death by Captcha, claims: *"An average response time of 11 seconds, with an average accuracy rate of 90% or more."* [7]. We think that services with such accuracy, can be used efficiently to solve these image recognition based Captchas.

Nowadays Turing test protecting page from crawling keeps evolving to adapt current crawlers. Google reCAPTCHA v2 [8] is example of system using artifical inteligence for behavioral analysis of user behaviour to detect if user is a crawler. *"This system uses Google's AI to look for signs of human behaviour. It runs in the background detecting movements of a mouse, how long it takes to click on a page ..."* [9]. According to Nan Jiang, these Turing test based systems will fail completely in the future: *"If we have really good AI technologies these could be mimicked by some AI algorithms we don't really know yet. It's a challenge between how we can retain the usability of the Captcha scheme whist maintaining good security."* [9]. We believe that the protection from crawling will become more significant in future and crawlers will need to adapt it. Especialy after data privacy scandal, which are rising nowadays [10].

1.1.4 Challenges summary

This section showed major problems, which are connected with crawling RIAs. Main challenges, which we will address in this work are search state space explosion and optimization of crawler tool for non–technical user, as he/she needs to provide domain knowledge about crawled Web pages. We are not addressing Captcha system, as solutions for solving are already available and can be integrated to crawler, however we want to emphasize, that this challenge needs to be solved in future work. The integration with these systems need further work, which is outside the scope of this thesis.

1.2 Goals

The main goal of this thesis is to analyze, design and implement solution, which would enable user with domain knowledge about crawled Web page to crawl and extract data from RIAs. The thesis will also analyze specific Web pages suitable for crawling to illustrate challenges of crawling RIAs.

The thesis will analyze modern Web pages in general, existing approaches to crawling and existing crawler solutions. The proposed new crawler tool will comply with requirements designed to address RIAs crawling challenges stated in following section.

1.3 Requirements

In this section, we will describe requirements for RIA crawler, which are based on challenges described earlier. These requirement are chosen by author of this thesis and they are defining output of this thesis. Summary of functional and non–functional requirements follows in next subsections.

In the first section, we introduced an idea of tool for crawling RIAs, which allows user to crawl data (requirement FR1) with browser–like experience. In the second section, we derived, that this user with domain knowledge needs to define what parts of page needs to be extracted and how to crawl all pages. Definition created by this user is called *page model*. User experience of the crawler is optimized for this user, because he/she will interact with the system the most. This user is usually business analyst or data scientist and has no deep knowledge about Web technologies. The user visually[4] defines which elements are affected by commands (requirement FR2). This will provide the user a browser–like experience, when creating *page model*. Defining crawler on this level of abstraction will hide technical details.

If crawler crawls a Web page and it fail to extract data (for example when Web page changed and element on which the crawler clicks does not exist anymore), system will provide a way how to determine, what was the cause of the problem. System will provide to the user a possibility of visual inspection of crawler run (requirement FR3). Visual inspection is that user can look on exact state of the crawled page.

Data extracted by the crawler will be used to process in other systems such as spreadsheets or business intelligence tools. Data need to be exportable in format supported in these systems (requirement FR4).

The creation of RIA crawler from scratch is a challenging task and due to limited scope of this thesis, it is not possible to fully finish it. The crawler needs to reflect the fact, that it will be extended in the future, therefore it should provide architecture for future expansion (requirement NR1).

Modern Web pages may contain large volume of data. Crawler needs to execute large volume of actions to obtain all these data (requirement NR2). In

[4] In this context, visually means, that user can see Web page rendered in browser and he/she selects element by mouse pointer.

section 4.3, there will be presented examples of real Web pages suitable to crawling. To make rough estimation, how many action will crawler execute, the reader should imagine Web page with e–shop page. If this page would have 1000 pages with 50 shop items on each page and crawler would need 10 actions to extract all information about item, total number of evaluated actions to extract all data from page is:

$$1000\text{page} \cdot 50\frac{\text{item}}{\text{page}} \cdot 10\frac{\text{action}}{\text{item}} = 500k \text{ action}$$

Another example is that crawling real estate page can take approximately 600k HTTP requests [11]. These two examples of Web pages imply, that crawler need to be prepared for executing roughly hundreds of thousands actions to crawl these pages. The crawler needs to incorporate these challenges to be able crawl Web page in short time.

To avoid accidental Denial of Service attack by accessing Web page too many times in short period of time, the crawler needs to limit rate of accesses of the crawled Web page (requirement NR3).

Crawler will be used by other systems. To enable automation of the crawler, it will expose an API (requirement NR4). This API will be used for several tasks such as starting crawler by external scheduler or automatically transfer data to another system.

List of functional requirements

FR1 *System must be able to obtain data from Web page.*

This is basic functionality of the system, which describes core scenario. The user wants to extract data from Web page and he/she uses crawler to achieve that.

FR2 *System must let the user define page model visually.*

When user is browsing Web pages, he/she is clicking on elements rendered on the screen. This requirement ensures the user similar user experience, therefore *page model* will be easy to create.

FR3 *System must be able to let the user inspect current crawler state.*

Page model is difficult to maintain as Web pages can change over time. The user needs to have a way how to determine what search state is crawler currently evaluating. This is used for troubleshooting *page models* used for crawling.

FR4 *System must be able to export extracted data*

Users will exploit data for further analysis. This analysis can be done in popular tools such as Microsoft Excel or business intelligence tools such as Microsoft PowerBI.

List of non–functional requirements

NR1 *Solution should have the crawling algorithm easily extended/changed.*

Due to search state space explosion (mentioned in section 1.1), the performance of the searching algorithm will have impact on performance of whole solution. This algorithm will be changed in the future for optimization and improvements, therefore crawler should be prepared for extension of this algorithm.

NR2 *System should be able to handle high volume of processed search space states when running crawler. Rough estimation is hundreds of thousands actions when crawling one Web page.*

Current Web site contains large amount of data. Crawler need to obtain all data in short time.

NR3 *System needs to have option to limit frequency of HTTP accesses to target page.*

By adressing requirement NR2, crawler will optimize number of Web page accesses in time. The crawler will obtain as much data as possible in short period of time. This can lead to Denial of Service attack to crawled Web page server, which the crawler can avoid by setting a limit of accessed to Web page.

NR4 *System needs to be controllable by other system. Therefore external API needs to be build.*

Integration to other system is crucial to usability of RIA crawler.

1.4 Organization of thesis

The thesis is organized as follows. In Part I, author introduces challenges coming with modern Web pages in context of crawling and data extraction. Goals of the thesis are defined in the next section, followed by the requirements needed for the solution, which are the core part of this thesis. Part II describes the preliminaries and technological background of modern Web and the next part of the thesis, Part III, contains a short case study on modern Web pages. In the same chapter, we highlight common approaches to

crawling modern Web pages and popular implementations based on these approaches. The design of the solution, which will fulfill the requirements stated in the Part I, will be presented in Part IV. Implementation of the solution is presented in the next part, Part V. Last part of the thesis, Part VI, is the conclusion and it sets the basis for future work on the topic of Web pages crawling and data extraction.

Part II

Background

2. Web technologies

This chapter describes technologies related to crawling and data extraction. Main focus is on Web related technologies, which the reader needs to know in order to understand concepts described later in the thesis. This chapter can be omitted, if the reader is already aware of these technologies. Second part of this chapter describes technologies used during implementation, such as databases or runtime.

2.1 World Wide Web and Web resources

The World Wide Web (WWW, or simply Web) is usualy referred as “information space” containing items of intererest, referred as Web resources, which are identifiable by Uniform Resource Identifiers (URI). [12]

Web resources are various types of documents identified by MIME type [13]. They are transfered using HTTP protocol [14] and mainly interpreted by browser. Typical examples are Web page, pictures, JavaScript script or JSON files. Web resources contain information, which is worth crawling. Web pages on single domain are usually referred as website.

2.1.1 HTML documents

Most common type of Web resource is HTML document. Hypertext markup language (HTML) is a markup language used to describe layout of the Web page and current used version is HTML 5 [15]. HTML 5 specification [16] defines series of interfaces using Web IDL [17]. These interfaces define interaction with the Web page. Browsers usually use JavaScript as primary language to implement these IDLs [18]. RIAs normally use JavaScript scripts interacting with these interfaces in order to provide interactivity to the user.

2.1.2 DOM

Document Object Model (DOM) is the set of APIs used to manipulate tree–like structure of Web page. This structure consists of nodes and node’s attributes, which represent layout of the Web page. “DOM defines a platform-neutral model for events, aborting activities, and node trees.” [19].

Part of DOM specification are selectors used to select subset of DOM tree. Selectors were primaly used by Cascading Style Sheets (CSS) [20] to apply visual properties (for example color, font size, borders, etc.) on set of HTML nodes. Web extractor can use selector to define subset of page they want to extract. Other option how to select subset of DOM tree is XPath [21]. These selector are commonly used by data extractor to specify which part of page to extract.

Another part of DOM specification that will be used during implementation of our crawler is MutationObserver [22]. MutationObserver provides a way how to listen for changes of DOM. When DOM is changed, MutationObserver is called with arguments representing change in DOM. Note that MutationObserver evolved from events of specification DOM Level 3 Events called MutationEvent and MutationNameEvent [23]. MutationObserver will be used for implementing page mirroring (see more in subsection 9.2.4).

2.2 Rich Internet Application

Web pages originally were non–interactive pages using JavaScript for small stylistic changes. This era is often called Web 1.0 [24]. User content was rare in this era and pages were usually hand crafted. Programmers used JavaScript to show and hide menus, dialogs, etc., but content of page remained the same.

Rise of Web 2.0 [25] era brought us Web pages with user generated content. Originally Web page was generated on server and sent to a browser with user content. When user wanted to access another page, browser needed to download the whole page again. Since interaction with DB storing data has began to rise, refreshing all Web resources from server started to be waste of bandwidth. Web developers are now starting to use Asynchronous JavaScript and XML (AJAX) to fetch data from server and change content of browsed page. These pages start to contain more application logic in client side JavaScript. High amount of client side JavaScript results into Web pages behaving more like an application than set of pages. This pattern is called Web application, dynamic Web page, Single Page Application or Rich Internet Application.

Nowadays Rich Internet Applications are mainly HTML 5 based. Other platforms such as Flash or Silverlight are used, but they are on sunset [26, 27].

2.3 Same–origin policy

Client side JavaScript cannot fetch data from any server for security reasons. This restriction is called same–origin policy. This policy restrict access of client base script to potentially malicious documents. [28] By default, scripts can access only the same domain. This policy will affect page mirroring in subsection 9.2.4 as we cannot simply embed page on different domain to our tool.

2.4 WebDriver

WebDriver [29] is protocol for remote control of browsers. Interface of the WebDriver enables other program to perform action, such as going to specific URL or perform user action on Web page. Proposed crawler will use WebDriver interface to interact with the Web page. Controlled browser needs to support WebDriver by implementing the WebDriver interface. These implementations are called drivers. For example for Chrome browser there is ChromeDriver [30], for headless browser PhantomJS there is embedded GhostDriver [31].

WebDriver can provide us with abstraction to use same automation script with variety of browsers. This can be useful when Web page is not properly rendered in particular browser or when Web page is trying to avoid crawling by detecting browser type [32]. Another powerful feature of WebDriver is that it can inject JS code to execute arbitrary JavaScript code in context of Web page [33]. By injecting JavaScript crawler has full access to DOM, therefore any logic for crawler/data extractor can be implemented without modifying commonly used browsers.

3. Other technologies

This chapter covers technologies used during implementation phase of the solution. Used runtime, message queue and database is described in this section. Full decision making process (why we chose these technologies) is described in Part IV.

3.1 Docker

Proposed solution is dependent on several other technologies such as SQL database, message bus or JavaScript engine, which brings challenges with installing and maintaining solution on different servers. To overcome this, crawler components are using Docker runtime to ensure that they will be behaving same across the different servers.

Docker is container–based virtualization tool. Docker containers run as native processes within operating system and they are sandboxed from other processes for security concerns. Docker containers have sandboxed file system as well, which contains dependencies and binaries for the containers. Docker images are definition of Docker containers, which are runnable as containers. Images can be built from Dockerfiles, Dockerfiles are text–based, therefore ideal for development, as they can be versioned in Version Control System. Using Docker helps to reproduce solution's builds and runtime on different machines.

Kubernetes is production ready platform for deploying, scaling and managing Docker containers [34]. Kubernetes will help to manage crawler solution in production environment. Proposed crawler will be composed of several components such as Node.js application or database to persist data. These components will be deployed as Docker containers into Kubernetes cluster.

3.2 RabbitMQ

Proposed solution needs to handle several concurently running crawler instances (more in section 9.3). A message queue is used to share data between these instances and schedule instances. RabbitMQ [35] is popular open–source message broker which is suitable for proposed crawler. RabbitMQ is able to scale to handle more messages in case crawler perfor-

mance demands will raise in future. Message can be persistent and RabbitMQ can be deployed in cluster, therefore it can prevent data losses.

Message brokers can be used to decouple two systems and can handle huge spikes in data quantity. Producer is enqueuing messages to message queue, therefore if producer generates more messages than consumer can process, messages are buffered in the queue and processed later.

3.3 PostgreSQL

Crawler needs to store data about *page models*, *runtime of crawlers* and *crawled information*. Whole data model will be described in section 8.1, but in summary, part of the model is relational and part is schemaless (JSON alike) result from crawler. PostgreSQL is suitable database which can store this type data. *"PostgreSQL is a powerful, open source object-relational database system. It has more than 15 years of active development and a proven architecture that has earned it a strong reputation for reliability"* [36]. PostgreSQL is preferred over other SQL databases due to more maturity of features [37]. There are other SQL databases such as Microsoft SQL Server, MySQL, SQLite, MariaDB, IBM DB2, etc., which would be also sufficient option for crawler.

In this thesis, SQL database will be used for object–relational mapping to provide abstraction on top of the database provided by library Sequelize [38]. A programmer will have an access to objects stored in the database using JavaScript classes, which simplifies the development.

Part III

Analysis

4. Analyzing Web pages

This section will focus on analysis of modern Rich Internet Applications. Such analysis is necessary for understanding of crawling techniques and data extraction presented in chapter 5. To start with, the categorization of Web pages will be presented, followed by the elaboration of method used to interpret Web pages. This chapter will be finished by a case study. First two parts will a help reader to understand, how modern Web pages look like and the last part will give reader an overview about approaches used for crawling.

4.1 Page categorization

Raghavan et al. [39] present categorization, which gives basic overview about Web page types. Authors present categorization of the most common types of Web pages with an analysis of how these categories of pages are crawlable. The article contains a major categories list and despite the fact that modern pages can be in multiple categories, this taxonomy provides useful summary of Web pages to the reader. Authors make categorization of Web pages according to two criteria: type of dynamism and generative mechanism.

Each category of Web pages is crawled by different type of crawler (see Figure 4.1). Note that based on requirements we set in section 1.3, our proposed crawler needs to be crawler for pages with Embedded code (client-side execution) generative mechanism, therefore it is field of no existing crawlers according to Raghavan et al. [39]. Other crawler types will be described in chapter 5.

4.1.1 Categorization based on page dynamism

Content of Web page is usually not the same all the time. Content of the page can vary based on several factors such as time of page load or based on interaction with the page. Following subsection contains description of different types of this page dynamism.

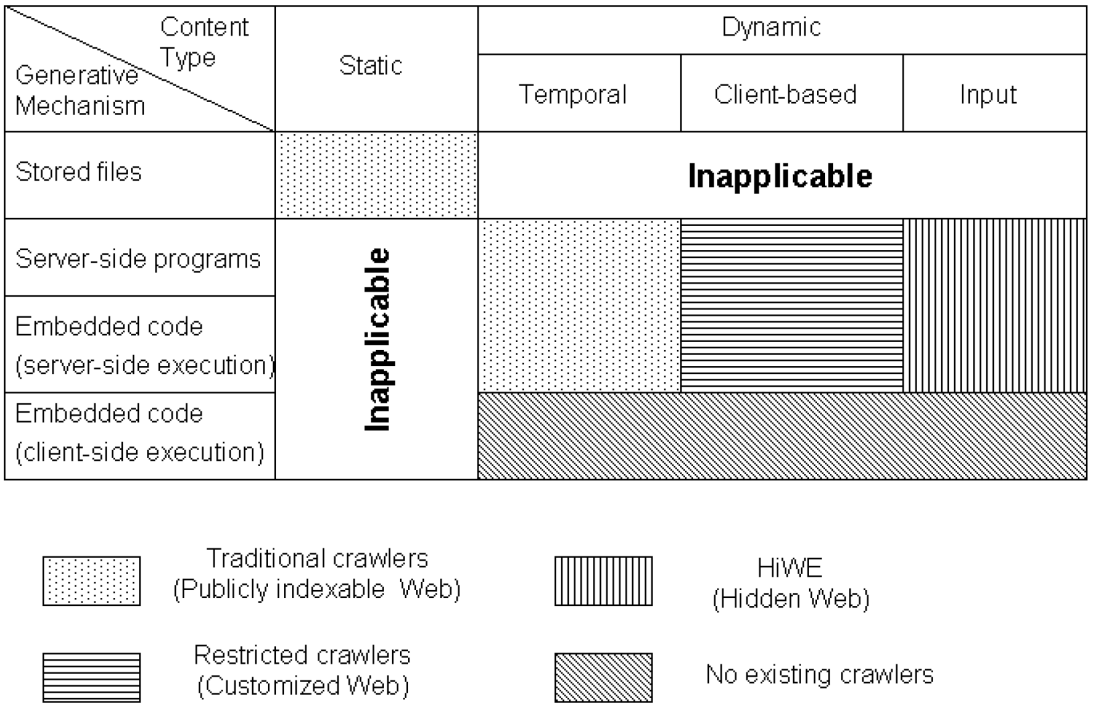


Figure 4.1: *Categorization of crawler according page dynamism and page generative mechanism.* Table is taken from [39].

Static Web pages

Static Web pages do not change after creation. This category of pages is the easiest one to crawl, as crawler need to access the page only once to crawl and crawler does not need interact with the page in any way.

Temporal dynamism

Pages with temporal dynamism are the pages that can vary over time. If page is retrieved in a different time interval, it may show different content. For example, it can be a page showing current stock value or current news. We usually need to crawl these pages several times or infinitely in a loop, to obtain all content of the page.

Client-based dynamism

This is a category of pages, which have content generated per each client. Typical cause of this is page personalization. This can be achieved for example by analyzing user cookies. The problem with pages with client-based dynamism is that crawlers usually start crawling with no stored cookies,

therefore the crawler might not obtain personalized content and it cannot obtain all content from Web page.

Input dynamism

The content of some pages is dependent on submission of a form. Typical example is search engine, which is showing content according to the input query. Typical problem with crawling of these pages is to determine what values should be filled in the form to access right content. Another example is the user login. The crawler needs to have stored user credentials in order to log as a user and obtain data from the page.

4.1.2 Categorization based on generative mechanism

The way how pages are generated change techniques of crawling. There are two main ways how to generate a page: on the server or in the client environment. The server–side generation is splitted into more categories, based on generative process. In the following subsections we will focus on these categorization in more detail.

Stored files

Static HTML files are stored on server. They are provided to client and they are not changed in any way. Crawlers can simply load the HTML document to obtain all content of the page.

Server-side programs

Server–side program generate all content of Web page on the server. This is typically done by Common Gateway Interface scripts (CGI), Java Servlets or similar technologies. These pages may not be HTML documents, but they can be simple plain text documents. Usage of server–side program was popular in the past. Nowadays this approach is usually not used for Web pages themselves, rather for their underlying HTTP API. API is used for accessing database from underlying page with client–side executed script.

Embedded code with server-side execution

The server contains static HTML files and embedded code snippets in various script languages. Difference from server–side programs is that only part of page is generated, not this page as a whole. HTML–like template is usually used for basic layout of the page and data are injected into this template. The crawler needs to load only HTML page to obtain its content. These pages are input, client-based or temporally dynamic, therefore crawler needs to obtain the page several times or crawler needs to submit forms to obtain all content of whole website.

Embedded code with client-side execution

The server provides static HTML template with embedded (or linked) code. The code is executed in client (browser) environment. Scripts can be written in JavaScript, developed as Java Applets, ActiveX, etc. These technologies raise new challenges as crawler need to interpret them. Based on goals set in chapter 1, we will focus only on HTML5 and JavaScript based pages in this thesis. The crawler needs to interpret Web page in browser, in order to let client-side script fill page with data.

4.2 Web page interpretation

Technologies used for modern Web pages were discussed in chapter 2. This section focuses on different tools used by crawler in order to obtain all content from the Web page. Although users interact with the Web page by using the browser, crawlers use different suite of tools to interact with Web pages. For example HTTP programming, DOM interpretation or browser automation.

4.2.1 HTTP programming

Web pages are served on top of HTTP protocol. Crawlers can use this fact to download Web resource via HTTP. Crawlers which are using this approach are usually focused only on downloading Web resources with content – HTML documents. HTML documents obtained by using this method need to be processed to extract relevant data only. These documents often contain menus, buttons, advertisements, visual elements such as background images and other elements, not containing any useful information. Extraction of relevant data can be achieved by using regular expressions,

interpretation HTML document as XML, together with using XPath to retrieve relevant elements. It can be achieved by program with custom logic written in general purpose language as well. Programs which use this approach are GNU Wget [40][1], curl [42] or any HTTP client library of any programming language.

Solutions based on HTTP programming works only on pages without embedded code (client–side execution). This happens because they do not interpret JavaScript. These solutions are usually fast, as they do not implement DOM, but they are limited only to subset of Web pages.

4.2.2 DOM interpretation and JavaScript execution

More advanced approach than HTTP programming is to interpret HTML as DOM. When DOM APIs (more in subsection 2.1.2) are present, JavaScript script can be executed to change Web page in the client environment. Libraries using this approach such as JSDOM, implement DOM API on its own. They are also using own JavaScript engine to interpret client scripts. In case of JSDOM, which is Node.js library is for implementation of DOM API and for interpreting client–side script used internal Node.js JavaScript engine. Other JavaScript engine can be used, such in case of libarary HtmlUnit, which is headless Java library for Web page interpretation. It implements subsection of DOM API and uses JavaScript engine Rhino [43].

However, JavaScript execution is complicated as these libraries do not implement all methods from DOM API (for example part of navigation API is usually missing [44]). Cookies are usually not handled as well, therefore user scenarios such as cookie-based login are not possible.

