BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

AUTOMATED PROCESSING OF WEBSITES REQUIRING JAVASCRIPT

AUTOMATIZOVÁNÉ ZPRACOVÁNÍ WEBOVÝCH STRÁNEK VYŽADUJÍCÍCH JAVASCRIPT

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **KAREL NOREK**
AUTOR PRÁCE

SUPERVISOR **Ing. DANIEL DOLEJŠKA**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

145378

Institut:	Department of Information Systems (UIFS)
Student:	**Norek Karel**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**Automated Processing of Websites Requiring JavaScript**
Category:	Networking
Academic year:	2022/23

Assignment:

1. Study, analyze and describe the issue of automated processing of web pages requiring JavaScript
2. Identify and compare the currently used methods for solving the problems of this work, i.e. mainly the software tools and technologies used.
3. Design a system enabling automated processing of web pages requiring JavaScript. Furthermore, the proposed solution must communicate through the interfaces provided by the thesis supervisor.
4. Implement the proposed system according to the supervisor's instructions.
5. Analyze, verify and adequately test the implemented system according to the supervisor's instructions.
6. Discuss the results obtained and their possible extensions.

Literature:

- DIOUF, Rabiyatou, SARR, Edouard Ngor, SALL, Ousmane, BIRREGAH, Babiga, BOUSSO, Mamadou and MBAYE, Seny Ndiaye, 2019. Web Scraping: State-of-the-Art and Areas of Application. In: *2019 IEEE International Conference on Big Data (Big Data)*. Online. Los Angeles, CA, USA: IEEE. December 2019. p. 6040–6042. [Accessed 20 October 2022]. ISBN 978-1-72810-858-2. DOI 10.1109/BigData47090.2019.9005594.
- GLEZ-PEÑA, Daniel, LOURENÇO, Anália, LÓPEZ-FERNÁNDEZ, Hugo, REBOIRO-JATO, Miguel and FDEZ-RIVEROLA, Florentino, 2014. Web scraping technologies in an API world. *Briefings in Bioinformatics*. September 2014. Vol. 15, no. 5, p. 788–797. DOI 10.1093/bib/bbt026.
- according to the supervisor's instructions

Requirements for the semestral defence:
Points 1, 2 and 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Dolejška Daniel, Ing.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	31.7.2023
Approval date:	26.10.2022

Abstract

This thesis focuses on the automated processing of websites requiring JavaScript. For this purpose, a scraper was created. The Scraper uses a configuration file containing a sequence of commands. These commands control a website using Selenium commands, extract wanted data, and store them in a database. This tool allows for building a complex flow on websites simulating user activity, mainly on websites using JavaScript. A web application for the scraper was also created, allowing configuration files to be created, running the scraper, and viewing data from the database. The solution provides correct results when processing data from dynamic websites and allows them to be shown in the web application. The benefit of this thesis is the option to process any website and store its data.

Abstrakt

Tato práce je zaměřena na automatizované zpracování dynamických webových stránek vyžadujících JavaScript. Pro tento záměr byl vytvořen scraper, který za pomocí konfiguračního souboru, obsahující sekvenci příkazů, ovládá webovou stránku použitím Selenia a extrahuje požadovaná data a ukládá je do databáze. Tento nástroj umožňuje sestavit komplexní sekvenci kroků simulující aktivitu uživatele na webových stránkách, především na stránkách využívajících JavaScript. Pro scraper byla také vytvořena webová aplikace umožňující vytváření konfiguračních souborů, spouštění scraperu a kontrolu dat v databázi. Výsledné řešení poskytuje správné výsledky při zpracování dat z dynamických stránek a jejich zobrazení ve webové aplikaci. Přínosem této práce je možnost zpracovat jakoukoliv webovou stránku a uchovat její data.

Keywords

Dynamic web scraping, Selenium, WebDriver, automated web processing, Blazor web app

Klíčová slova

Dynamická extrakce dat z webu, Selenium, WebDriver, automatické zpracování webových stránek, Blazor web app

Reference

NOREK, Karel. *Automated Processing of Websites Requiring JavaScript.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Daniel Dolejška

Rozšířený abstrakt

Úvod

Automatizované zpracování dynamických webových stránek vyžadujících JavaScript poskytuje způsob, kterým lze testovat a zpracovat jak dynamické, tak statické webové stránky. Jelikož v dnešní době se vyskytuje čím dál více dynamických stránek, je potřeba mít způsob, jak dané stránky zpracovat.

Cílem práce je vytvořit nástroj, který toto bude umožňovat. Webové stránky nemají danou obecnou strukturu, proto je nutné, aby vytvořený nástroj měl způsob, jak se adaptovat na každou stránku ve formě konfiguračních souborů. Dalším cílem této práce je vytvořit webovou aplikaci, která umožní data z nástroje sledovat. Dále také umožní vytváření konfiguračních souborů, aby je uživatel nemusel tvořit manuálně. Všechny tyto části spolu musí správně komunikovat a dodávat korektní výsledky.

Pro umožnění zpracování dynamických stránek je potřeba použít prohlížeč, aby se načetla všechna data z dané stránky, protože dynamické stránky používají jazyky na straně