These solutions do not layout Web page as well, therefore they cannot access Web page's visual information such as size and position of elements. With no information about page layout, they cannot take screenshot of elements or Web page as a whole. Some of visual crawling techniques are not possible with this page interpretation. Another downfall of this approach is that Web developers commonly optimize JavaScript code to work with major Browser and JavaScript code may not execute properly.

In general, this approach is slower and more resource demanding than HTTP programming, however it can handle Web page with simple client–side scripts.

[1] Wget has embedded crawler, which can be used for crawling static pages out of the box [41].

4.2.3 Browser automation

The most universal approach of Web pages interpretation is to use Web browser during crawling. Browsers can be automated using different mechanisms. One way to automate browser is to use WebDriver protocol (more in section 2.4). For every Web browser, there is program called driver, which implements WebDriver API for that particular browser. An example of driver is ChromeDriver [30] which is implementation of WebDriver API for Google Chrome. Drivers automate browsers as they are, therefore they render browsed page on the screen. Rendering of the Web page on screen is not necessary for crawlers and it consumes computer resources such as memory and processing time. Headless browsers overcome this issue, by not rendering anything to screen. PhantomJS is popular headless browser built on top of WebKit [45]. It provides JavaScript API to interact with browser and it has also embedded GhostDriver which implements WebDriver API.

Although WebDriver is a popular way to automate browsers, there are other option how to automate them, such as Puppeteer for Google Chrome. This browser provides native JavaScript API for browser automation called Puppeteer [46]. This is relatively new solution, compared to PhantomJS, but it is getting quite popular, as it is maintained by Google company and PhantomJS development is freezed [47].

The benefit of browser automation approach is that crawler has access to full scale DOM implementation. Therefore crawler and Web extractor can select elements by CSS selector, XPath, determine position and size of elements, inspect elements visually (for example make screenshots), and so on. JavaScript is executed in browser, therefore Rich Internet Application based on HTML5 and JavaScript can be processed.

4.2.4 Web interpretation summary

Based on requirements set in chapter 1, proposed crawler needs to automate browser in order to fully support RIAs. PhantomJS is used in this thesis, because it is mature technology fully supporting RIAs. Other approaches such as HTTP programming would not obtain data from RIAs correctly.

Practical difference between HTTP programming and browser automation can be found in work of Brunelle et al. [48]. Authors used headless browser to download Web resources when crawling. With this approach, browser executed JavaScript, therefore it was able to download data from any custom HTTP calls. They measured that while using HTTP programming (Her-

itrix crawler [49] and wget) they missed 19.70 times more content per URI, then if they used browser. On the other hand this solution was 12.13 times slower. This result shows that using browser is an approach, which obtains most of the data from Web pages, but it is the slowest one.

In the following section we will present an example of typical modern Web pages together with validation of different Web interpreting techniques.

4.3 Case study of Web pages crawling

In the previous chapters, this thesis was working with the premise, that there are RIAs containing useful information. In this section, selected Web pages and their case studies to confirm that RIAs are worth crawling will be presented. Architecture of each Web page and elements structure of the page will be described. This knowledge let us define strategy for extracting elements from these Web pages. To set baseline for this strategy, we will define CSS selectors, which choose important information from the Web page. These CSS selectors will be used to create heuristic algorithm defining them visually (more in subsection 9.2.3). We selected RIAs and static page as well, because our crawler needs to crawl both RIAs and static pages to be universal.

Note that if you would like to crawl these pages, you should first read terms and conditions of the site. Authors of particular page can prohibit crawling of their data. More about topic of legality and ethics of crawling can be found in Appendix A.

4.3.1 Bucharest stock exchange

Bucharest Stock Exchange website [50] contains current information about the exchange rates and various news related to companies, which are part of this stock exchange. This website is typical example of source data for further analysis. As we can see on Figure 4.2, Bucharest Stock Exchange website consists of press releases published by stock exchange media division. These releases are useful source of information, which can be used for business analysis of these companies. Such analysis could extract entities (such as company names) from the releases and shows mentions of the company over time, which would give insight about company. Another possible application of crawler is to periodically crawl releases and notify user when new press release about specific company is released. This will save time to the user, because he/she does not need to check press releases for new information and it will provide competitive advantage, because infor-

mation about the company can be retrieved faster than accessing website as regular user.

Custom crawler which would download press releases would have following structure:

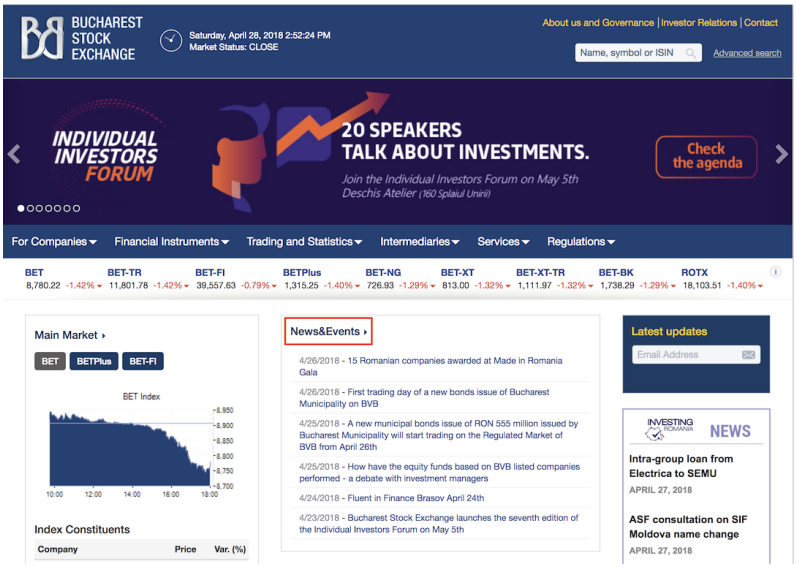
1. Go to `http://www.bvb.ro/AboutUs/MediaCenter/PressReleases`.
2. In order to choose years of release, click on button with CSS selector `button[data-id=ctl00_ctl00_body_rightColumnPlaceHolder]`.
3. Click on all anchors with CSS selector `div.dropdown-menu.open a` in order to visit all pages with press releases.
4. Download document which is referenced by anchor identified by CSS selector `a.title-comunique`.
5. If there is no element with CSS `a.paginate_button.next.disabled` click on element with CSS selector `a.paginate_button.next` and go to step 4.

Crawler will download press releases as documents, because there are not only HTML documents (PDF, Excel documents, etc.). During runtime of this custom crawler URL of Web page remains the same. This fact implies that Bucharest stock exchange is RIA and client-side script needs to be executed in order to access all content.

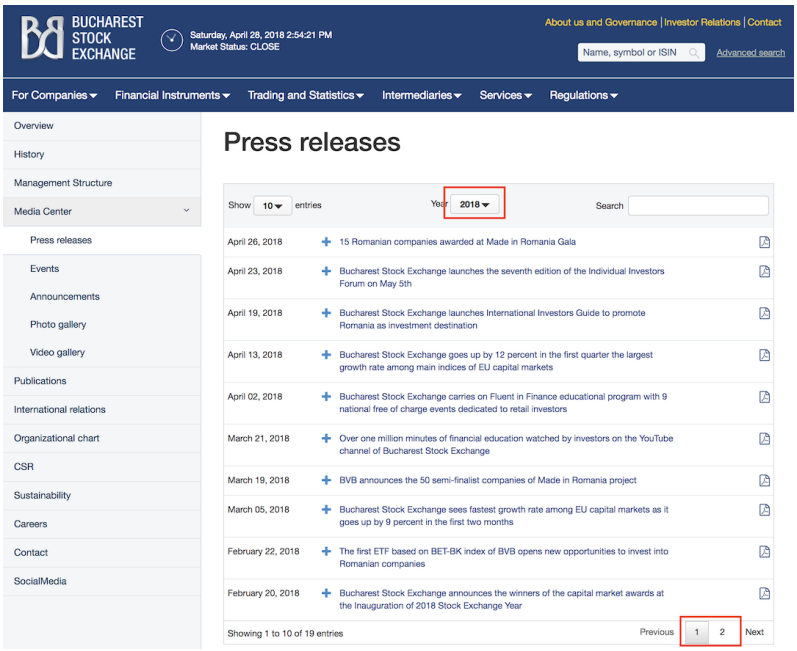
From prespetive of categorization in chapter 4 this is page with client–side embedded code and temporal content dynamism. Therefore if we want to crawl this Web page, we need to use client, which execute embedded code (browser) and crawl the page periodically to obtain new data. To handle pagination, we need to execute command to select all years and visit all pages.

4.3.2 Bezrealitky

Bezrealitky website [51] is providing platform for advertising real estate offers. Bezrealitky provides a wide range of houses and flat profiles dedicated for purchase or rent. It provides the user possibility to view property for sale of rent according to selected criteria such as location, price or living area size. These information can be used for further analysis on real estate market. Such analysis can indicate actual average price of real estate, which can be used for making decision during purchase of a flat [11].

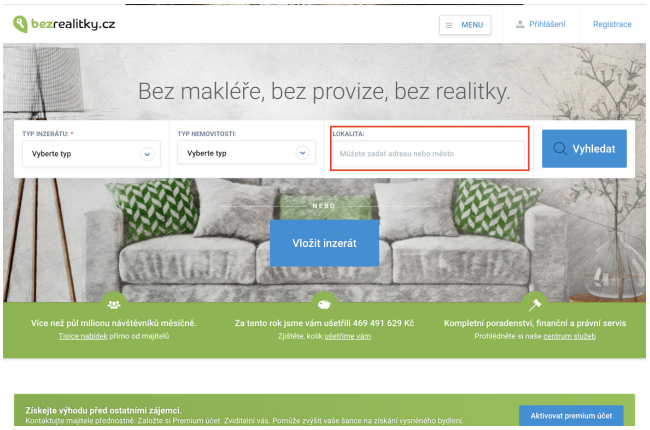


(a) Stock exchange homepage

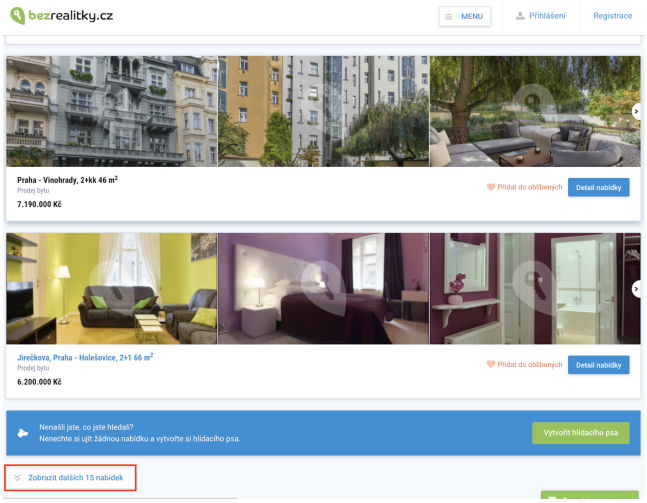


(b) List of press releases

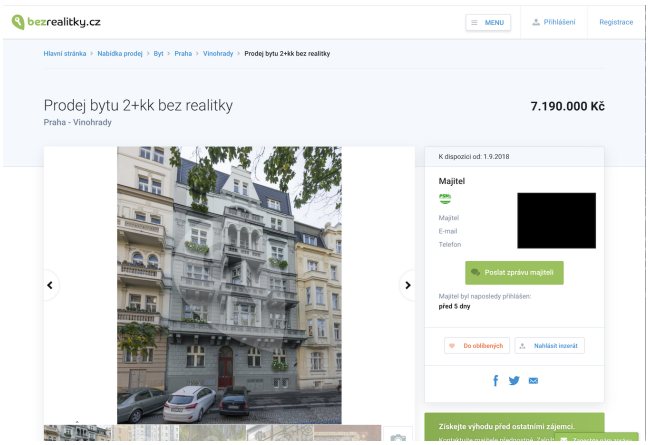
Figure 4.2: *Bucharest stock exchange.* Highlighted element on subfigure (a) is link to page with press releases. On subfigure (b), there is page with press releases from stock exchange office. Highlighted elements are buttons which show press releases for particular year and in case there are more than ten press releases on the page, they control pagination.



(a) Homepage with search form



(b) List of flats to buy



(c) Flat to buy detail

Figure 4.3: *Bezrealitky.cz* Highlighted element on subfigure (a) is search form for finding real estate in particular ares. On subfigure (b), there is list of found real estates in the area. Highlighted button on left bottom part of page shows loads more results. Subfigure (c) shows detail of flat offer.

The home page contains search form for searching offers (see Figure 4.3). Figure 4.3b shows list of first fifteen offers returned for search. For obtaining more searched offers, crawler needs to click on button *Zobrazit více nabídek*, which load more offers to Web page. URL in this case remain same, which indicate that this page is RIA. Each offer page (Figure 4.3c) has unique URL and it is showing information about offer and photograph of offered flat/house. To obtain all photographs, crawler need to click on arrows, whick load other images on same page without changing URL. This suggest that page is RIA as well.

Custom crawler, which obtains price of each offer in particular area:

1. Go to `https://www.bezrealitky.cz/`.
2. Fill location of requested offers into input identifiable by CSS selector `input#location.form-control.form-control-md`.
3. Click on button `button.btn.btn-primary.b-intro__submit` to perform search.
4. Follow link with CSS selector `a.product__link.js-product-link`.
5. Click until there are new result on the page on button with CSS selector `span.icon-svg.icon-svg-double-arrow`.
6. Obtain offer price, which is inner text of element with CSS selector `p[data-fancybox-price]`.

To handle pagination, crawler needs to click repeatedly on button *Zobrazit více nabídek* and for obtaining all images of the offer, crawler needs repeatedly click on arrows on image.

From prespetive of categorization in chapter 4 this is page with client–side embedded code and temporal content dynamism. Therefore crawler needs to use client, which execute embedded code (browser) and crawl the page periodically to obtain new data.

4.3.3 Bloomberg

Bloomberg [52] is news agency, which is publishing and reselling news articles to other news agencies. They post some of their technology realted articles on website `https://www.bloomberg.com/technology`, which is typical example new agency website. Published articles can be interesting source of information for finding what are the hot topic in technology (with

help of entity extraction algorithms). Other application of articles automatically crawled from news agency website could be automatic alerting on a company name appearance in news.

News agency websites commonly contain two types of page: homepage with thumbnails of articles and separate page for every article. On homepage of Bloomberg website (see Figure 4.4) there are many thumbnails of articles with different visual styles. Different visual styles typically means, that HTML elements will have different CSS classes, therefore extracting CSS rules is harder as more than one rule needs to be defined. This is typical issue with these kind of websites. Bloomberg homepage has a lot of animated (interactive) element, however all are advertisements and initial HTML contains all link to articles and articles can be access by URL. This mean that page is generated on the server.

Custom crawler, which obtains news articles:

1. Go to `https://www.bloomberg.com/technology`.
2. Follow all links with `a.highlights-v6-story__headline-link` and `a.story-package-module__story__headline-link` CSS selector.
3. Download page with article.

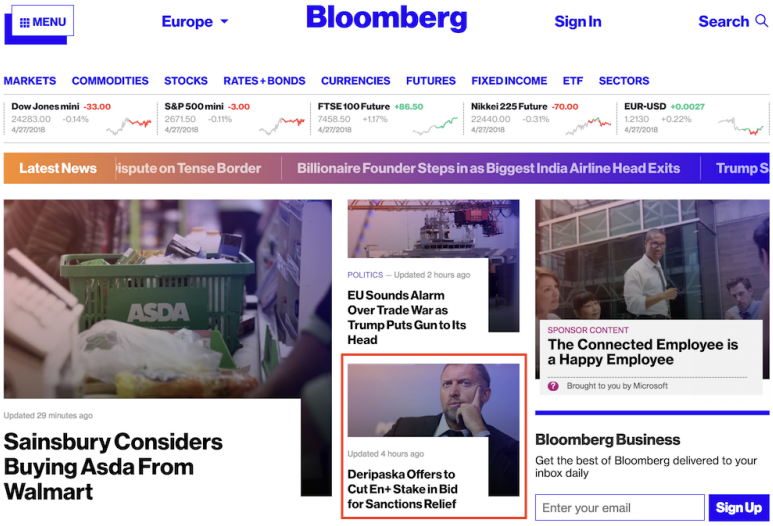
Notably on home page there is no easy way to select all article thumbnails using single CSS. For illustration we presented only two CSS rules. Page is with embedded client code, but effectively it can be treated as server code as all data are generated on server. Therefore crawler can use HTTP programing. Web page is changing in time, therefore is temporal by presented categorization.

4.3.4 Sbazar

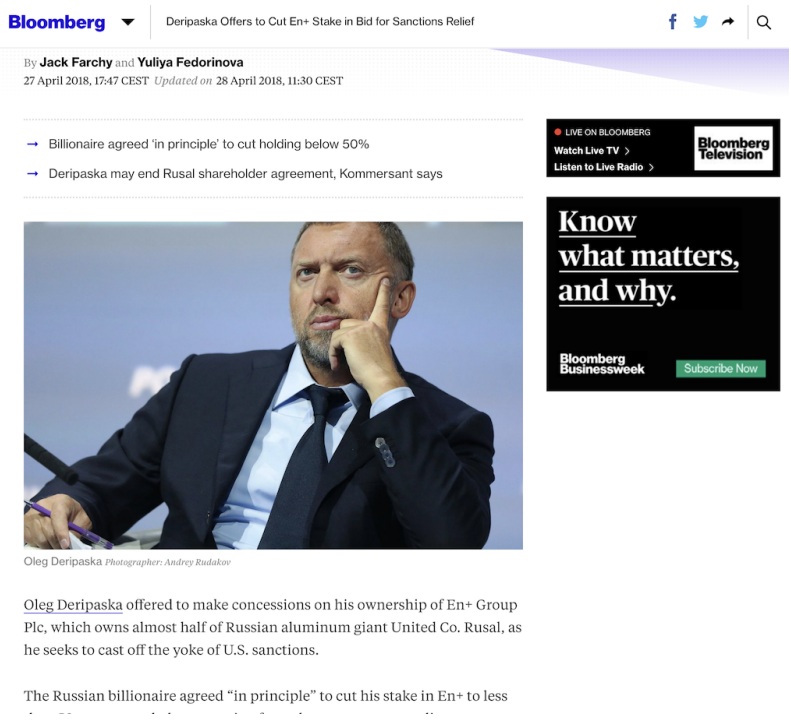
Sbazar [53] is the site for advertising personal advertisements. User's of sbazar advertise items, which they want to sell to other users. If this data are crawled, they can be used for alerting and price analysis of sold items.

Website structure is very simple. Website (Figure 4.5) shows list of advertised items on landing page. Each item has separate detail page. Landing page offers to the user full text search or filter particular category.

This website is static Web page. In theory, it can be crawled using current static Web page crawlers, however proposed RIA crawler should be able to crawl as well to be universal and hide technical details from user.

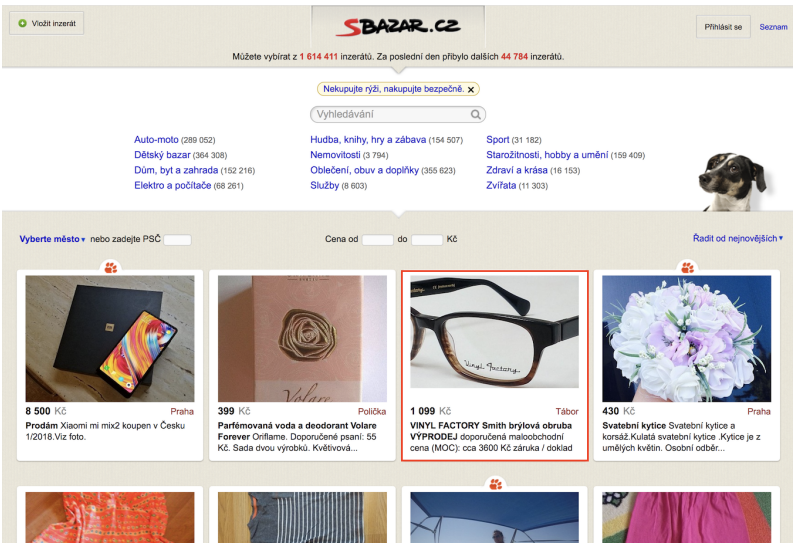


(a) Homepage

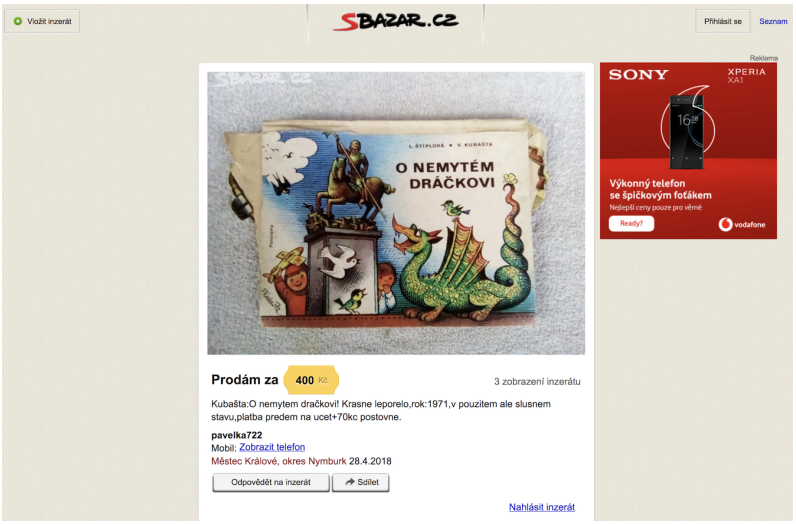


(b) Article detail

Figure 4.4: *Bloomberg* Subfigure (a) is showing homepage of the website. On this page there are thumbnails of articles. On figure (b) is an article in detail view.



(a) List of items to sale



(b) Item detail

Figure 4.5: *Sbazar.cz* On subfigure (a), there is list of advertised items to sell. Highlighted element is thumbnail of one particular item. Highlighted button on left bottom part of page shows loads more results. Subfigure (c) shows detail of adverised item.

Custom crawler which extract price and name of advertised items:

1. Goto `http://www.sbazar.cz/`.
2. From inner text of element with selector `span.c-item__name-text` extract item name.
3. From inner text of element with selector `b.c-price__price` extract item price.
4. Click infinetely on link `a.atm-button.c-prev-next__link` to access all pages.

From perspective of our presented categorization this Web page is generated on server (server–side embedded code) and it is temporal dynamism as advertisements are created by users continuously and it is input dynamism as there is full text search capability, when searching for advertisement. Therefore crawler needs to crawl this page periodically to obtain fresh data and HTTP programing methods can be used as HTML is generated on server and it contains all data.

Note that `http://www.sbazar.cz/` gone complete graphical redesign during time of writing this thesis. Page accessed nowadays look different, than on Figure 4.5. Other implication is that CSS rules defined above, will not work for crawling anymore. Crawler needs to be updated in order to crawl redesigned Sbazar page. This process is commonly called crawler adaptation and will be discussed in chapter 16.

4.3.5 Amazon

Amazon [54] is an e–commerce platform used by third party sellers to sell their products on–line. It contains enormous number of advertised items. These items have price, description and customer's comments. Amazon advertises today's deals, which are item in special sale – today's deals will be

These data can be used for example for competitive analysis or crawler extracting today's deal can alert user about interesting deals. User comments are good source of data for sentiment analysis to determine which products are well received by customers.

Homepage of Amazon website (Figure 4.6) shows disambiguation of items categories, so user can choose only category of items he/she is interested in. On homepage there are also search bar for full text search on specific item

on Amazon page. For case study purpose, we choose analyze today's deals. Page with today's deals can be opened by button in top menu on homepage. Today's deal page show list of today's deals including item name, price and sale.

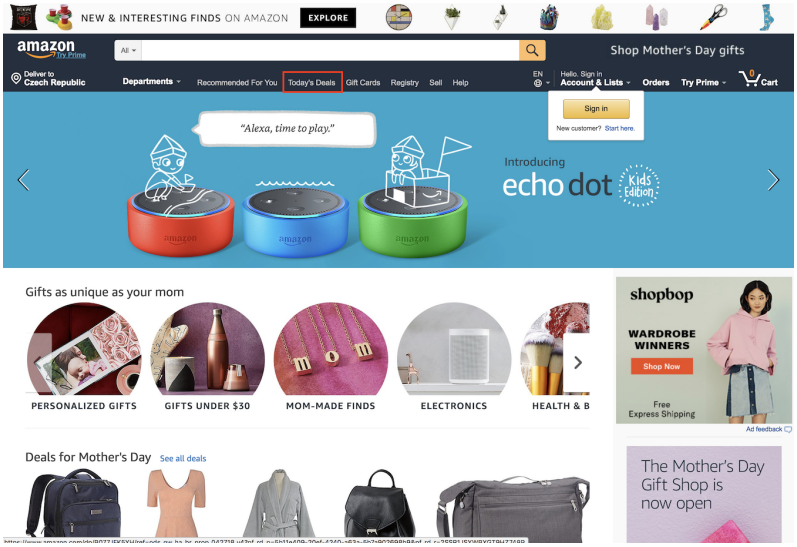
Custom crawler, which extracts today's deals from Amazon would look like:

1. Go to `https://www.amazon.com/gp/goldbox/`.
2. Extract item price from element identifiable by following CSS selector `span.a-size-medium.a-color-base.inlineBlock.unitLineHeight`.
3. Click indefinetely on element with CSS selector `a#next` to access all pages with items.

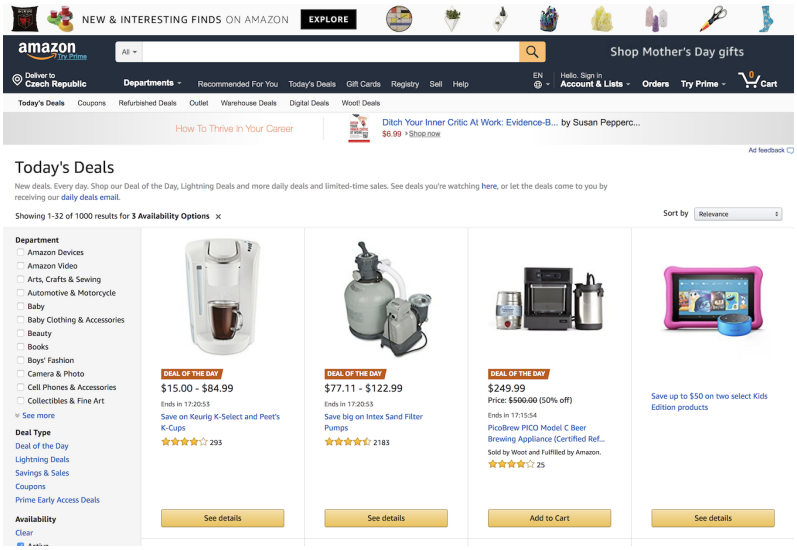
Amazon Web page has relatively complicated CSS rules. HTML document with today's deals contains all data, however all data are inside `<script>` tag. This script injects the data into DOM after page is loaded in browser. Given categorization from subsection 4.1.1 this Web page is generated on server (server–side embedded code) and also processed on client (server–side embedded code) it is temporal dynamism as advertisements are created by users continuously and it is input dynamism as there is full text search capability, when searching for shop item. Therefore crawler needs to crawl this page periodically to obtain fresh data and DOM interpretation or browser automation methods are need to be used. HTTP programming techniques can not be used for scraping, as JavaScript needs to be executed to extract data from HTML. DOM interpretation techniques with JavaScript execution can not be used as well, because Amazon has no obfuscated URL, therefore crawler can not easily obtain URL of next page with items. Example of such URL is below:

```
https://www.amazon.com/gp/goldbox/ref=gbps_ftr_s-
4_d724_page_4?gb_f_deals1=dealStates:AVAILABLE%
252CWAITLIST%252CWAITLISTFULL%252CEXPIRED%252CSOLDOUT%
252CUPCOMING,includedAccessTypes:GIVEAWAY_DEAL,page:
4,sortOrder:BY_SCORE,dealsPerPage:32&pf_rd_p=695f29ac-ec28-
4005-ae23-4a6ff667d724&pf_rd_s=slot-4&pf_rd_t=701&pf_rd_i=
gb_main&pf_rd_m=ATVPDKIKX0DER&pf_rd_r=4WJ81ASEPJAF0PDNMPTS&ie=
UTF8
```

Although there is part of URL `page_4`, crawler cannot generate URL for page 5 simply by incrementing URL. Obfuscation mechanism of Web pages is preventing that. We assume that other parts of URL needs to be change in order to go to next page. However we do not know which one.



(a) Amazon homepage



(b) List of today's deals

Figure 4.6: *Amazon* On subfigure (a) there is Amazon home page and on subfigure (b) there is page with today's deals.

4.4 Case study summary

The intention of this study was to show that there are RIAs with useful information for the business users. Several pages with description of the data usage was presented to give reader idea, how modern Web pages look like. There are more of Web pages, however we choose only these five examples presented in this section to show that RIA crawler is needed.

Case study showed how CSS is structured as well, so reader has better view on how are CSS usually structured. It will be used during designing algorithm, which interfere CSS selector from selected element on screen. This algorithm will be essential in our crawler as it will allow user extract data visually (more in subsection 9.2.3). CSS selector are not only way to select relevant elements on the screen, however given fact that CSS selectors in this study could express all extracted elements, gives us confidence, that we can use CSS selector for data extraction.

5. Web crawling and Web data extraction

In context of Web page classification presented in previous chapter (nicely illustrated in Figure 4.1, proposed crawler will focus on crawling page with embedded code (clien–side execution). Regarding to work Raghavan et al. [39] this are is labelled as “no existing crawlers”. Our solution is required to be universal (as stated in chapter 1), therefore crawler will handle static page and server–side embedded code generated page with temporal dynamism (Raghavan et al. refers it as “traditional crawler”). Page with client based dynamism will be handled as well (restricted crawlers). Only category our crawler will not handle is “hidden web”, these page are required user input. Proposed crawler can be submit input, however user needs to define exact values of this input.

To satisfy requirements stated in section 1.3, that proposed tool should extract data from the Web pages without overloading users technical with technical details, crawler needs to navigate Web pages as well. Therefore proposed tool will be a hybrid system between Web data extractor and Web crawler. Different approaches of crawling and data extraction from Web pages will be presentend in this chapter to let us select the most suitable approach for our crawler. Web data extractors categorization based on work of Laender et al. [55], Ferrara et al. [2] is following this section and crawler categorization based on work of Choudhary et al. [56]. This categorization will complete full view on this topic.

5.1 Web data extraction categorization

Web data extractor is program used to extract information from Web page. Data on the page are usually not provided in structured form[1], but more often as a HTML–like structure, which is rendered to a visual form to be readable by human. Web data extractor is used to mine data from Web page in structured way. Typical example of usage is e–shop page with a list of items – Web data extractor is used to extract all items and its metadata in format, which can be used by other systems. Another problem Web Data Extractors are solving is how to extract only important part of the page. Web page often consists of advertisements and menus, which are not intend to be extracted.

[1] Meaning that information has predefined structure.

Ferrara et al. [2] divide Web data extractors to three main categories: Web wrappers, tree–based techniques and hybrid systems. We believe this categorization is good for understanding all techniques.

5.1.1 Web Wrappers

Web wrapper is a procedure implementing any algorithm, which extract unstructured or semi–structured data from a Web page into structured format to provide a way to further process these data in other systems [2]. Due to this vague definition, Web wrappers applies for all mentioned types of Web pages, however they usually need to be hand crafted to specific site.

Web wrappers are created in many different ways. The most simple approach is to manually write a program in some Generic Purpose Language or Domain Specific Language [57, 58]. Advanced techniques will be mentioned in next subsection. When wrapper is created and Web page is changed (new version of page is deployed or layout of page is changed) wrapper may stop working. This problem is called wrapper resilience and adaptation and will be elaborated in the last subsection.

Wrappers generation

Wrapper can be created from labeled sample of Web pages using induction algorithm [59]. With this approach, there is no need for developer to create wrapper, however a labeled data set is needed to be created.

Annotated Web samples can be obtained automatically based on simple rules like user defined regular expression. Using these rules is not accurate method and it is generating many false results. These inaccurate data are used as labeled data set, which with a noisy tolerant induction algorithm is used to generate wrapper [60]. This approach reduces work needed to implement wrapper in contrast with wrapper implemented in GPL.

Another approach how to generate wrapper is based on visual information from Web page. These wrappers can exploit visual information (size, color and style of text) extracted from page to increase accuracy of extraction, when it automatically clusters Web elements [61, 62].

Wrapper execution and maintenance

When Web wrappers are generated or inducted, they can extract data by executing itself on extracted Web page. In case that structure of the Web page remain same, wrapper will return relevant data. In case Web page change its structure, executed wrapper can return irrelevant data. Avoiding this outcome of page change is called Web wrapper adaptation. In our best knowledge best approach to adaptation is make wrapper generation noisy tolerant.

5.1.2 Tree–based techniques

Tree–based techniques are exploiting tree structure of Web pages. Typical example of these techniques is using addressing elements by XPath. Limitation of this approach is that XPath only selects elements in Web page HTML tree and it does not represent transition in page state (clicking on button, filling form, etc.). Extension of this approach is OXPath [63] with support for user interactive actions and more stable selectors of page elements. Another extension of OXPath is designed for handling collections of elements – Kleene star operator for extraction sets of data is provided.

Another approach is to take HTML tree structure with labeled data to extract and partially align it with extracted Web page HTML tree [64]. Alignment is achieved by comparing minimal tree edit distance. Jindal et al. [65] are addressing problem of tree matching of elements containing nested list. Nested list can lower matching score due the variable length. Authors are proposing enhancement of tree matching algorithm by grammar generation for the tree.

Last mentionable approach of tree–based techniques is system proposed by Zhang et al. [66]. Authors propose method which take presumption, that modern Web pages are generated using only a few templates. Using only tree structure method generate wrapper and distinct between different templates.

Tree–based techniques can be used on modern Web pages, as they are commonly using HTML templates.

5.1.3 Hybrid systems

These techniques are also called techniques of learning based wrapper generations. In recent literature they are differentiated from Web wrappers as

they differs in two aspect: degree of automation and amount of human engagement.

First example of hybrid systems is template–based matching. This approach is based on idea, that modern pages are generated from template. It takes at least two pages generated by same template and induct wrapper from similarity of templates.

Another examples are systems exploiting spatial reasoning. These techniques exploits computer vision to identify data for extraction on the page. Machine learning can be used to identify what needs to be extracted on prepared human annotated data set. This approach is used in work of Gogar et al. [67], authors are using convolutional neural networks to incorporate visual and spatial information to create universal Web wrapper for text extraction.

These hybrid systems can be used on modern pages, however they are usually crafted only to specific types of Web pages (e–shops, forums, etc.).

5.2 Web crawling

Web crawler is program designed to retrieve complete Web pages based on given seed (starting page). Commonly, crawlers are traversing pages based on hyperlinks. With rise of RIAs, crawlers need to also traverse by interacting with page elements such as buttons, which trigger JavaScript action, which may traverse to other page or load new data. This fact brings new challenges to this area (more was elaborated in section 1.1).

This section is focused on RIA crawlers. Choudary et al. [56] divide RIA crawlers to two categories: general RIA crawlers and model–based crawlers. We believe this categorization is good for understanding all possible techniques.

5.2.1 General RIA crawler

General RIA crawler dispatchs all possible actions to visit all states of the page. Duda et al. make search application for crawling dynamic Web [68, 69]. They defined Ajax crawling problem and challenges. Main problem is that crawler cannot say if states are equal, therefore crawl visit one state many times. They using caching of resources and removing duplicated states based heuristic to optimize crawler performance.

Another example of general RIA crawler for inspiration is work of Mesbah et al. Authors propose crawler using DOM deltas and creating state machine. The solution can be run concurrently and it is scriptable. Solution is called Crawljax [70, 3].

5.2.2 Model–based crawler

Choudary et al. discuss approaches to determine state equivalence which is very important for crawler performance [71] of RIA crawler. Another challenge is optimize how many time crawler needs to reset to initial state during crawling. This is referred as reset cost [56]. Model–based crawlers is category of RIA crawler which are optimizing performance by making assumption about crawled page structure (model).

First model Choudary et al. defines is probabilistic model. Authors defines probabilistic model as set of all dispatchable actions, which are dispatch with particular probabilities. These probabilities are set at default value during crawler initialization and they are adjusted during crawler runtime in order to reflect how many new states can particular action discover.

Choudhary et al. uses probabilistic model with heuristic information about typical Web application pattern to increase speed of visiting all application state. Pattern is called “menu” and it is based on observation that a lot of events in different state of application direct in same state [72]. For example when page has a menu with categories “home” and “about us”, clicking on any of these categories ends in that category nevertheless, in which application state Web page is. Another model is “hypercube”. This model is based on hypothesis that events can be dispatched in any order to explore same set of states.

6. Analysis summary

Page case study in last chapter showed that there are RIAs containing data worth to crawl. Study demonstrated that these data can be extracted using CSS selector, but these CSS selector are too complicated to manual creation for non–technical user. To overcome this limitation, crawler will use algorithm, which generate this CSS selector for user. Algorithm will be presented in subsection 9.2.3.

In Web crawling and Web data extraction techniques categorization, different approaches was presented to let reader understand, what are nowadays common approaches for Web crawling and extraction. Proposed solution will be hybrid approach of crawler and data extractor to satisfy requirements declared in section 1.3. Our proposed tool will be Web wrapper using user generated CSS selector and it will be model–based RIA crawler. CSS selectors and model will be defined by user. This will comply with requirement, that user which domain knowledge about page will have opportunity to extract data he/she want to extract.

Part IV

Design

7. Solution design

In first part of this thesis (chapter 1), there was introduced an idea of universal crawler, which let user to crawler Web page regardless if page is RIA or static page. In previous part, modern Web pages were analyzed and theoretical approaches to crawling and data extraction were described. Appropriate approach of crawling modern RIA and static pages was chosen based on requirements set in section 1.3.

In this chapter, there will be propose crawler solution, which fulfill stated requirements from first part of the thesis. First, there will be defined user roles and basic functionalities will be described based on requirements from section 1.3. User interface will be designed and solution will be decomposed into basic components.

Data model and architecture of proposed solution will follow in following chapters. Implementation will be described in Part V.

7.1 Roles

In order to design solution, users and their roles needs to be understand. Three major roles interacting with the system are end user, administrator and developer.

End user

The most important user of the system. He/she has business needs for obtaining data from the internet and he/she will use crawler tool to retrieve such data[1]. This user will use crawler to make page model. End user will be able to run crawler and export results. End user has no specific knowledge of modern Web technologies, but he/she can use browser to access Web pages in his/her daily life.

System administrator

System administrator is user of the system, who is maintaining the system and its infrastructure. Administrator needs to be able to deploy and setup

[1] In other words, end user knows the domain of crawled page.

system in production environment. Administrator will be able to monitor system and scale components if needed. Administrator needs to have possibility to inspect running crawler in case of error (end user can not debug crawler itself as he/she has no technical background).

Developer

Developer is a user who will maintain the code base and extends the solution. Because of time constraints of master thesis, this solution will be extended in future. Selection of developer as recognized role is important in our opinion, because we believe, that active development effort will be needed after finishing this thesis to reflect change of Web and provide future optimizations.

7.2 Solution overview and functionalities

Because crawler needs to let user define *page model* visually (requirement FR2), crawler needs to be Rich Internet Application (trait T1). RIA front end will also supply other requirements for managing and troubleshooting crawlers.

Conclusion of last chapter was to use browser automation in order to crawl RIAs. Automated browsers will be controlled via WebDriver protocol (trait T2). WebDriver protocol will decouple crawler from browser, therefore multiple versions of browser can be used to crawl Web pages. Cloud based clusters of browser can be used as well. Separation of browsers can be convinient in case, crawler needs to rotate an IP address to avoid geo–blocking or other networking issues.

Solution is aiming to provide universal way of crawling Web pages by creating model of specific page. This *page model* is configuration for crawler how to mine data from Web page. Typical user browses page in linear order, therefore *page model* should be defined as sequence of commands[2] (trait T3).

This *page model* is defined visually by user of the system. Commands in page model interact with page elements. User will define these elements by selecting them using mouse pointer. WebDriver can select elements only by CSS selector or XPath, therefore *page model* needs to use these selectors.

[2] Commands are high level action such as clicking on Web pages, filling forms, extraction text from Web page, etc. Commands will be defined in following sections.

These selector will be althought defined visually (trait T4). This choice let user define selector as CSS selector manually.

Crawling larger pages is very computing resources expensive operation (every browser requires a lot of CPU time and memory). Crawlers should be able to run among more browser instances (trait T5).

Extracted data by crawler should be further processed in other systems. System must be able to export extracted data as file in CSV or JSON format. System needs to be able let other system use extracted data using shared SQL table. CSV is convenient format for importing data to variety of tools such as Microsoft Excel. Majority of business inteligence tools can be connected to SQL database. JSON format is popular to exchanging data between Web application. These types of data export will satisfy the most of integration scenarios with other systems (trait T6).

Summary of desired functionalities follows:

T1 *System should be Rich Internet Application.*

T2 *System should be able to run crawler on own infrastructure (for example single PhantomJS [73]) and on infrastructure provided by cloud providers using WebDriver protocol.*

T3 *System must be able to let user define page model as sequence of commands.*

T4 *System must be able to let the user define page model using CSS selectors or XPath. Commands in page model interact with page elements. User will define these elements using CSS selectors or XPath.*

T5 *System should be able to parallelize crawler between more browser instances.*

T6 *System must be able to export extracted data as file in CSV or JSON format. System needs to be able let other system use extracted data using shared SQL table.*

7.3 Basic decomposition of solution

This section describes decomposition of solution to smaller component. This decomposition will help to establish terminology and split functionalities according to single responsibility principle.

Solution is RIA, therefore client–server architecture. We will use term client as front end and term server as back end in rest of the thesis. In high

level decomposition of the solution there are four main components in this solution. These components are front end, crawler store, crawler runtime and session manager (see Figure 7.1).

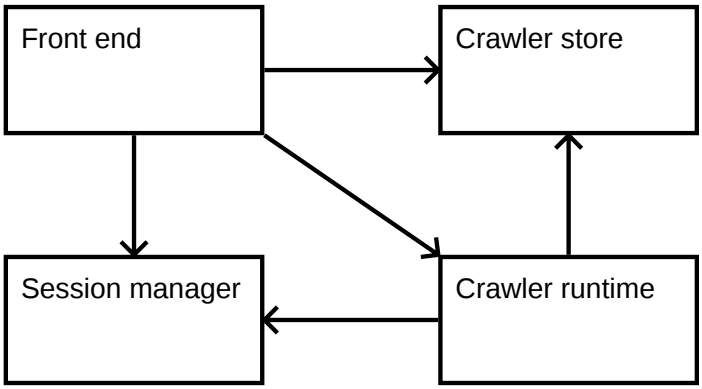


Figure 7.1: *Component model.* On diagram, there are components of the system. Big rectangles represent component. Line between components means that components are communicating (share information) with each other.

Front end

Front end component is user interface. It enables to user to interact with the system. Front end is Web application accessible by browser. Basic capabilities are described per each screen in section 7.4.

Crawler store

Crawler store component is a storage for all data. Detail of the data model is in section 8.1. This component is server application with REST API interface (for more detail see section 12.2). This component has a single purpose – storing and retrieving data. Business logic of application is stored in the client (front end) and crawler runtime component.

Crawler runtime

Crawler runtime manages execution of crawlers. Crawler runtime has responsibility for running and stopping crawler. It uses crawler store to retrieve crawler definition and it has its own persistent message queue for handling crawler computation (more in section 9.3). When crawler retrieves new results it is stored in the crawler store.

Session manager

Crawler is using a browsers for crawling. Session manager is handling creation of session within this browser. This component also injects scripts to the running session to mirror page in browser (more in subsection 9.2.4). Mirroring is used by front end during crawler definition or when troubleshooting the crawler.

7.4 User interface

System will provide both input and output thought user interface. Input (*page model*) will be created by the end user, though Web application. Output of the system is data extracted from Internet and will be viewable through user interface.

Application will have several screens (see Figure 7.2). User interface should be intuitive as the end user does not need to have technical background and usually works with other Web application with advanced user experience.

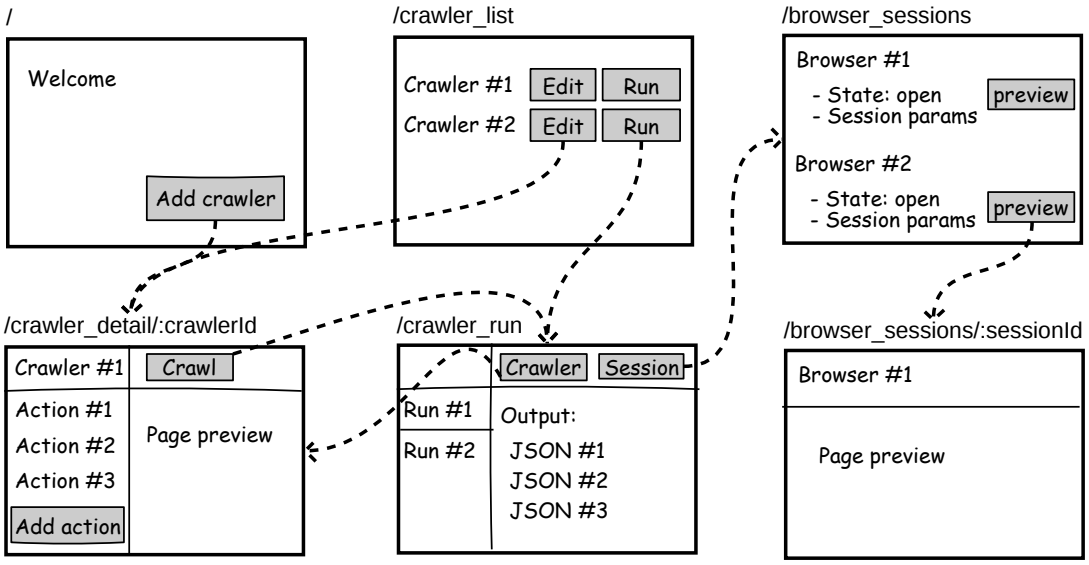


Figure 7.2: *Application screens.* Rectangles represent application screens, captions of rectangle represents URL path to screen. Dashed arrows mean from which screen we can go to other screens. Gray smaller rectangles represents buttons.

Crawler definition screen

Crawler definition screen is allowing live visualization on page we are crawling. This is important feature to graphically express all information on the page. Running crawler is also auditable by checking which page is currently open while crawling. On the crawler definition screen, there will be current sequence of commands showed, together with small wizard to add new command. If command needs to have element as an input (click on element, etc.), user can specify interacted element on the page visualization.

Crawler list screen

Crawler list screen will show a user defined crawler and let him/her run, delete or edit the crawler.

Crawler job screen

Crawler job screen will show user defined crawler and let him/her start and stop crawler job. The user will see how many results were returned by crawler and progress of the crawler.

Welcome screen

Welcome scree is used for a notification of user about new version and change release. We included button for adding new crawler as well, because it is the most used feature.

7.4.1 Out of scope

Solution is not aiming to curate or transform extracted data. Incremental crawling is also not in the scope (“What changed on page since last time?”), although it can be achieve by proposed solution. Deduplication of documents is solved on page hierarchy level only. This means that crawler is not checking if there are any duplicates in extracted data. This can lead to a situation when the item is moved during crawling from page 1 to page 2 (for example new item is added) and this item is crawled twice. Crawler will work with CSS selector for selecting data on the page. This will prevent to crawl pages with random generated CSS class names, however case

study introduced in section 4.3 suggests, that pages are not using random generated CSS class names often.

8. Data model

Crawler store will be storing data. This chapter contains description of data, which will be stored. Data model needs to contain definition of page model (crawler definition) and support for crawler runtime (data structures for storing internal state of runtime).

To give complete picture to the reader, we will describe runtime information, which are not stored in crawler store such as opened browser session and their internal state.

These entities needs to be used in order to satisfy requirements such as running or storing definition of crawler.

8.1 Stored data

Stored data in crawler store has three parts: crawler store, crawler runtime and open sessions.

All data created and defined by the end user are kept in Crawler store. There are kept crawler definitions and crawler definition states. Crawler definition consists of crawler definition model (more details in section 8.2). Crawler definition state (more in subsection 9.2.2) is used for storing state of variables during creation of model and storing id of the session used for definition of crawler. It is used for front end specific action, but needs to be persisted as front end can be closed/reopen any time. Diagram with full crawler store data model can be found in Figure 8.1.

The core entity of crawler runtime is crawler job. Crawler job represents crawler run with specific input and on specific crawler. Crawler job contains snapshot of crawler definition model, therefore model cannot be changed during crawling. Crawler job can be in “started” or “stopped” state. Crawler job can produce new crawler run state and process it according to given algorithm (more in section 9.3). Crawler run state has no defined schema and can be any shape (JSON). States are stored and queued by persistent message queue, therefore state remains if any of component disappears or is scaled to different number of instances. Crawler job progress is expressed as two numbers: count of processed state and count of states waiting to process. This caused by fact the fact that crawler does not know, how many results/pages it needs to process to complete the job.

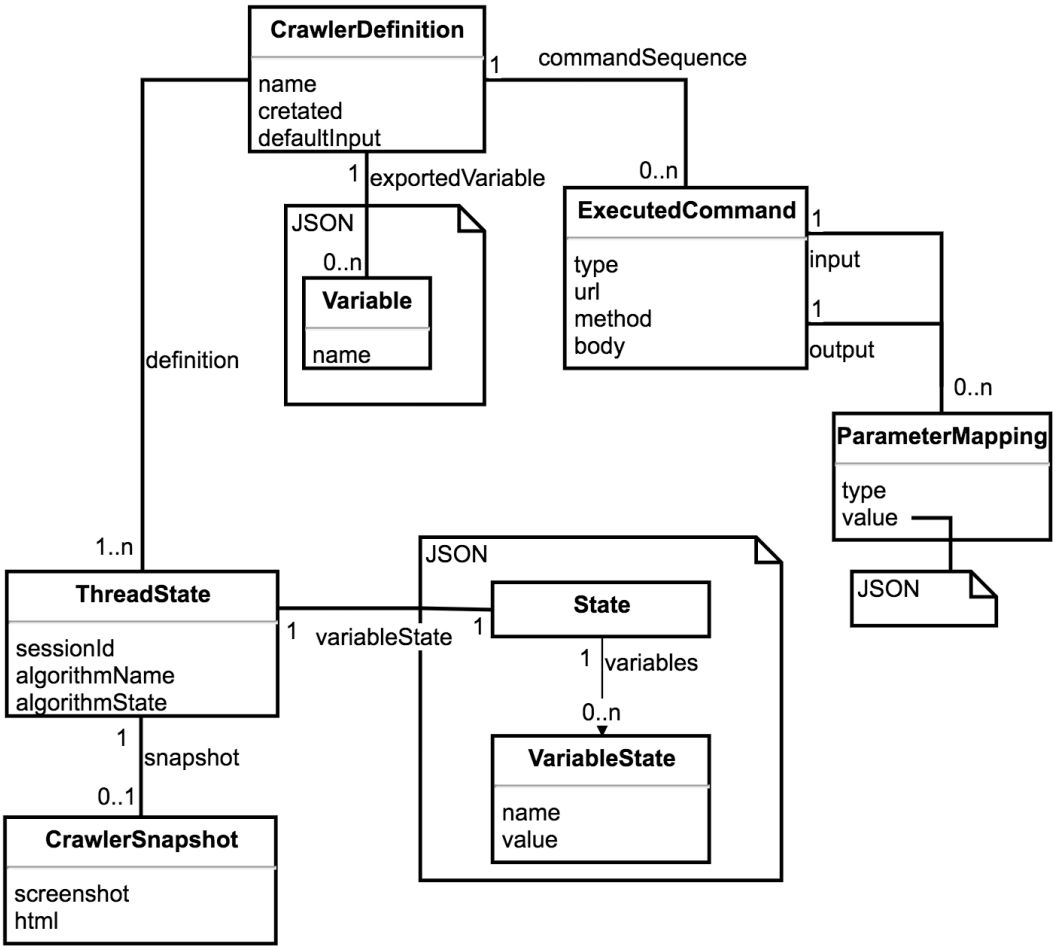


Figure 8.1: *Data model.* Rectangles represent entities and line represent associations between two entities. Rectangles with folder corner are comments.

Browser sessions have their own life cycle as they are handled by browsers themselves. If any other component creates new session, it is on responsibility of component to close it. If these components crash or does not behave properly, there can be memory leak with opened session. Sessions are volatile and they can be closed any time, therefore components need to check if session is opened before any operation. Session can be also reused. Session can reuse session state, if target browser provides such action. Browser sessions also open WebSocket channel per each session to send mirror session content (more in subsection 9.2.4).

8.2 Crawler definition model

Crawler definition model is used to crawl particular page. End user of the system creates this model of Web page using our solution. Model of the

page is defined as a sequence of commands. This sequence is interpreted by crawling algorithm to extract data from the Web page during crawling.

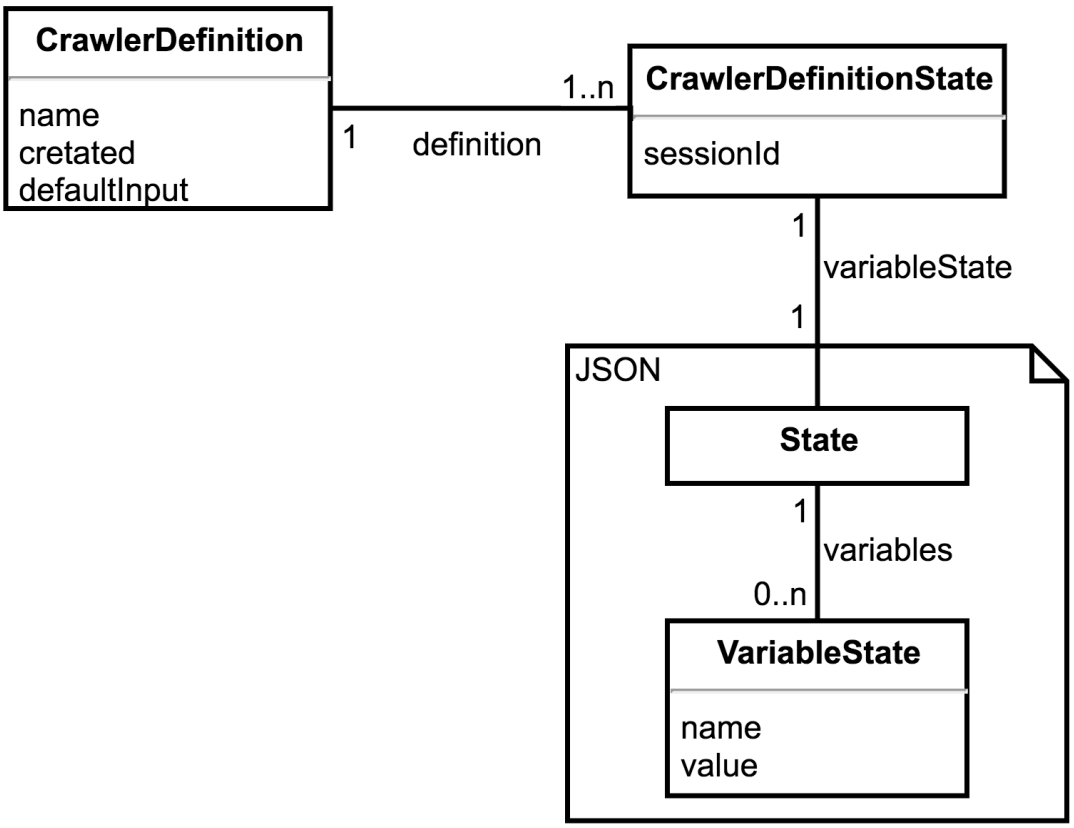


Figure 8.2: *Crawler definition data model.* Rectangles represent entities and line represent associations between two entities. Rectangles with folder corner are comments.

Commands (Table 8.1) are based on WebDriver specification[1]. Each of these commands have input and output parameters and they can use variables to pass data between commands[2]. For example, *find element* command defines the variable "name" in which it stores element id of element it selects. This element id can be used as input to other command. For example command *element click*, which can be evaluated using element id retrieved in previous step. Crawler is storing what variables to export when crawler yields result.

[1]There is no need to define new protocol, when there is one stable specification available. WebDriver is suitable for reuse as it contains low level command for Web manipulation.

[2]Note that variable are interpreted as string with one exception – array of string. Command can return array on as result (E.g. command *Find elements* returns array of element ids). This concept is used to process collections. Command which takes this array as input takes only one element of array as input, but whole process is forked to process all array elements simultaneously.

Special control flow command *yield* is also defined. *Yield* command is used to mark when crawler should emit result. Emitted result are exported variables with their current state.

Command name	**Input variables**	**Output variables**
Go to url	url	
Get url		url
Get title		title
Find element	locator	elementId
Find elements	locator	[elementId]
Find element from element	locator, elementId	elementId
Find elements from element	locator, elementId	[elementId]
Get element attribute	elementId, name	value
Get element text	elementId	text
Get element tag name	elementId	name
Get element rectangle	elementId	x, y, width, height
Element click	elementId	
Element send keys	elementId, keyCode	
Get page source		html
Executing script	jsCode	output
Take screenshot		screenshot
Take element screenshot	elementId	screenshot

Table 8.1: *Supported user command.* List of commands which can be used for definition of crawler. Input variables needs to be supplied to command in order to execute action and obtain results. Commands are subset of WebDriver specification [29]. Only those commands, which are relevant to end user during crawler definition are chosen.

9. Architecture

In this chapter architecture of the solution is described. Description of main algorithms used in the system follows. We advise to the reader to look at Figure 9.1 first to get holistic picture of the solution.

9.1 Front end application architecture

Before description of front end architecture, we will explain why we chose to make a client as Rich Internet Application in the first place. Page mirroring feature requires some kind of mechanism to render DOM received from WebDriver session. Therefore there is a need of having embedded Web browser engine in the crawler solution. There are four possibilities how to achieve that:

- *Custom native client application with embedded browser*, for example it could be Qt application with Webkit (QtWebkit).
- *Implementing Web browser extension.*
- *Using native UI framework interpreting DOM*, which is similar embedded browser (for example Electron [74]).
- *Creating custom Rich Internet Application*, which will be served via browser.

First three options have one thing in common – hard dependency on one version of browser. As modern Web pages and browsers change and evolve, crawler needs to have the most recent version of Web browser engine to be compatible with Web pages, that is crawling. For that reason we chose fourth option – Rich Internet Application with mirroring developed using iframe (more in subsection 9.2.4). This option has newest possible Web browser engine, because browsers are updated frequently.

For presentational layer React library [75] and for components library Material UI [76] is used. As framework Next.js [77] framework is used, because it provides application with initial setup of project (Webpack, etc.). It is also providing server side rendering and hot module replacement out of the box. We believe that these features are useful for developers of the page and will improve user experience during loading of the page. We chose this technology stack after research on modern Web technologies. There are many other options how to achieve our goal, this is only one of possible

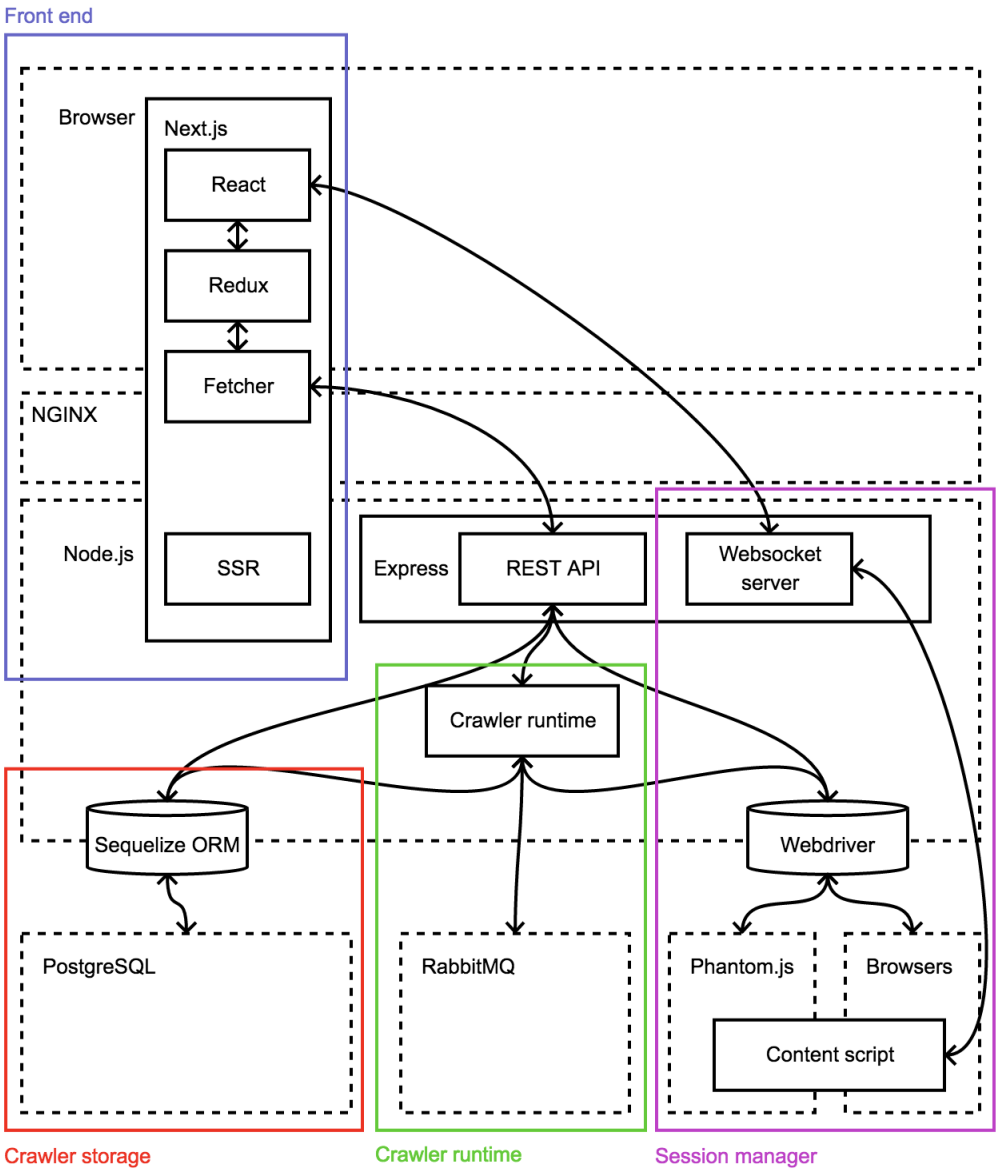


Figure 9.1: *Architecture.* Solid items represent parts of solution, which were developed for crawler. Dashed items are platforms, databases or message brokers, which were installed and configured only. Arrows connecting two items represent interaction between this two items. Note that React itself was not developed, but UI components using React were developed.

stacks. Front end component is written using Typescript [78] as main language as it will provide static typings, therefore more bugs will be found during compile time. We believe that static typing is important to scale large applications.

9.1.1 Presentational layer

React library is specialized on page rendering. It let compose UI from components implemented as JavaScript class and JavaScript function. This is used to modularize application. Components have internal state and properties set by their parents. When state or properties is changed, component and its children are re–rendered. One of main features of React is usage of virtual DOM[1], which construct tree structure of JS objects corresponding to original DOM. When there is a change, virtual DOM is changed at first, then difference from previous DOM are calculated and finally original DOM is updated only where DOM has changed. This will save processing time of unnecessary DOM updates, which are slow, because browser needs to change layouting, recalculate CSS style, re–render DOM, etc. This principle has synergy with Redux. Redux is used for managing presentational layer internal state (more in next subsection). When Redux state changes, whole React application will simply re–render and React will optimize DOM changes. React-redux [79] library provides a way how to set specific part of Redux state are rendered in which React component using `connect(mapStateToProps, mapDispatchToProps)` function. Screens are separated by application path using standard Next.js routing.

9.1.2 User interface application state

State management of front end is handled by Redux library [80]. As there are many HTML forms in our system we use library Redux Form [81] to handle these forms. For async operation front end is using redux-observable [82], which will also fit in crawler technology stack as back end is using RxJS reactive streams.

Redux library presents principle of singleton store (typically represented by single JavaScript/JSON object) containing whole state of UI application. This state cannot be changed directly, but action (single JS object) needs to be dispatched in order to change state. Dispatched action is processed by reducers, which change the state. Reducers are programmer–defined function with signature `(state, action) => state`. These functions take

[1] `https://reactjs.org/docs/faq-internals.html`

state as argument, alter it according to action, which was dispatched and returned new altered state.

9.2 Back end application architecture

Given the fact that front end application is based on JavaScript, we chose JavaScript for back end as well. Technology stack remains JavaScript, which helps to simplify development. "JavaScript stack" is proven technology for Web development, therefore is suitable for crawler development. As front end needs to be JavaScript application (only language browsers support), only one programming language for purpose of this thesis needs to be understand. This will save us time spent in learning phase, therefore more features can be implemented during development phase.

All solution's components on back end side are one Node.js [83] application, which are exposing REST API using Express library [84] (more in section 12.2) and using WebSocket server library called ws [85].

In back end there is large asynchronous code base. RxJS reactive streams [86] are used for handling asynchronous operations.

"Back end JavaScript" is hard to scale bigger project. In "front end JavaScript" module structure, naming conversion, etc. was determined using opinionated libraries like Next.js. We found is harded to scale and maintain pure JavaScript code, so we decided to use Typescript as main programmatic language. It will add compile time type safety and it is compiled to JavaScript.

Underlying storage of data in crawler storage component is PostgreSQL. Component use ORM library Sequelize to map runtime objects to database. Relational database was chosen, because data model is mainly relational in nature (as was mentioned in chapter 8). Exporting data from crawler is implemented using table to share data of crawler result. SQL database let listen on new data, therefore extracted data from crawler can be sent to other system in real time.

Running browsers is demanding on server memory. Crawler needs to use more instances of browser in order to let computation distribute over more than one machine. WebDriver is used to make abstraction for controlling the browsers. Crawler runtime use WebDriverIO [87] client to operate WebDriver protocol.

Page mirroring mechanism requires, that mirrored page open WebSocket connection to session manager (see subsection 9.2.4 for more details). Ses-

sion manager inject code which opens the connection using WebDriver method. Code is writen in TypeScript, however TypeScript is not natively executable in browser. This problem occurs on two places: injection our custom code and injection MuttationSummary library code. Custom code, which is written is serialized to string and inject JavaScript function. Therefore even if this function is written in TypeScript code it need to compliant with JavaScript (see file `/backend/src/session.ts` and function `pageScriptStub()` for detail). For library MuttationSummary, which is written in TypeScript, solution needs to transpile code into JavaScript during build process. Session manager inject JavaScript transpiled source code, not original TypeScript code.

Crawler job is using RabbitMQ to coordinate distributed runtime of (more was be elaborated in section 9.3 and reason of picking RabbitMQ as technology to implement coordination of runtime is described in subsection 9.3.2).

9.2.1 Application interface

Back end is exposing REST API, which is used by front end. REST API is common practise in world of Rich Internet Applications. Other systems can use this API to automatize crawler as well. This API is meant to be usable by other systems to integrate with our system. For example if other system wants to run a crawler to retrieve its result, it simply calls endpoint called run crawler.

Exported data are intended to be processed with other software systems. Crawler is supporting one time export of CSV or JSON file with results. JSON file is array of objects and every object in this array represents one result.

Other possibility to export data is using PostgresSQL table. Data are internally saved in a database table. Column called *data* type is JSONB and it is filled with resulting data. Exporting in such way also enables listening on newly received data.

9.2.2 Crawler definition process

Creation of crawler is complicated process, which needs three components to cooperate: crawler store, front end and session manager. In this section, whole process will be described in a detail. The process will be described from end user point of view to give a reader better insight.

Crawler definition state consist of session id of WebDriver session and current state of variables. WebDriver session is used to simulate end user's action and visualize state of Web page action user defined.

Process of crawler creation can be described as follows (pseucode can be found in algorithm 1). For crawler definition is used separate screen called *crawler definition screen*. When user clicks on button to create new crawler, he/she enters crawler name and default input of the crawler. Then crawler store create new empty crawler definition and crawler definition state with empty variables state. New browser session is opened and session id is stored in crawler definition store. Next crawler definition is initialized to default input by adding command *go to url* and executing it with default input. It this phase, end user can see initialized crawler and current opened Web page. The user can start editing the crawler.

Algorithm 1 *Crawler definition.* This pseudo code describes process of defining crawler definition from perspective of the system. Process itself is a loop which takes new command set by end user and update state of definition: variables and browser session.

```
1: procedure CrawlerDefinition(cra_def,brow_sess,var_state,new_cmd)
2:     cra_def ←add(cra_def,new_cmd)
3:     (brow_sess,cmd_result)←run(brow_sess,new_cmd)
4:     var_state ←update(var_state,cmd_result)
5:     return cra_def
6: end procedure
7: state ←emptyState()
8: while new_cmd ←userInput() do   ▷ Comment: Until user ends program
9:     state ←CrawlerDefinition(state,new_command)
10: end while
```

End user can add commands during crawler definition. After selecting new command he/she needs to set input and output parameters. Parameters can be saved/loaded from variables. Another option is that in case of input parameter it can be set as literal value. Some of the commands have special inputs. Special input can be predefined set of string constants (for example *Find elements* has five selector strategies: *css selector*, *link text*, *partial link text*, *tag name* and *xpath*). It can be also css rule which select specific elements. The system is providing way how to define these css rules visually. More information can be found in subsection 9.2.3. If command taking array of string as input is executed, system choose the first element of array.

When Crawler is defined, end user can still edit crawler using same screen. In some cases (garbage collection of non used session, system crashes, etc.) session associated with crawler definition could be deleted. In that case system will open new session and execute all commands in crawler definition to get session state synchronized with crawler definition state.

9.2.3 Visual definition of locators

Page case study in page case study in section 4.3 showed, that CSS selector are powerful enough to represent important elements in modern pages. These CSS selectors are defined visually to satisfy requirement FR2. This subsection described algorithm to interfere CSS selector visually. Note that this algorithm is heuristic and based on hypothesis we made based on page case study. This algorithm will be extended in future, however more advanced study needs to be done. We suggest to make study on page crawled by proposed tool by real users. For that crawler should store data about selected elements for future analysis. This subsection present current approach for visual definition of locators.

Task of visual selection of CSS selector can be expressed as function `(selected DOM element, DOM root element)` $\rightarrow$ `CSS selector`. User will be selecting DOM elements by mouse pointer (hover over the element), DOM root element represent whole Web page HTML tree as DOM root element contains pointers to its childrens. Algorithm will interfere CSS selector and show visually to the user, which elements are selected by this CSS selector (will draw border around elements). Note that by hovering mouse over one element, more than one element can be highlighted as CSS selector could select more elements. This behavior is for set of elements such as tables.

Any algorithm interfering CSS selector based on one selected element by mouse will be only heuristic, because DOM can change in any way and we based algorithm on hypothesis, that Web page structure remains same (or at least similar).

During testing of different heuristic algorithm, we found several approaches does not work:

- Structuring selector as full path of child combinator[2] (for example `div > span > a`) does not work, when there element has more than one child element of same type this selector choose all of them, not only highlighted one.
- Selecting by position of children of elements (for example `div:nth-child(3)`[3] `> span:nth-child(1) > a:nth-child(1)`), but it is not robust approach. When any node in DOM is added this selector is no longer valid. Unfortunately nodes are changing often. Note that this approach is very similar to using XPath (example above in XPath is `//div[3]/span[2]/a[1]`).

[2] Child combinator in form `a > b` selects all elements `b`, that are child of `a`.
[3] Nth-child pseudo–class `:nth-child(x)` select only `x`-th children of element.

- Selecting element by its id (for example `div#id123`[4]) is avoiding disadvantages of previous two approaches: it is more resilient to change in DOM tree and it can handle multiple elements of same type. Unfortunately this approach fails when element's ids are generated on the server. This was case for SBazar.cz Web page in case study.

Algorithm used for implementation in this thesis is based on hypothesis that Web designers use CSS classes to change style of important elements. Algorithm constructs CSS selector by selecting element that has defined class by selector `tag.class1.classN`. If selected element has no class algorithm traverse to its parent a make the check for class same as for selected element. This step is repeating until element with class is found. Traversed elements are used for second part of selector as simple select by tag name. Example of founded CSS selector is `div.class1.class2 span a`.

9.2.4 Page mirroring

Page mirroring takes important place in system, because it is used for creation and troubleshooting of the crawlers. Mirroring is hard to implement as it depends on several technologies. Therefore we chose to describe in this subsection in greater detail.

Main idea is to have an iframe in the front end which will make duplicated DOM tree of DOM tree in session. Because DOM in session usually varies in time, DOM is synchronized using WebSocket. Mirror is listening for DOM changes emitted by MutationObserver (see more in subsection 2.1.2). DOM changes are transferred from session to front end where they are applied to current DOM tree in iframe (Figure 9.2).

Because mirror interacts with session using only WebDriver, it has no direct access to DOM. It is using injected custom script which is interpreted in page context. Injection is done by executing script WebDriver method [88]. Script establishes WebSocket connections to session manager. One channel is used for sending commands to open new DOM streaming channel to the session. As there can be any number (even zero) of page mirroring clients we need to have same amount of DOM changes streaming channel as clients.

When session receives new WebDriver command which changes page execution context (for example *go to url* command), session will lose WebSockets connection and MutationObserver object is also deleted. In that case,

[4]Note that # operator does not support character _ in identificator. In case that identificator contains this character crawler needs to use attribute operator (for example `div[id=id_123]`).

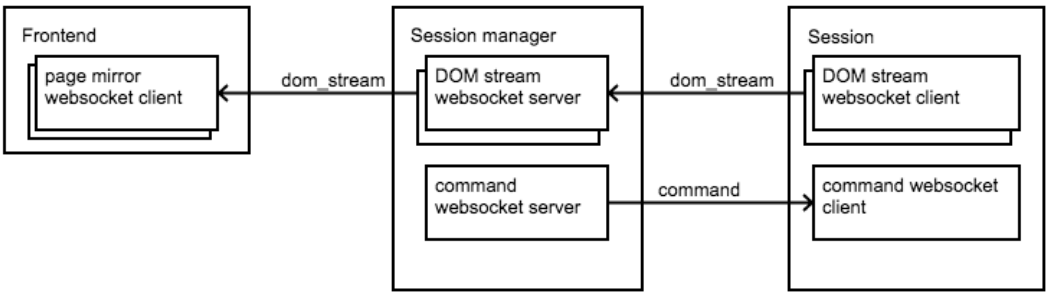


Figure 9.2: *Page mirroring components.* Page mirroring interacts with three entities: front end, session manager and session itself. Front end shows mirror streams DOM changes from session manager WebSocket. Individual WebSocket connection is opened for each front end mirror. Session itself is injected with JavaScript code, which starts streaming DOM changes to session manager. This injection is done ad–hoc, when mirror is created.

session manager detect lost of connection and reinjects scripts again. See Figure 9.3 for illustration.

When copy of DOM is mirrored to front end page served from different domain, there is problem with same origin policy. DOM element like fonts, scripts, images, etc. are loaded in a standard way. This means that fonts from other domains will not load as they are on another domain. Images with absolute source url will also load from original server, not from mirrored session. To prevent reevaluation of script on the page, mirroring is replacing `<script>` tag with `<noscript>` tag.

Note that mirroring is dependent on Web browser engine implementation it is using. Each Web browser core (Webkit, Gecko, Edge, etc.) can display pages differently. For example when WebDriver session is using PhantomJS, but front end is mirroring in Google Chrome. We are not reflecting this in our implementation, as crawled pages are usually optimized for major browsers.

9.3 Crawler runtime and algorithm

Interpretation of crawler is process (Figure 9.4) where each command is taken from sequence and evaluated it in open browser session. Current variable state is tracked during this process. In the beginning of the process, only *input* variable is assigned. This variable is filled with actual input of the crawler. In case input is an array, we will run crawler for each value

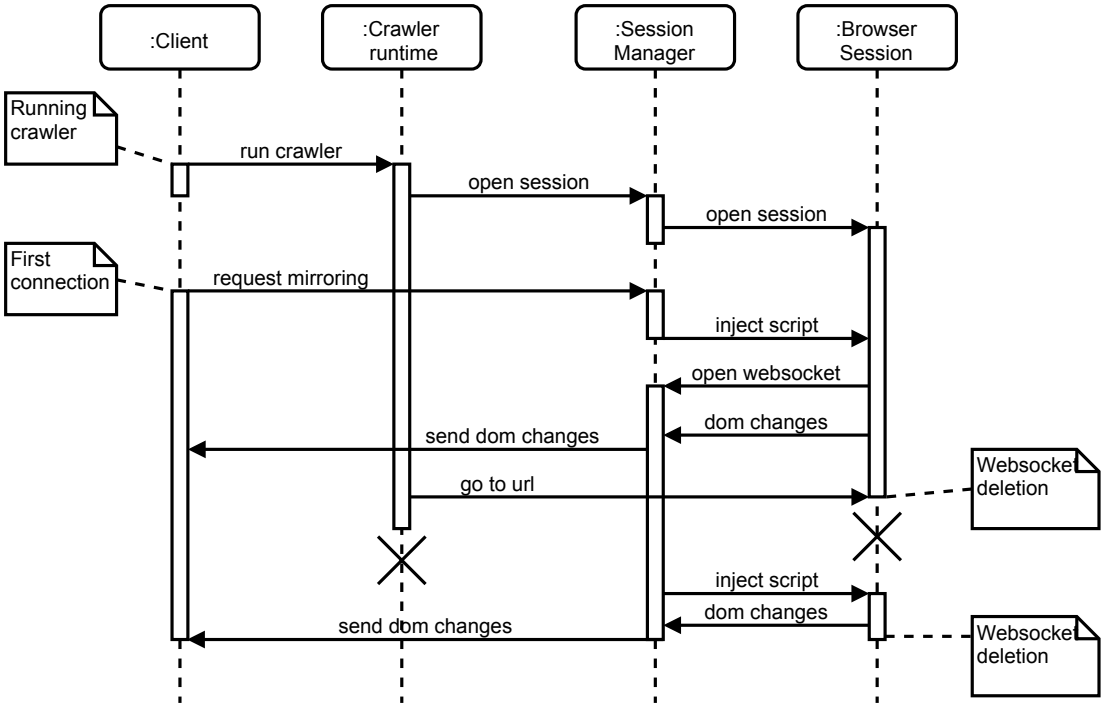


Figure 9.3: *Page mirorring.* This sequence diagram is describing common interaction between components, when session is initiated and mirror is requested. Note that when session moves to a different url, it drops whole DOM and therefore WebSocket connection with Session manager is lost. Session manager detect lost of connection and reinject the script.

of crawler as input. When command is evaluated, its input parameters are filled with values according to current variable state with one exception – when input variable has a value of array (this special case is described in following paragraph). After command is evaluated, the result is taken an stored to given variables. If command is “yield”, then crawler yields current values of exported variables. If there is no command left, crawler is considered ended.

When command has an input parameter taking array as value (for example when clicking on multiple elements), then only one element of the input array is evaluated, however whole process is forked to process rest of the elements (handling several processed is described in subsection 9.3.1). Forked process runs in separate WebDriver session. Forked process needs to synchronize its WebDriver session and variable state by running commands from the beginning. This could have performance impact on crawler runtime as crawler needs to create new WebDriver session and spend time on command re–execution. There are possible optimizations of this issue and they will be elaborated in chapter 16.

Changing algorithm of processing these parallel computations and way how algorithm “backtrack” has huge effect on crawler efficiency. Algorithm it-

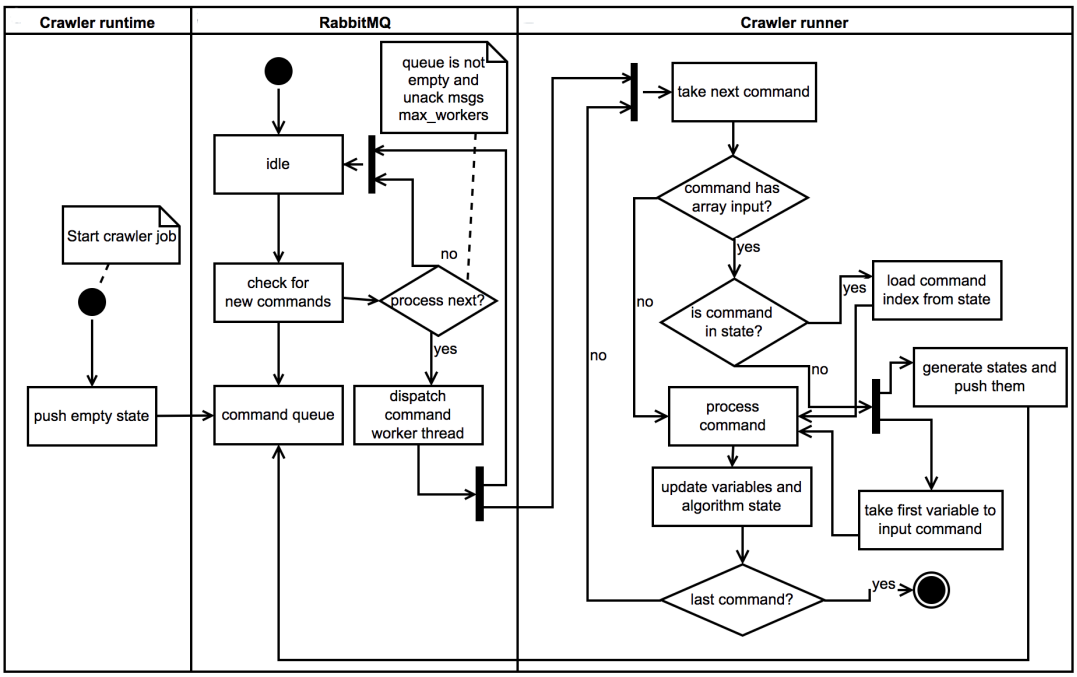


Figure 9.4: *Lifecycle of running job.* This diagram describes flow of running job. Diagram is divided to three parts crawler runtime, message queue and crawler runner. When activity or conditional check is drew in one of these part, it means that this activity or check is executed in corresponding component.

self is set of routines, which take crawler and the state as parameters and return new state. Implementation of this algorithm is not hard coded and schema of the algorithm state is not explicitly defined. Algorithm can be implemented and added to the system by changing these routines. Each algorithm instance interprets state in its own way. When its forking it emits new state to message queue and message queue delivers this state to new algorithm instance (see Figure 9.5). Sample implementation of this algorithm using depth first search is described in subsection 9.3.1.

Crawler limits parallelism (how many algorithm instances are processing command sequence) using set max count of algorithm instances. This will save system resources and protect crawled Web page from accidental Denial of Service attack.

9.3.1 Depth first search implementation

Depth first search algorithm is implemented to satisfy basic crawling scenario. This algorithm can be easily changed in future versions of crawler (see possible optimizations in chapter 16).

Process of executing depth search is following (see Figure 9.5). In the step one, previous state of the algorithm is taken from queue. In the step two, command with only one value is evaluated and cell is marked as “done”. During step three, command which results in array is evaluated. Index of first value in array is marked to current state. In the step four, rest of the command sequence for this state is evaluated in current session. Step five generates new states for rest of values from previous state. They have last cell filled as index of value in the array. In the step six, new states are enqueued. Process then returns to step one.

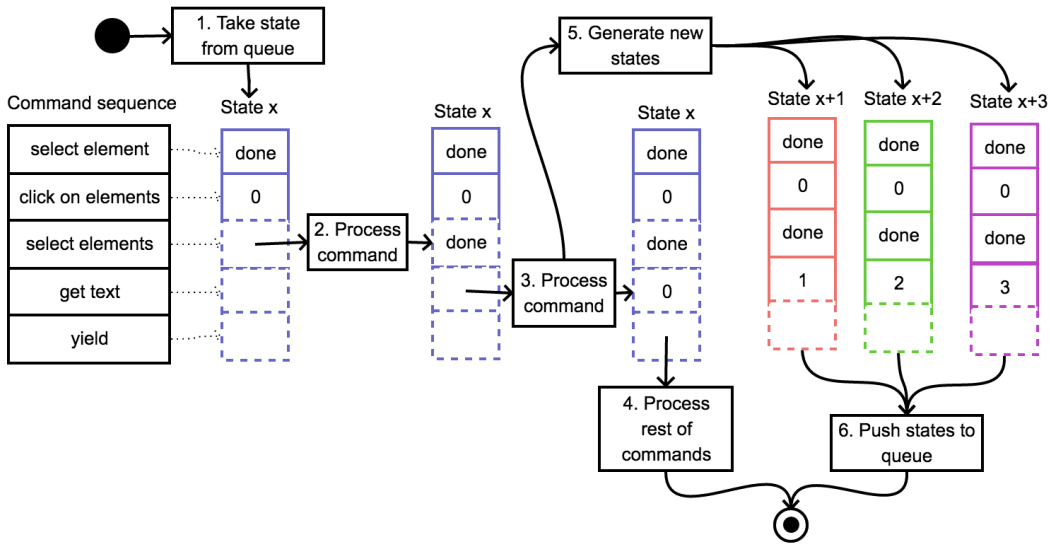


Figure 9.5: *State expansion.* This diagram shows the difference between processing command which results in one value and processing command which results in array of values. Solid colored cells represent state value which were already evaluated. Each cell is corresponding to command in sequence and it is containing information if command which was already evaluated. In case command has an input array of values, then cell is index of value which was taken.

When algorithm calculation encounters command which results in more than one output value, algorithm will finish only one (first) value within current session. Other values are planned to execute later. They are executed in new sessions, which need to be synchronized to the same state (rerun previous commands on new session) when process is forked.

To support forking of the process, crawler define state of algorithm as follows. Each state is array of values. This array corresponds to command sequence of running crawler. This array represents history of execution. When command is executed, it marks special value (“done”) in corresponding array. When command takes as input array, index of taken input value is marked. This state shape mimics stack memory for standard depth first

search algorithm with the difference, that it does not store all possible values, but only one. This happens because backtrack is not possible[5], therefore whole state will be executed from the beginning in other session (reset session and rerun all commands).

When new state is received through message queue to new worker, history is evaluated to get session to the same state as it was during branching. After that, crawler can execute rest of the commands and potentially branch when other commands have array as an input. Whole process of branching using state expansion is described in Figure 9.5.

9.3.2 Message queue technology selection

In prototype version of crawler we used PostreSQL table to synchronize distributed crawler runtime. PostreSQL table served as queue for crawler algorithm states. We found easy to query table to extract information such as count of remaining states in queue, however we found difficult to write queuing logic using SQL.

We found much easier and maintainable to implement message queue using RabbitMQ. The scaling of the solution is easier as well. One of the features message queue system need to offer is detection when queue is empty. In SQL case it was easily implementable as crawler could query DB and determine if there is any message inside. In RabbitMQ, this method is not in standard API. Rabbit Extension management API [89][6] can be used instead. One downfall of this approach is that these statistics are counted every five seconds [90], which effectively means, that crawler can detect that it ended with five seconds delay. We find this delay acceptable.

[5] We would needs have defined reverse action to be able to backtrack.

[6] Method called `/api/queues/:vhost/:name` can be used to solve this issue.

Part V

Implementation

10. Implementation overview

The entire solution is stored in one monolithic git repository. It contains two folders: one for front end called `/frontend` and one for all back end components (crawler store, crawler runtime, session manager, etc.) called `/backend`. These two folders are standalone Node.js projects. Front end is JavaScript Rich Internet Application and back end is written using Node.js. There are configuration files provided to run solution on Kubernetes platform.

Markdown readme file `/Readme.MD` is entry point of documentation for developer, who wants to maintain the project. Documentation for JavaScript code is written in JSDoc and it is generated in `/doc` folder. JSDoc can be generated using `/generate_docs.sh`. In the same folder REST API is described using Swagger file.

Developer experience is optimized for Linux–like system (and it is tested on MacOS). Solution itself is supported to run on Kubernetes on x86-64 architecture.

Both back end and front end are standalone Node.js Web server, therefore they can run on different IPs and listen to different ports. As browser has Same Origin Policy, which prohibits calling back end on different domain than front end is, crawler addresses this issue by using NGINX reverse proxy to serve application on one host only.

10.1 License

To let community use proposed crawler, we chose publish our work under open–source license. We chose MIT license in version published in website `https://choosealicense.com/` [91], because it is short and simple. License is used in following version:

```
MIT License

Copyright (c) 2018 Petr Fejfar

Permission is hereby granted, free of charge, to any person
obtaining a copy of this software and associated documentation
files (the "Software"), to deal in the Software without
restriction, including without limitation the rights to use,
copy, modify, merge, publish, distribute, sublicense, and/or
```

sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

11. Front end

Front end is React application, using Redux to manage application state and it is using Next.js as platform. Crawler's React components use UI components from Material UI. Redux–form library is used for handling any forms on the screen.

11.1 Front end folder structure

List of important folders and files within `/frontend` follows.

- `/package.json`, `/package-lock.json` and `/tsconfig.json`.

  These files are definition of JavaScript and TypeScript project. They contain basic project metadata and project dependencies.

- `/nginx.conf`

  NGINX configuration files for reverse proxy used for development environment (see more in chapter 13).

- `/Dockerfile`

  `Dockerfile` is used for building production container with front end application.

- `/pages`

  This folder contains Web application entry points: `/index.tsx`, `/browser_sessions.tsx`, `/crawler_jobs.tsx`, `/crawler_list.tsx` and `/crawler_definition.tsx`. These files are implementation of user interface screens defined in section 7.4 and they will be described in next section in more detail.

- `/components`

  React components used for UI. Note that this folder's subfolder are organized by feature. Each feature has UI file with React component and file with suffix `Store` which contains implementation of Redux Store and all needed Redux–observables.

- `/store`

  Top level implementation of Redux store. It imports all stores from `/components` folder.

- `/fetcher`

  Customly developed component for fetching REST API and serving result of these query to Redux store.

- `/static`

  Static files served by Next.js. For example custom icons are in this folder.

- `/styles`

  Initialization of Material UI styles.

11.2 React components

By Next.js conventions entry points of application are in folder `/pages`. These entry points are top level React components. List of all entry point is in Table 11.1.

Screen name	**URL path**	**React components**
Welcome screen	`/`	`index.tsx`
Crawler definition	`/crawler_detail`	`crawler_definition.tsx`
Crawler list	`/crawler_list`	`crawler_list.tsx`
Crawler job	`/crawler_run`	`crawler_jobs.tsx`
Browser sessions	`/browser_sessions`	`browser_sessions.tsx`

Table 11.1: *Mapping screens to source code and URL.* Screen name corresponds to screen defined in section 7.4. React components are stored in folder `/pages`.

Other non–root level components are stored in folder `/componets/`. These components are used by root level components and they are in separate files in order to let *developers* easily navigate in code written in React. These components[1] have corresponding file with name `<component_name>Store.ts`, which contains Redux store for the component.

Every root level component needs to be wrapped in `/components/AppWrapper.tsx` in order to top application bar with application menu. Every root component needs to be wrapped with function `withRoot` and `withRedux` in order to properly initialize Redux store, Next.js and Material UI.

[1] With exception of `withRoot` component

In component's folder `/components/`, components are divided by features. This helps hold semantically similar code close together.

In file `/components/formComponents.tsx`, there are stored render functions for Redux form. Render functions enable to Redux form use custom UI components. In this case, we are using Material UI components.

11.3 Application state

Each React application needs to define strategy for managing its internal state. In proposed solution state is managed by Redux. In files `/store/{Action,Reducer,State,Store}.ts` there is defined Redux store as well with redux-observable epics in variable `rootEpit`. For making easier work with asynchronous Redux actions implemented by redux-observale, an utilization function `waitUntil` is prepared. This function is returning promise, which will be resolved when particular action is dispatched. This is especially useful in `getInitialProps`, when component logic wants fetch data on server and render component on server when data arrives.

Reducers are defined for every component in its corresponding `<component_name>Store.ts` file and `/store/Reducer.ts` only combined them together. File `/store/State.ts` defines TypeScript type for Redux state, however implementation is stored in each `<component_name>Store.ts` file.

Crawler redux state shape is illustrated in Listing 11.1.

Listing 11.1: *Redux state shape.* Note that this is not valid JSON, this is for illustration of Redux store state schema.

```
{
  "fetcher": {
    /* state  fetcher for several endpoints */
    [endpoint]: {
      isFetching: boolean,
      data: any | null,
      error: string | null
    }
  },
  "uiState": {
    "addCrawlerDefinitionState":
      { /* state of UI of dialog addCrawlerDefinition */ },
    "crawlerDefinition": { /* UI state of screen
       crawlerDefinition */ }
  },
```

```
  "crawlerDefinition": { /* state of crawler definition */ }
  "crawlerDefinitionState": { /* state of crawler definition
     state */ },
  "form": { /* Redux-form internal state */ }
}
```

11.4 Fetcher component

Data from back end are fetched by customly developed module, which fetches data automatically. It is using redux-observable streams and it stores data in the Redux store. Endpoints needs to be declared explicitly. Source code is located in `/frontend/fetcher` folder.

Fetcher component needs to have explicitly provided API method for fetching the result in file `/frontend/fetcher/api/API.ts` also define name of redux action associated with this endpoint in file `/frontend/fetcher/Fetcher.ts`. After redux action which match endpoint name is dispatched fetcher will automatically start fetching data from endpoint and dispatch action with suffix `_FETCHED` when fetched is completed. In case that an error occurs, action with suffix `_ERROR` is dispatched. Fetched data, error message when error occured and fetching state is stored in redux store in property `fetcher`.

12. Back end

Back end is server Node.js application. Entry point of this application is `main.js`, which is Node.js script. Unlike front end's entry points, which are Web pages.

12.1 Back end folder structure

- `/package.json`, `/package-lock.json` and `/tsconfig.json`.

  These files are definition of JavaScript and TypeScript project. They contain basic project metadata and project dependencies.

- `/src`

  Folder with all source code for backend. It contains Express.js configuration and Session manager WebSocket server.

- `/src/routers`

  Definition of custom routes for Express.js REST API.

- `/src/model`

  Definition for Sequelize ORM model.

- `/src/runner`

  Implementation of Crawler runner component.

- `/src/server_files`

  Folder with files, which are served statically. These files are used for page mirroring. It contains copy of `muttation-summary` library [92], which is served as static files, which are injected to session (more was elaborated in subsection 9.2.4).

12.2 Back end API

The crawler has two exposed API: REST API and WebSocket for page mirroring. REST API is used by front end and it is potential extension point of the solution. WebSocket API used to internal communication for page mirroring feature.

REST API is exposed by Express.js server and consists of three major parts: session management, crawler definition management and crawler job management. API methods are described in Swagger documentation enclosed to the thesis in folder `/doc/`. Express server is defined in file `/src/server.ts` and all routes are located in files in folder `/src/routers/`.

WebSocket server belongs to component session manager. It receives connections from front end and from sessions themselves. Messages are in JSON format and they contain property `type`, which can be one of these values: `setBase`, `initialize`, `applyChanged`, `heartbeat` and `start`. Detail of protocol can be found in Table 12.1. Connections coming from session to register itself are accessing HTTP resource with URL end with `/register/:sessionId`. All other connections for exchanging DOM changes are accessing `/register/:mirrorId`.

Message type	**Sender/receiver**	**Description**
Start new mirror	session manager → session	This message instantiate new mirror and returns new mirrorId.
DOM changes	session → session manager	DOM changes emitted by browser session.
DOM changes	session manager → front end	DOM changes received from session, which needs to be applied in browser.
Heartbeat	front end → session manager	This message keeps WebSocket alive.
Heartbeat	session → session manager	This message keeps WebSocket alive.

Table 12.1: *WebSocket protocol for page mirroring.*

12.3 Crawler store

Crawler store component stores crawler related entities (entities were defined in chapter 8) in database. Crawler store is using PostreSQL as underlying storage. Object–relational mapper (Sequelize library) is used to map underlying storage data to JavaScript objects, which can be used in Node.js code. Whole model of Sequelize ORM is in file `/src/model/store.ts`. When

this model is synchronized with PostgreSQL, corresponding table are created, therefore *developer* does not need to manage PostgreSQL in any way expect installing. ORM model created in `/src/model/store.ts` is used for data manipulation from Node.js code.

12.4 Crawler runtime

Whole implementation of DFS algorithm correspoding to subsection 9.3.1 is in file `/src/runner/job_runner.ts`. The most import method is `runQueue` this method is called by crawler, when user starts crawler job. Purpose of this method is to start listening on RabbitMQ channel with name `task_queue_<crawlerJobId>` and start executing command in order to crawl new data from Web page (process in detail was described in section 9.3).

12.5 Session manager

Session manager is managing browser sessions. Whole implementation is in file `/src/session.ts`. Another functionality of session manager is to provide page mirroring functionality described in subsection 9.2.4. For page mirroring there are important three parts of code: file with MuttationSummary library in folder `/src/server_files/`, injected `function pageScriptStub()` and Session Node.js wrapper `class Session`.

Folder `/src/server_files/` contains all files, which are injected to session in order to support MuttationSummary in browser automated session. In `function pageScriptStub()` there is code which handle connecting opened session to session manager. This code is injected to the session every time session changes URL (see subsection 9.2.4 for more details). On Node.js server there is created instance of `Session` for every automated browser session. This object handles injection of code to session and opening/closing mirroring WebSocket as well.

13. Deployment

Solution is supporting two types of deployment: production and development. Development environment is used by *developer* and it is optimized for speeding up the development process. Production is environment is managed by *system administrator* and it is used for running application in stable manner.

13.1 Production environment

Production components are Docker containers deployed on Kubernetes. There are five different Docker images that solution is using. Front end, back end, PhantomJS, RabbitMQ and PostgreSQL (see Figure 13.1). Deployment manifest is file `/deployment/kubernetes.yaml`.

Back end image consist of Crawler runner, Crawler storage and Session manager, which are implemented as one Node.js project stored in `/backend`. Its `Dockefile` is placed in `/backend`, it is builded by `npm run` and run by `npm start`.

Front end image is running `npm build; npm start` on start. Its `Dockefile` is placed in `/frontend` and it is inspired by this blog article [93].

XNGINX image contains configuration of proxy. It is defined in deployment manifest.

Docker image with PostgresSQL is initialized by ORM (Sequelize), therefore clean image is sufficient. It is defined in deployment manifest.

Kubernetes will manage runtime of components - it will restart them in case of crash and also it will let scale components (Phantom.js, RabbitMQ) increasing number of Kubbernetes replicas.

Process of deployment is consist of these steps:

1. Install Kubernetes.

   See `https://kubernetes.io/docs/setup/` for details.

2. *Build Docker images for front end and back end.*

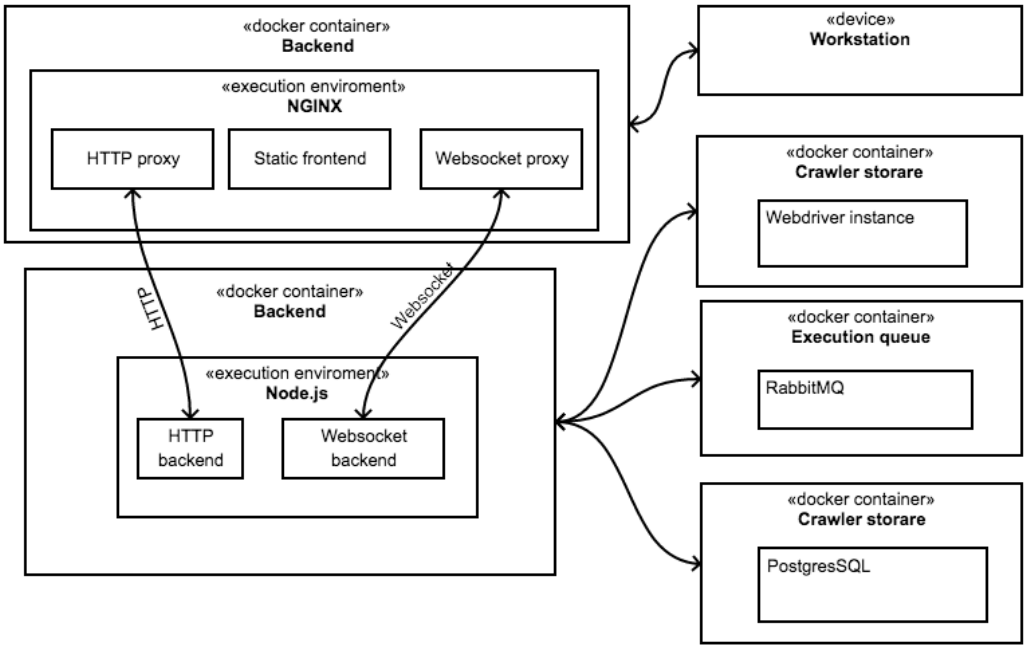


Figure 13.1: *Production deployment model.*

For building image there are prepared scripts in `/deployment/build_docker_images.sh`, which uses `/frontend/Dockerfile` and `/backend/Dockerfile`.

3. *Deploy crawler to Kubernetes.*

   Run `kubectl create -f <work_dir>/deployment/deployment.yaml` to create deployment.

4. *Enable managment API for RabbitMQ.*

   Connect to RabbitMQ CLI and run `rabbitmq-plugins enable rabbitmq_management`.

5. *Test deployment.*

   Visit `http://<kubernetes_ip>:8080/`. If deployment was successful, welcome page of crawler should be shown.

13.2 Development environment

Development environment let *develepor* use comfort of hot module replacement for React and continuous compilation of TypeScript sources. Development environment needs installed PostgresSQL and RabbintMQ[1].

In order to setup developement environment, run set of scripts script for *developer* needs to run four command listed below. We recommend run these commands in terminal multiplex (for example tmux [94]) in multiple terminal pane. Output of these commands is important to *developer* as it contains logs and compile messages. Example setup using iTerm2 [95] is shown on Figure 13.2.

- `/watch_backend.sh`

  This command starts to compile source TypeScript files `*.ts` into JavaScript code `*.js`. It will watch source files for change and it will recompile it, therefore if developer make change, he/she does not need to trigger compilation.

- `/watch_frontend.sh`

  This command starts to compile source TypeScript files `*.ts` and `*.tsx` into JavaScript code `*.js`. It will watch source files for change and it will recompile it, therefore if developer make change, he/she does not need to trigger compilation.

- `/start_frontend_dev.sh`

  For development *developer* can use Next.js dev script called by command `npm run dev` for starting front end. Afterwards he/she will not use generated files from build but Next.js will start dev server which provide him/her with hot module replacement [96]. Node that back end and front end is listening on different ports. To overcome problems with Same Origin Policy this script runs NGINX proxy defined in `/frontend/nginx.conf`.

- `/start_backend_dev.sh`

  This run back end in developer mode. It mean, that it can be debugged by for example Visual Studio Code debugger, because in listening for remote debugger on default port.

[1] We recommend use Docker image `library/rabbitmq` and `library/postgres` for development environment. Using Docker will save time, because *developers* do not need to install PostgreSQL and RabbitMQ from scratch.

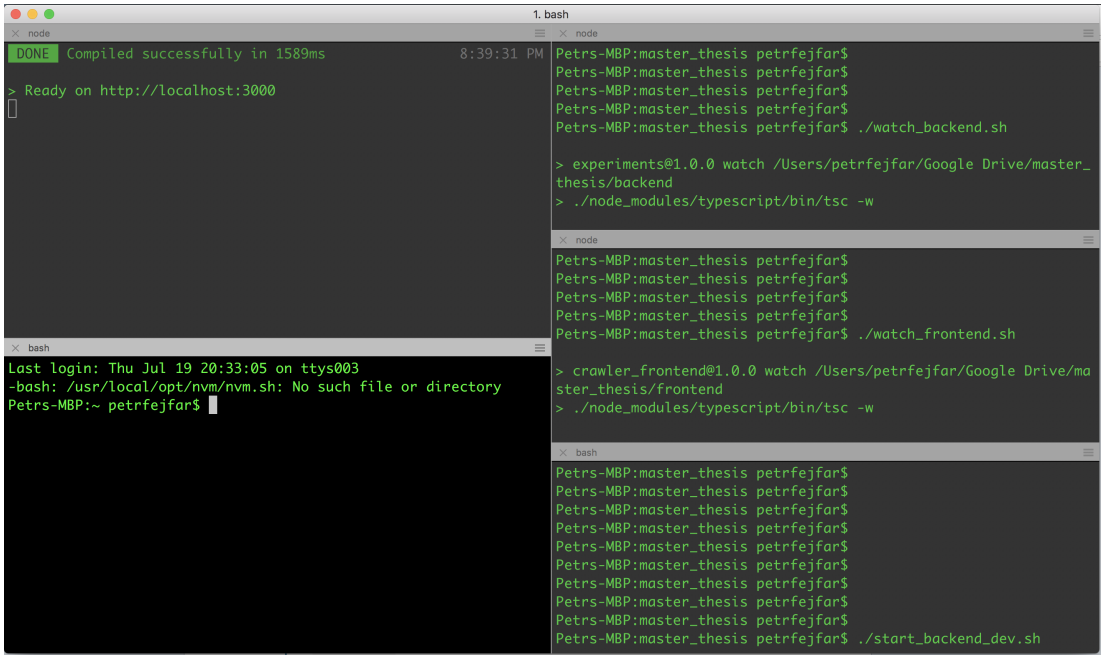


Figure 13.2: *iTerm2 based working environment.*

Part VI

Conclusion

14. Comparison with other crawlers

This chapter compares proposed solution with other available crawlers. Goal of this chapter is to show difference of proposed solution from others and put proposed crawler in context. Each crawler is discussed in context of requirements set in section 1.3.

14.1 Apache Nutch

Apache Nutch[1] is mature crawler for static Web pages. It is based on Apache Hadoop data structures for batch processing (crawling process is implemented using MapReduce jobs). Nutch is designed to be highly scalable in distributed environment and it is created for massive parallel processing. Apache Nutch is typically good for crawling of large portions of publicly indexable Web.

Apache Nutch is internally representing found pages by URL, which disqualifies Nutch for fully support crawling of RIAs. There is extension for pre–rendering HTML page by PhantomJS, however no action on page is invoked afterwards. Apache Nutch is following all links in HTML, which is good strategy for obtaining large amount of pages, however users cannot define other custom strategy (definitely not visually).

14.2 Scrapy

Scrapy [97] is popular Python framework for crawler creation based on principle of HTTP programming (therefore not supporting RIA). It can be integrated to other systems extensively and it is highly customizable, however developer effort is required. Elementary techniques used for data extraction are simple CSS/XPath selectors [98]. Developer cannot define these selectors visually. Developer can also find convenient methods prepared for link extraction for developer's convenience.

An internal queue for processing tasks is provided, however it is using memory or file–system to store pending task and therefore it cannot scale to more then one server. Scrapy is open source software and there is cloud based service providing Scrapy with infrastructure called ScrapingHub.

14.2.1 Splash

Splash [99] is a service used for helping Scrapy dealing with RIAs. Splash acts as proxy for Scrapy. It loads Web page in headless browser, waits until JS is executed, other resources are loaded and then returns a page in format of HTML to Scrapy. Therefore only initial render of the page is performed, but crawler can interact with the page in any way. Splash is not a crawler, it is rather powerful extension for Scrapy.

14.2.2 Portia

Portia [100] is Web application used for visual definition of Web data extractor. It is using live preview of the Web page to let user visually define which data to extract from one specific page. When Portia crawls website, it is starting at specified seed page and it is following all links to obtain all pages of website. For every page retrieved this way, Portia tries to extract data by visually defined extractor.

Portia is built on top of Scrapy, therefore it is using queue of URL to crawl all pages of website. It can handle initial load of JavaScript via Splash, but not whole RIA.

14.3 Import.io

Import.io is online tool for definition of Web data extractor. It can handle Web page initialization (by loading in headless browser), however it cannot manage RIAs. Web extractor is created visually, also Import.io presents special Magic extractor. This extractor tries to extract data from the page without user interaction. The magic extractor is not working all times, however it can handle majority of websites. Implementation of this extractor is following: *"The Magic algorithm looks for the biggest list (with the most data) on the page. It then uses that list to auto-generate the rows and columns and bring them into a table. In some cases it can even get the subsequent pages."* [101]

Import.io use similar strategy for crawling website as Portia. It access all page by following all links on page and then executes extractor on these pages. Other option how to obtain all pages with data, is to generate its URL by regular expression provided by user.

Import.io cannot crawl RIAs and it is on cloud solution with closed source code.

14.4 UiPath

UiPath is a program for solving task called Robotic Process Automation (RPA). RPA is trying to automate GUI systems which does not have API. These systems does not need to be browser, but even native applications. RPA is simulation real user interacting with system. To select the element for interaction, RPA uses several techniques from simple "click on x,y-cooridnate on screen" up to XPath when automating browsers.

As RPA automate browsers they can be used for crawling and data extraction task as well. This also concludes that RPA handles RIAs. Downside of RPA is that they are not designed for crawling and data extraction natively, therefore they provide only basic techniques for selecting elements. Other downside of UiPath is that it is using virtualized OS with full browser (not head–less). This will negatively impact demands on CPU a RAM, especially when crawling in parallel.

UiPath (in general even other RPA) are closest to our crawler. UiPath can crawl RIA and provide user browser–like experience when defining crawler. Despite this fact UiPath is not perfect solution of our problem. One reason is that due its complexity is very demanding on computational resources and also time to learn how to work with UiPath is much longer than with our crawler. Other reason is bad extensibility in context of crawling algorithm a visual selection algorithm. We planned to extend these functionalities in future and UiPath is not suitable for extension this way.

14.5 Diffbot

Diffbot [102] is crawler and Web data extractor using visual information to extract data. Differently from previous presented solutions, Diffbot uses machine learning techniques to find element on page to extract. It is using WebKit underhood [103] to render page, therefore initial load of page is handled even if JavaScript is present. Diffbot is using human annotated dataset to train machine learning model to extract finite set of pages types: *"We've identified roughly 20 types of pages that all the Web can fall into. Article pages, people pages, product pages, photos, videos, and so on. So one of the fields we return will be what is the type of this thing. Then, depending on the type, there are other fields."* [104]

Limitation of Diffbot is that it cannot handle page of different type than indentified 20 types. However big advantage of this approach is that Diffbot does not need any user configuration for crawling pages it is supporting.

14.6 Comparative analysis summary

None of the solutions in this comparison (see Table 14.1 for feature matrix of these solutions) can optimally solve problems that we decided to deal with in this thesis (requirements are in section 1.3). There are other solution, which lets the user define crawler and extractor visually however none of these fully support crawling RIAs. The closest solution to our proposed one is UiPath. UiPath is a program focusing on different task and it is heavy–weight for crawling RIA. We believe that our solution brings better potential for crawling as many optimization can be done (see chapter 16). Immediate added value to our solution in comparison to UiPath is usage of headless browser, which brings better performance than provisioning whole OS with browser.

	Apache Nutch	**Scrapy**	**Splash**	**Portia**	**Import.io**	**UiPath**	**DiffBot**	**Our solution**
Custom crawler logic		✓	n/a			✓	n/a	✓
Custom extractor logic		✓	n/a	✓	✓	✓		✓
Handle RIA	?(1)		?(2)		?(2)	✓		✓
Visual definition of extractor				✓	✓	✓		✓
High performance	✓	✓	✓	✓	n/a		n/a	✓
Crawler extendability	✓	✓						✓
On premise	✓	✓	✓	✓		✓		✓
Open source	✓	✓	✓	✓				?(3)

Table 14.1: *Comparision of crawlers.* (1) Apache Nutch has installable extension for pre–rendering page HTML by PhantomJS. (2) Note that Splash and Import.io handles only initial load of RIA as well, therefore they do not fully support RIA. (3) Source code will be released on page of the university after thesis release.

15. Conclusion

The purpose of this thesis was to develop a tool that allows user to crawl Web pages with no programming experience; at the same time, the tool renders the static Web pages and RIAs indistinguishable. As a result, the tool provides the user with browsing–like experience when setting up crawler, and further, the tool supports crawling RIAs.

At the beginning of the thesis, we outline requirements for this new crawler to ensure that it has features of a universal tool for crawling RIAs. The following part was dedicated to relevant Web technologies and to the analysis of modern Web pages. We found Web pages which are problematic to crawl using regular crawlers, but which the proposed crawler will be able to crawl easily. We also described static pages crawlable with ordinary crawler to understand portfolio of pages which needs to be crawlable. In the next part, we presented theoretical approaches for implementation of universal crawler and we found out that although the problem has been described in theory, practical implementations for crawling static pages and RIAs were missing.

Combining our analysis of modern Web pages with analysis of theoretical crawling approaches, we have put forward a hybrid approach using Web wrappers defined by CSS selector and model–based crawling. Based on these chosen techniques, we have designed a crawler tool which would let user visually extract data from a page. First, the user specifies the model of the page, which is a sequence of commands executable in the context of Web page (mouse clicks, keyboard hits, etc.). Second, this model is used as an input for the crawler and data extractor, which executes these commands and retrieves data from the page.

The proposed tool was implemented as RIA as well, therefore it provides user graphical interface and it is convenient for displaying crawled page to the user. Troubleshooting crawlers is a challenging process and displaying current state of crawled Web page to the user helps identify the issue with the crawler faster. Architecture and design of our tools is described in this thesis.

During design and implementation phases, we found that our solution can be extended/optimized in ways that we have elaborated on in the next chapter; however, we believe that implementation of this optimization is beyond the scope of this thesis, mainly because of time constraints. Possible future extensions and optimizations are elaborated on in the next chapter to let the reader understand how to extend the solution in the future.

16. Future work

The proposed tool fulfilled all requirements declared in section 1.3, which was stated as goal of the thesis. Therefore, the tool is able to help non–technical users to crawl and extract data from RIAs and these users does not need to understand difference between static pages and RIAs to successfully crawl them.

However, we believe that our tool could help users with crawling Web pages more. During testing of our tool on Web pages, we found three areas for future improvements: optimization of crawler performance, automation of *page model* creation and *page model* adaptation.

Performance optimization

Crawler runtime can take a lot time when crawled website is big. There are many factors influences performance. During testing our tool on Web page, we identified two main reasons: crawler resets and time need to execute one command step.

Crawler reset happens, when crawler needs to fork during processing command with array of values as input (was described in section 9.3). New forked processes need to re-execute all previous commands from beginning – this is call the reset. Re-executing commands during the reset is slow.

Possible optimization of time spent on resets:

- When crawler changes URL during processing of command, crawler can make a reset to page with this URL. However this optimization has assumption that this URL identifies this page uniquely. Therefore algorithm for determining if URL represents page uniquely needs to be developed.
- Reseting session and the start from beginning during reset could be sometimes slower, than backtracking. Backtracking would be done by hitting history button. This possible optimization has assumption that backtracking return crawler in state when it was before executing command. Therefore algorithm for determination of state uniques needs to be developed.
- Previous two optimizations have an assumption that crawler needs to return to same RIA state. In theory this state can be copied as DOM can be serialized and executed JavaScript state can be snapshotted.

Making a copy of DOM and making snapshot of v8 is costly operation[1], however it can be faster than full reset (especially with slow network request).

- Another methods is developed method by Choi et al. [105]. Authors are creating model of page using learning algorithm trying to minimize number of restarts this model determines states of RIA and its equivalence and optimize crawler specifically for this model.

Second reason for slow crawler performance is long time needed to execute one command. First possible optimization is to not interpret page via browser. Some page could be interpret via HTML programming, which will save resources of the server. This optimization has assumption that page is not RIA (or at least some of command are not JavaScript actions). Crawler can determine this by executing command sequence using HTTP programming and browser automation in parallel and compare its results. If they are the same, HTTP programming can be used.

Current implementation is waiting defined constant time to let command execute. Waiting between command takes a lot of extra time. Problem is crawler itself cannot determine if command has ended (it is NP–hard problem). However user can define condition how to recognize that command has ended.

Automation of page model creation

Creation of *page model* is time consuming task, specially when more than one page needs to be crawled. Our tool provides a way to select extracted elements visually, which help to speed up the process of definition, but we believe, that our tool could help more user with *page model* definition in the future.

Selecting interesting element to crawl by mouse pointer is a easy task for a person as he/she can see elements he/she wants to select. Current progress in machine learning, more specifically with convolutional neural network, helps to solve data processing tasks. This model could be trained for extraction of data from Web page or at least advising which elements on page is worth to extract/which element is intend to interact with. Similar approach was used by Gogar et al. [67], in future work we can expand this by using their solution.

Machine learning can be used more in future of the crawling. Reinforcement learning could be used in the entire crawling process (transfers on

[1] And difficult to implement.

the page and data extraction). Input for this machine learning task would be description of data to extract and output extracted data. Initiative to automate Web robot in similar way has already started, for example Karpathy et al. [106] created dataset for training reinforcement learning model for specific Web task. However, no solution is ready yet.

Adaptation of page model

During creating section 4.3 with page case study, we encountered a problem, when Web design of Sbazar.cz website changed. During website layout redesign the CSS classes changed and *page model* was no longer corresponding to Sbazar.cz page, therefore crawler was not able to extract data from the page using this *page model*. Solution of this issue is to create new *page model*, which requires user interaction. This issue costs users some time as they need to check regularly if crawler is extracting correct data. Changing model automatically to reflect site change is called Web wrapper adaptation and it can be used to make our tool maintenance–free.

Bibliography

[1] Apache Nutch Highly extensible, highly scalable Web crawler. http://nutch.apache.org/, 2018. [Online; accessed 01-February-2018].

[2] Emilio Ferrara, Pasquale De Meo, Giacomo Fiumara, and Robert Baumgartner. Web data extraction, applications and techniques: A survey. *Knowledge-based systems*, 70:301–323, 2014.

[3] Ali Mesbah, Arie Van Deursen, and Stefan Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):3, 2012.

[4] The WARC File Format (Version 0.16). http://archive-access.sourceforge.net/warc/warc_file_format-0.16.html#anchor1, 2018. [Online; accessed 05-February-2018].

[5] The Official CAPTCHA Site. http://www.captcha.net/, 2018. [Online; accessed 05-February-2018].

[6] How to break a CAPTCHA system in 15 minutes with Machine Learning. https://medium.com/@ageitgey/how-to-break-a-captcha-system-in-15-minutes-with-machine-learning-dbebb035a710, 2018. [Online; accessed 05-February-2018].

[7] CAPTCHA Bypass done right. http://www.deathbycaptcha.com/, 2018. [Online; accessed 05-February-2018].

[8] reCAPTCHA Protect your site from spam and abuse. https://developers.google.com/recaptcha/, 2018. [Online; accessed 05-February-2018].

[9] Wired Captcha is dying. This is how it's being reinvented for the AI age. http://www.wired.co.uk/article/captcha-automation-broken-history-fix, 2018. [Online; accessed 05-February-2018].

[10] The New York Times Facebook Says Cambridge Analytica Harvested Data of Up to 87 Million Users. https://www.nytimes.com/2018/04/04/technology/mark-zuckerberg-testify-congress.html.

[11] Jakub Kudela. First nerdy steps in buying an apartment. https://www.linkedin.com/pulse/first-nerdy-steps-buying-apartment-jakub-k%C3%BAdela/, 2017. [Online; accessed 05-February-2018].

[12] Architecture of the World Wide Web, Volume One W3C Recommendation 15 December 2004. https://www.w3.org/TR/webarch/, 2018. [Online; accessed 06-February-2018].

[13] Media Type Specifications and Registration Procedures. https://tools.ietf.org/html/rfc6838, 2018. [Online; accessed 06-February-2018].

[14] Hypertext Transfer Protocol – HTTP/1.1. https://tools.ietf.org/html/rfc2616, 2018. [Online; accessed 06-February-2018].

[15] Usage Statistics and Market Share of HTML for Websites, February 2018. https://w3techs.com/technologies/details/ml-html5/all/all, 2018. [Online; accessed 06-February-2018].

[16] HTML 5.2 W3C Recommendation, 14 December 2017. https://www.w3.org/TR/html52/, 2017. [Online; accessed 06-February-2018].

[17] WebIDL Level 1 W3C Recommendation 15 December 2016. https://www.w3.org/TR/WebIDL-1/, 2016. [Online; accessed 06-February-2018].

[18] WebIDL bindings Implementing WebIDL using Javascript . https://developer.mozilla.org/en-US/docs/Mozilla/WebIDL_bindings#Implementing_WebIDL_using_Javascript, 2018. [Online; accessed 06-February-2018].

[19] DOM Living Standard — Last Updated 3 February 2018. https://dom.spec.whatwg.org/, 2018. [Online; accessed 06-February-2018].

[20] Selectors Level 4 W3C Working Draft, 2 February 2018. https://www.w3.org/TR/selectors-4/, 2018. [Online; accessed 06-February-2018].

[21] Document Object Model XPath. https://www.w3.org/TR/2004/NOTE-DOM-Level-3-XPath-20040226/xpath.html, 2004. [Online; accessed 06-February-2018].

[22] Mutation observers DOM, Living Standard — Last Updated 3 February 2018. https://dom.spec.whatwg.org/#mutation-observers, 2018. [Online; accessed 06-February-2018].

[23] UI Events Legacy MutationEvent events, W3C Working Draft, 04 August 2016. https://www.w3.org/TR/uievents/#legacy-mutationevent-events, 2016. [Online; accessed 06-February-2018].

[24] Is there a Web 1.0? https://computer.howstuffworks.com/web-101.htm, 2018. [Online; accessed 06-February-2018].

[25] Tim O'reilly. What is web 2.0. http://www.oreilly.com/pub/a/web2/archive/what-is-web-20.html, 2005. [Online; accessed 01-February-2018].

[26] Wired Adobe finnaly kills Flash. https://www.wired.com/story/adobe-finally-kills-flash-dead/, 2018. [Online; accessed 01-February-2018].

[27] Microsoft Developer Silverlight Support Roadmap. https://blogs.msdn.microsoft.com/webapps/2014/01/16/silverlight-support-roadmap/, 2018. [Online; accessed 01-February-2018].

[28] Same-origin policy MDN web docs. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy, 2018. [Online; accessed 28-January-2018].

[29] WebDriver W3C Candidate Recommendation 30 March 2017. https://www.w3.org/TR/webdriver/, 2018. [Online; accessed 28-January-2018].

[30] ChromeDriver WebDriver for Chrome. https://sites.google.com/a/chromium.org/chromedriver/, 2018. [Online; accessed 06-February-2018].

[31] Ghost Driver. https://github.com/detro/ghostdriver, 2018. [Online; accessed 06-February-2018].

[32] Evan Sangaline. It is *not* possible to detect and block chrome headless. https://intoli.com/blog/not-possible-to-block-chrome-headless/, 2018. [Online; accessed 06-February-2018].

[33] Evan Sangaline. JavaScript injection with selenium, puppeteer, and marionette in chrome and firefox. https://intoli.com/blog/javascript-injection/, 2018. [Online; accessed 06-February-2018].

[34] Kubernetes Production-Grade Container Orchestration. https://kubernetes.io/, 2018. [Online; accessed 20-March-2018].

[35] RabbitMQ. https://www.rabbitmq.com/, 2018. [Online; accessed 06-February-2018].

[36] PostgreSQL The world's most advanced open source database. https://www.postgresql.org/about/, 2018. [Online; accessed 06-February-2018].

[37] Cybertec Why favor postgresql over MariaDB / MySQL. https://www.cybertec-postgresql.com/en/why-favor-postgresql-over-mariadb-mysql/, 2018. [Online; accessed 06-February-2018].

[38] Sequelize. http://docs.sequelizejs.com/, 2018. [Online; accessed 20-March-2018].

[39] Sriram Raghavan and Hector Garcia-Molina. Crawling the hidden web. Technical report, Stanford, 2000.

[40] GNU Wget. https://www.gnu.org/software/wget/, 2018. [Online; accessed 11-February-2018].

[41] How to crawl website with Linux wget command. http://www.tupp.me/2014/06/how-to-crawl-website-with-linux-wget.html, 2018. [Online; accessed 11-February-2018].

[42] curl command line tool and library for transferring data with URLs. https://curl.haxx.se/, 2018. [Online; accessed 11-February-2018].

[43] Rhino. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino, 2018. [Online; accessed 30-March-2018].

[44] JSDom Executing scripts. https://github.com/jsdom/jsdom#executing-scripts, 2018. [Online; accessed 30-March-2018].

[45] WebKit A fast, open source web browser engine. https://webkit.org/, 2018. [Online; accessed 30-March-2018].

[46] Puppeteer Headless Chrome Node API. https://github.com/GoogleChrome/puppeteer, 2018. [Online; accessed 11-February-2018].

[47] Github ariya/phantomjs Issue: Archiving the project: suspending the development. https://github.com/ariya/phantomjs/issues/15344, 2018. [Online; accessed 30-March-2018].

[48] Justin F Brunelle, Michele C Weigle, and Michael L Nelson. Archiving Deferred Representations Using a Two-Tiered Crawling Approach. *arXiv preprint arXiv:1508.02315*, 2015.

[49] Heritrix Wiki page. https://webarchive.jira.com/wiki/spaces/Heritrix/overview, 2018. [Online; accessed 11-February-2018].

[50] Bucharest Stock Exchange. The Stock Exchange is for the people! http://www.bvb.ro/.

[51] bezrealitky.cz prodej a pronájem nemovitostí bez provize. https://www.bezrealitky.cz/.

[52] Bloomberg European Edition. https://www.bloomberg.com/europe.

[53] Sbazar.cz Bazar a inzerce zdarma. https://www.sbazar.cz/, 2018. [Online; accessed 27-April-2018].

[54] Amazon.com Online Shopping for Electronics, Apparel, Computers, Books, DVDs & more. https://www.amazon.com/, 2018. [Online; accessed 27-April-2018].

[55] Alberto HF Laender, Berthier A Ribeiro-Neto, Altigran S Da Silva, and Juliana S Teixeira. A brief survey of web data extraction tools. *ACM Sigmod Record*, 31(2):84–93, 2002.

[56] Suryakant Choudhary, Mustafa Emre Dincturk, Seyed M Mirtaheri, Ali Moosavi, Gregor Von Bochmann, Guy-Vincent Jourdan, and Iosif Viorel Onut. Crawling Rich Internet Applications: the state of the art. In *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*, pages 146–160. IBM Corp., 2012.

[57] Tomáš Novella. Web Data Extraction, Master Thesis, Univerzita Karlova, Matematicko-fyzikální fakulta. 2016.

[58] Chun-Nan Hsu and Ming-Tzung Dung. Generating finite-state transducers for semi-structured data extraction from the web. *Information systems*, 23(8):521–538, 1998.

[59] Nicholas Kushmerick, Daniel S Weld, and Robert Doorenbos. Wrapper induction for information extraction. 1997.

[60] Nilesh Dalvi, Ravi Kumar, and Mohamed Soliman. Automatic wrappers for large scale web extraction. *Proceedings of the VLDB Endowment*, 4(4):219–230, 2011.

[61] Tomas Grigalis and Antanas Čenys. Generating XPath expressions for structured web data record segmentation. *Information and Software Technologies*, pages 38–47, 2012.

[62] Tomas Grigalis and Antanas Čenys. Unsupervised structured data extraction from template-generated web pages. *Journal of Universal Computer Science (J. UCS)*, 20(3):169–192, 2014.

[63] Tim Furche, Georg Gottlob, Giovanni Grasso, Christian Schallhart, and Andrew Sellers. OXPath: A language for scalable data extraction, automation, and crawling on the deep web. *The VLDB Journal*, 22(1):47–72, 2013.

[64] Yanhong Zhai and Bing Liu. Web data extraction based on partial tree alignment. In *Proceedings of the 14th international conference on World Wide Web*, pages 76–85. ACM, 2005.

[65] Nitin Jindal and Bing Liu. A generalized tree matching algorithm considering nested lists for web data extraction. In *Proceedings of the 2010 SIAM International Conference on Data Mining*, pages 930–941. SIAM, 2010.

[66] Shuyi Zheng, Ruihua Song, Ji-Rong Wen, and Di Wu. Joint optimization of wrapper generation and template detection. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 894–902. ACM, 2007.

[67] Tomas Gogar, Ondrej Hubacek, and Jan Sedivy. Deep Neural Networks for Web Page Information Extraction. In *IFIP International Conference on Artificial Intelligence Applications and Innovations*, pages 154–163. Springer, 2016.

[68] Cristian Duda, Gianni Frey, Donald Kossmann, and Chong Zhou. Ajaxsearch: crawling, indexing and searching web 2.0 applications. *Proceedings of the VLDB Endowment*, 1(2):1440–1443, 2008.

[69] Cristian Duda, Gianni Frey, Donald Kossmann, Reto Matter, and Chong Zhou. Ajax crawl: Making ajax applications searchable. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 78–89. IEEE, 2009.

[70] Crawljax Crawling Ajax-based Web Applications. `http://crawljax.com/`, 2018. [Online; accessed 28-January-2018].

[71] Suryakant Choudhary, Mustafa Emre Dincturk, Gregor V Bochmann, Guy-Vincent Jourdan, Iosif Viorel Onut, and Paul Ionescu. Solving some modeling challenges when testing Rich Internet Applications for security. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 850–857. IEEE, 2012.

[72] Suryakant Choudhary, Mustafa Emre Dincturk, Seyed M Mirtaheri, Gregor von Bochmann, Guy-Vincent Jourdan, and Iosif-Viorel Onut. Model-Based Rich Internet Applications Crawling: "Menu" and "Probability" Models. *J. Web Eng.*, 13(3&4):243–262, 2014.

[73] PhantomJS Full web stack No browser required. `http://phantomjs.org/`, 2018. [Online; accessed 28-January-2018].

[74] Electron Build cross platform desktop apps with JavaScript, HTML, and CSS. `https://electronjs.org/`, 2018. [Online; accessed 28-January-2018].

[75] React A JavaScript library for building user interfaces. `https://reactjs.org/`, 2018. [Online; accessed 28-January-2018].

[76] Material-UI React components that implement Google's Material Design. `http://www.material-ui.com/`, 2018. [Online; accessed 28-January-2018].

[77] Next.js Framework for server-rendered or statically-exported React apps. `https://github.com/zeit/next.js/`, 2018. [Online; accessed 28-January-2018].

[78] TypeScript JavaScript that scales. `https://www.typescriptlang.org/`, 2018. [Online; accessed 28-January-2018].

[79] React Redux Official React bindings for Redux. https://github.com/reactjs/react-redux, 2018. [Online; accessed 28-January-2018].

[80] Redux Redux is a predictable state container for JavaScript apps. https://redux.js.org/, 2018. [Online; accessed 28-January-2018].

[81] Redux Form The best way to manage your form state in Redux. https://redux-form.com/, 2018. [Online; accessed 28-January-2018].

[82] redux-observable RxJS 5-based middleware for Redux. Compose and cancel async actions to create side effects and more. https://redux-observable.js.org/, 2018. [Online; accessed 28-January-2018].

[83] Node.js. https://nodejs.org/, 2018. [Online; accessed 28-January-2018].

[84] Express Node.js web application framework. https://expressjs.com/, 2018. [Online; accessed 28-January-2018].

[85] ws Simple to use, blazing fast and thoroughly tested WebSocket client and server for Node.js. https://github.com/websockets/ws, 2018. [Online; accessed 28-January-2018].

[86] RxJS 5 A reactive programming library for JavaScript. https://github.com/ReactiveX/rxjs, 2018. [Online; accessed 28-January-2018].

[87] WEBDRIVERI/O WebDriver bindings for Node.js. http://webdriver.io/.

[88] WebDriver, W3C Candidate Recommendation 30 March 2017 Execute script. https://www.w3.org/TR/webdriver/#execute-script.

[89] RabbitMQ Management HTTP API. https://rawcdn.githack.com/rabbitmq/rabbitmq-management/v3.7.2/priv/www/api/index.html.

[90] RabbitMQ Management Plugin Statistic interval. https://www.rabbitmq.com/management.html#statistics-interval.

[91] Choose an open source license. https://choosealicense.com/.

[92] rafaelw/mutation-summary rafaelw/mutation-summary. https://github.com/rafaelw/mutation-summary.

[93] Hasura An Exhaustive Guide to Writing Dockerfiles for Node.js Web Apps. https://blog.hasura.io/an-exhaustive-guide-to-writing-dockerfiles-for-node-js-web-apps-bbee6bd2f3c4.

[94] Welcome to tmux! https://github.com/tmux/tmux/wiki.

[95] iTerm2: iTerm2 is a terminal emulator for MacOS that does amazing things. https://www.iterm2.com/.

[96] Hot Module Replacement Webpack documentation. https://webpack.js.org/concepts/hot-module-replacement/.

[97] Scrapy An open source and collaborative framework for extracting the data you need from websites. In a fast, simple, yet extensible way. https://scrapy.org/, 2018. [Online; accessed 09-February-2018].

[98] Selectors Scrapy 1.5.0 documentation. https://docs.scrapy.org/en/latest/topics/selectors.html, 2018. [Online; accessed 1-April-2018].

[99] Splash Lightweight, scriptable browser as a service. https://scrapinghub.com/splash, 2018. [Online; accessed 1-April-2018].

[100] Portia Visual scraping with Portia. https://scrapinghub.com/portia, 2018. [Online; accessed 1-April-2018].

[101] Import.io Magical new tool: The fastest way to get data from the web. https://www.import.io/post/magical-new-tool-the-fastest-way-to-get-data-from-the-web/, 2018. [Online; accessed 5-April-2018].

[102] Diffbot Turn Websites Into Data in Seconds. https://www.diffbot.com/, 2018. [Online; accessed 3-April-2018].

[103] Xconomy Diffbot Is Using Computer Vision to Reinvent the Semantic Web. http://www.xconomy.com/san-francisco/2012/07/25/diffbot-is-using-computer-vision-to-reinvent-the-semantic-web/#, 2018. [Online; accessed 3-April-2018].

[104] Xconomy: Diffbot Is Using Computer Vision to Reinvent the Semantic Web. https://www.xconomy.com/san-francisco/2012/07/25/diffbot-is-using-computer-vision-to-reinvent-the-semantic-web/3/.

[105] Wontae Choi, George Necula, and Koushik Sen. Guided GUI testing of Android apps with minimal restart and approximate learning. In *Acm Sigplan Notices*, volume 48, pages 623–640. ACM, 2013.

[106] Andrej Karpathy. Mini World Of Bits benchmark. http://alpha.openai.com/miniwob/index.html, 2016. [Online; accessed 15-August-2017].

[107] TypeDoc A documentation generator for TypeScript projects. http://typedoc.org/.

[108] Swagger: World's Most Popular API Framework. https://swagger.io/.

[109] FindLaw: Controversy Surrounds Screen Scrapers: Software Helps Users Access Web Sites But Activity by Competitors Comes Under SCrutiny . http://corporate.findlaw.com/law-library/controversy-surrounds-screen-scrapers-software-helps-users.html.

[110] Quora: What is the legality of web scraping? https://www.quora.com/What-is-the-legality-of-web-scraping.

[111] Salvador Rodriguez. U.S. judge says LinkedIn cannot block startup from public profile data. http://www.reuters.com/article/us-microsoft-linkedin-ruling-idUSKCN1AU2BV?il=0, 2017. [Online; accessed 15-August-2017].

[112] Reuters, Factbox: Who is Cambridge Analytica and what did it do? https://www.reuters.com/article/us-facebook-cambridge-analytica-factbox/factbox-who-is-cambridge-analytica-and-what-did-it-do-idUSKBN1GW07F.

[113] Wikipedia, Web scraping, Legal issues. https://en.wikipedia.org/wiki/Web_scraping#Legal_issues.

[114] Mike Thelwall and David Stuart. Web crawling ethics revisited: Cost, privacy, and denial of service. *Journal of the Association for Information Science and Technology*, 57(13):1771–1779, 2006.

[115] About /robots.txt In nutshell. http://www.robotstxt.org/robotstxt.html.

[116] ArsTechnica: Some websites turning law-abiding Tor users into second–class citizens. https://arstechnica.com/tech-policy/2016/02/some-websites-turning-law-abiding-tor-users-into-second-class-citizens/.

Part VII

Attachments

A. Electronic attachments

This thesis has attached electronic materials with source codes for proposed tool and electronic version of this text. Whole electronic attachment is folder, which is organized as follows:

- `/Readme.MD`

  Description of content of electronic attachment folder.

- `/source/`

  Folder containing all source for whole solution additionally all script used for building Docker images and running solution.

- `/source/backend/`

  Source code for back end. See section 12.1 for more details.

- `/source/frontend/`

  Source code for front end. See section 11.1 for more details.

- `/source/deployment/`

  Folder which contains `Dockerfiles` for creation Docker images with back end and front end. This folder contains `deployment.yml` file used for Kubernetes deployment.

- `/source/docs/`

  This folder contains documentation for source code generated using TypeDoc [107]. This documentation contains comments for all class and functions used in solution. Folder also contains Swagger [108] description of REST API exposed by our solution.

- `/thesis/`

  Source LaTeX files for this text. These source files can be build using `/thesis/Makefile`. This folder also contains PDF version of this text in file `/thesis/master_thesis_fejfar_2018.pdf`.

B. Legality and ethics of crawling

Web crawlers usually crawls data publicly available on Internet. This implies, that crawler cannot obtain data, which would not be obtainable by regular user. However owners of Web pages often claims right to content they share and they do not want their data to be processed in massive and automated manner. This creates controversy, because crawlers can be used by commercial entities to gain competitive advantage, however legislation is in this matter unclear if this behavior is illegal [109]. Good starting point for reader on this topic is this Quora thread `https://www.quora.com/What-is-the-legality-of-web-scraping`. In nutshell scraping for indexing is usually tolerated behavior, however scraping for "such as denial of service attacks, competitive data mining, online fraud, account hijacking, data theft, stealing of intellectual property, unauthorized vulnerability scans, spam, and digital ad fraud" [110] is considered bad. We leave reader to decide interpretation of following legal/ethics questions.

There are examples of judgment, which allow companies to crawl publicly available data [111], however whole topic still creates enormous controversy. One of recent example is company Cambridge Analytica [112], which took attention by using dataset of personal information about enormous count of users to provide insight to these user behavior. They are accused of interfering to presidential election in USA.

Note that legislation is different in every country and therefore crawlers needs respect origin of crawled Web page in order to satisfy legal constraints [113]. We advise to the users not to crawl Web page, if they do not have explicit permission from website content owners. We advise to the user to seek more information in work of Thelwall et al. [114] as this thesis is not focusing on this topic and cannot provide full view on the problematic.

Techniques for prevention of crawling exists, but from technical perspective they do not prevent is completely. The easiest option is to write term and condition on the Web site, which prohibits to crawl. Crawler's user need to respect these conditions to prevent crawling. More automatized approach is to expose standardized file `robot.txt` [115], which is machine readable by crawlers. Crawlers can read these files automatically before crawling Web page and stop crawling of parts of website, which is prohibited by `robot.txt`. These two techniques are not forced in any way and it is only on consideration of crawler if they follow them.

There are techniques to ban IP address from which crawler accesses the website. This techniques can be evade by using VPN, rotating IP addresses

or using Tor. Some website therefore restrict access from Tor network [116].

C. User guide

This document is intended for user of crawler to understand, how to control proposed crawler. This guide is showing all step needed to use this crawler tool to extract data from Web pages. Crawler let user define how to extract data from Web page and automate it.

Crawler is supported on Chrome v67 and newer on computers with resolution at least 1280x1024px. Note that other browsers will probably work as well, however crawler is not tested to run smoothly on these browsers.

Crawler can crawl HTML5 Web pages both static and dynamic (with client–side JavaScript). Crawler enables to define instructions, how to crawl end extract data from particular Web page. Extracted data can be obtained as CSV/JSON file or loaded from SQL table.

Note that some Web pages have term and condition, which prohibits automated crawling and data extraction. Although is can be not forced to prohibit pages by crawler, it may be illegal in some country to do so. In any case it is considered as fair/moral behavior to respect term and conditions of the Web page. More information can be found in Appendix B of this thesis.

C.1 Terminology

There are several often occurring terms used in this guide. Understanding of these term is important to effectively work with this guide.

Crawler is program which is used to automate interaction with a Web page. It is used for navigation on Web page and extraction data from the Web page.

Command is one specific instruction of crawler. For example it can be click on button or extraction of text from elements on Web page.

Page model is sequence of commands, which is defined by user. Page model is used by crawler as instructions for crawl data from Web page. Page model contains information about starting URL and what portion of data to export.

Crawler variables are used during running crawler to store value generated by commands. Variable can be used as input to other command. For example command which select element by CSS select returns id of this

elements, which can be stored to variable called “a”. Next command can click on element with id stored in “a”, therefore on element selected by CSS selector in previous step.

Crawler job represents running of crawler. Job can be started or stopped. Crawler job which is running is producing result, which can be downloaded by the user or stored in SQL DB.

Browser session Crawler is automating browser to obtain data from Web pages. When crawler is connected to browser it is open session, which is corresponds which one user interacting with browser. This session opens one browser tab and it is able to execute crawler runtime’s command. This session can be also mirrored to crawler tool UI in anytime.

C.2 Accessing solution

Crawler tool can be accessed using browser. Administrator of the system provide user URL on which is tool served. After loading this URL, user should see Welcome screen (Figure C.1). Welcome screen contains introduction information about tool and shortcut for creating new crawler in bottom right corner.

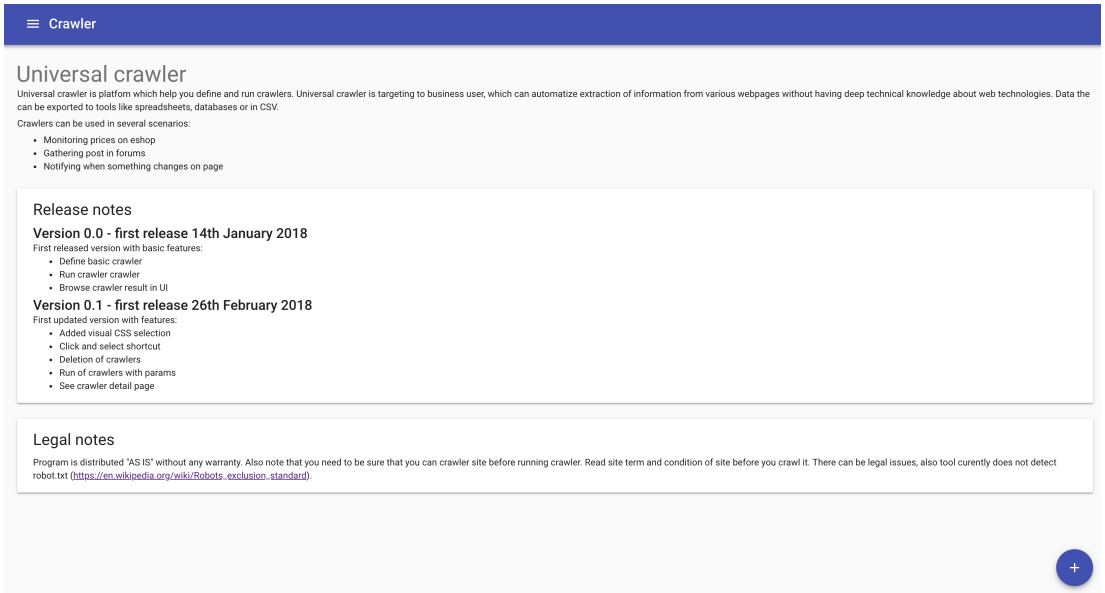


Figure C.1: *Welcome screen.*

In top left corner, there is a menu (Figure C.2), which can be accessed by hamburger button. Menu enables to the user navigate between main screens: crawler list, crawler jobs and browser session.

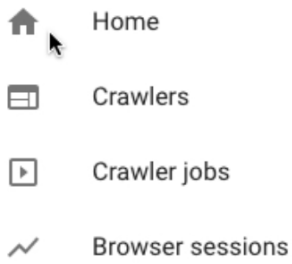


Figure C.2: *Menu*

C.3 Page model creation and definition

By click on plus button on welcome screen or plus button on crawler list screen, user can create new page model. Crawler will show dialog form (Figure C.3) with two text fields: Crawler name and default input. User will set these and click on create.

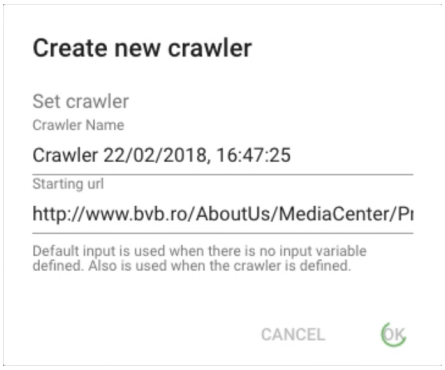


Figure C.3: *Add crawler dialog.*

Crawler then creates page model with instruction go to URL which is set to default input and navigate to crawler definition screen (Figure C.4).

User can see preview of the crawled web page on right side, exported variables and current variable state on the left upper side. On left bottom side, user can see sequence of commands of currently defined page model.

C.3.1 Adding command to page model

Page model is sequence of command, which are displayed in left panel during model definition (Figure C.5). Adding of new command is process user will spend most of the time of page model definition. Commands can be added manually or visually.

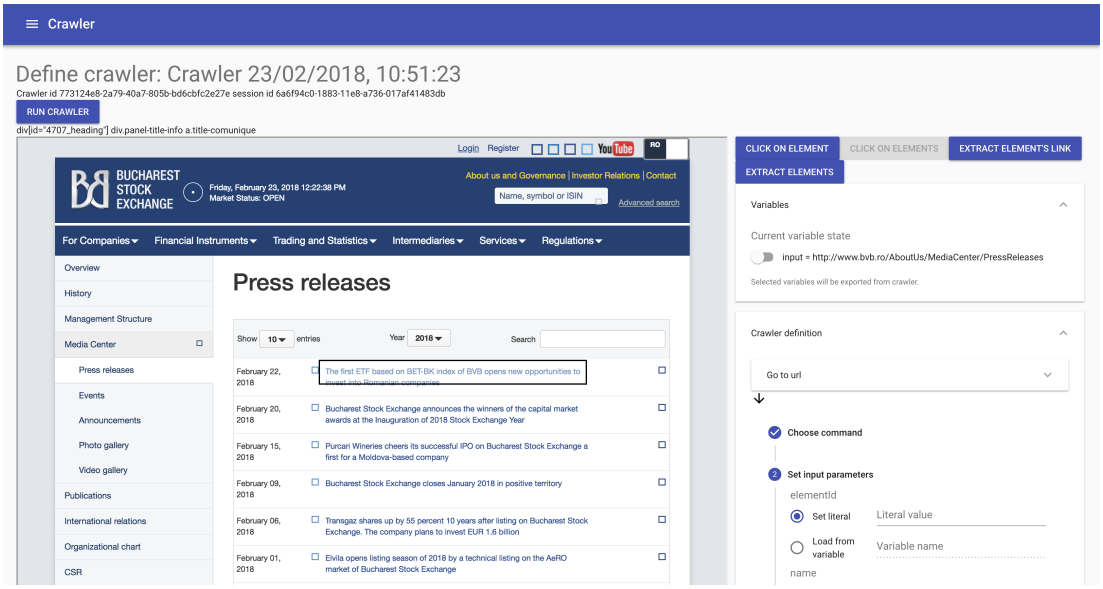


Figure C.4: *Crawler definition screen.*

By clicking on “add adv. command” user can set any command from Table C.1. This let user set all input and output parameters manually, which enable low level manipulation with Web page.

In command panel (Figure C.5), there is list of used variables. Variables shows its current value and can be selected for export by switch button. Exported variables are used as output of crawler.

If user clicks “yield result” command “yield” will be added to page model. When this command is executed during crawler runtime, crawler will yield new result with value of exported variables.

If user click “click on element”, “extract elements” or “extract element’s link” visual mode will be started. During visual mode (Figure C.6), user can select element he/she wants to interact in preview of the crawled page. Selected elements will be highlighted. CSS selector which is representing selected element/s is displayed in top above the page preview. When user confirm selection of the element series of new commands is added to page model. For example when “click on element” is invoked command “find elements” with input variable “locator” set to CSS selector is added to page model. Output of this command is stored in variable “lastClicked” and it is used as input for next command “element click”.

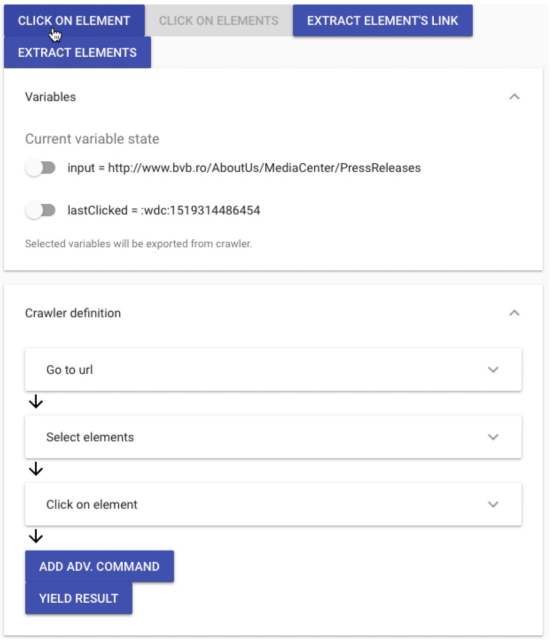


Figure C.5: *Crawler definition screen detail.*

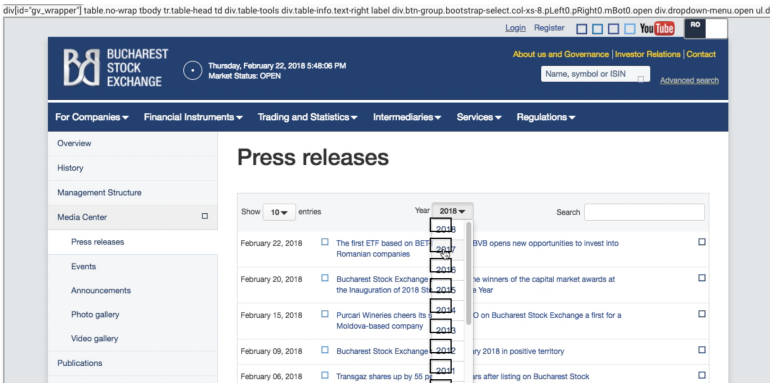


Figure C.6: *highlighting.*

C.4 Running crawler

Created crawlers can be showed on crawler list page (Figure C.7). Crawler can be run simply by hitting button “run”. Shortcut how to run crawler is from crawler definition screen (Figure C.4). When crawler is run, new crawler job is created. Progress of crawler job is displayed on crawler jobs screen (Figure C.8). Crawler can be stopped in crawler job detail page by hitting button “stop”.

Command name	Input variables	Output variables
Go to url	url	
Get url		url
Get title		title
Find element	locator	elementId
Find elements	locator	[elementId]
Find element from element	locator, elementId	elementId
Find elements from element	locator, elementId	[elementId]
Get element attribute	elementId, name	value
Get element text	elementId	text
Get element tag name	elementId	name
Get element rectangle	elementId	x, y, width, height
Element click	elementId	
Element send keys	elementId, keyCode	
Get page source		html
Executing script	jsCode	output
Take screenshot		screenshot
Take element screenshot	elementId	screenshot

Table C.1: *List of advanced commands.* Input variables needs to be supplied to command in order to execute action and obtain results.

C.5 Troubleshooting crawlers

On page browser session, user can find list of all opened browser sessions. These sessions can be opened and showed to the user (by page mirroring). Preview of browser session shows current state of crawled page, therefore user is able to determine what is problem with crawler, if any.

C.6 Exporting results

During runtime of crawler user can watch crawler status on crawler jobs page (Figure C.8). First five results are shown and total count of results as well. When crawler stops or when user decides there is enough results,

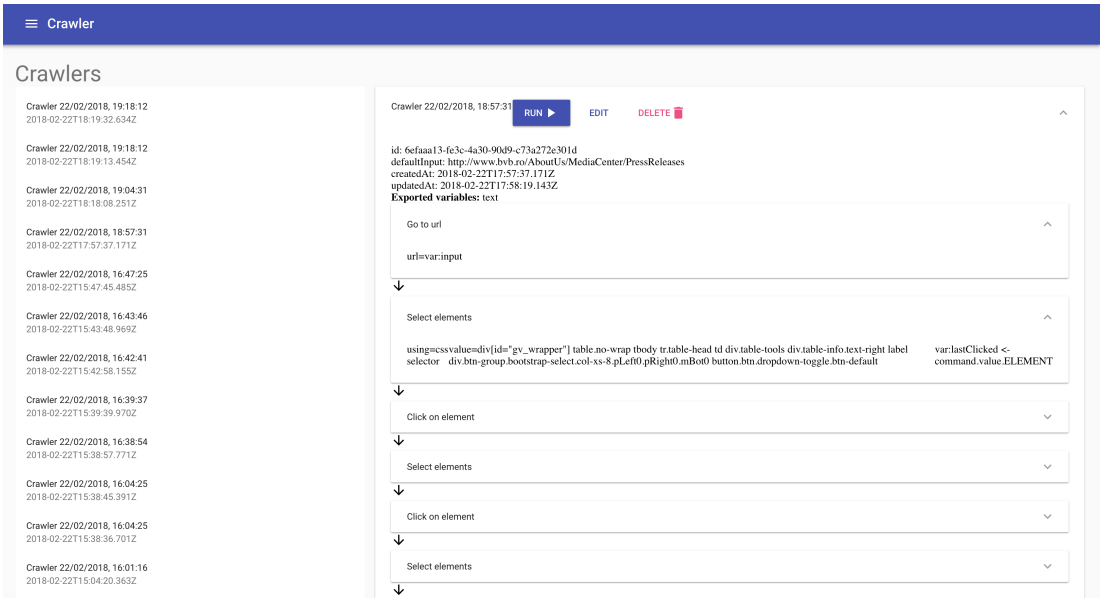


Figure C.7: *Crawler list screen.*

he/she can export data for further analysis or for checking that crawler works properly.

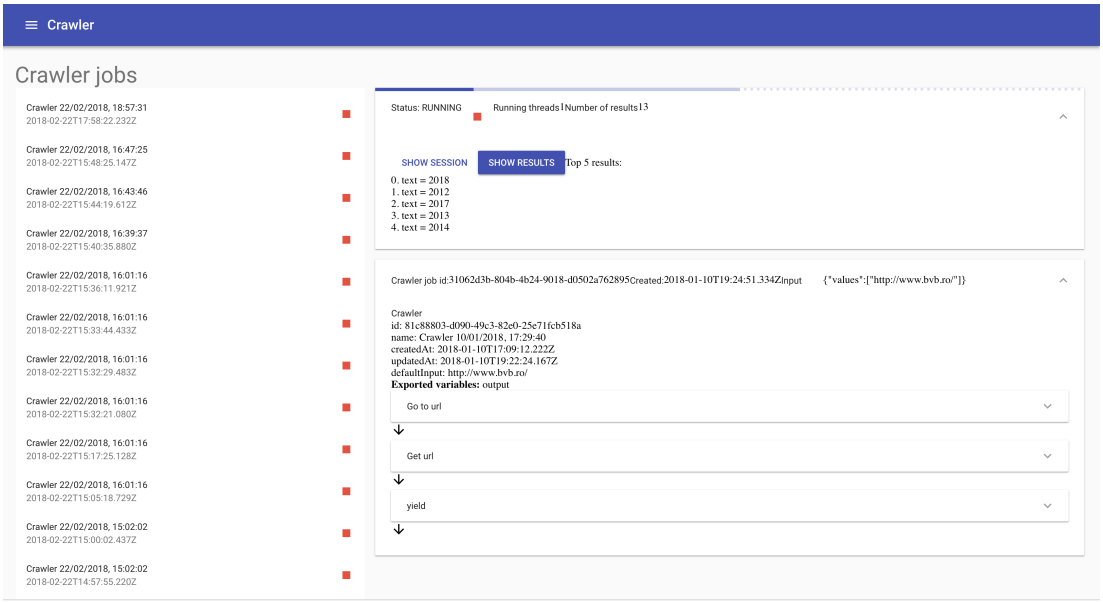


Figure C.8: *Crawler jobs screen.*

C.6.1 Export to CSV and JSON

In crawler job detail there is button “show results”, which will show results in specified format.

C.6.2 Export to SQL table

Most advanced export of crawled data is to use SQL table, which is used to store results. This is not typically done by end user of crawler, but rather by developer, who want to integrate crawler with other system. Table content can be retrieved using simple SQL query `SELECT * FROM results_<crawler_job_id>`. Credentials of SQL database will provide administrator of crawler.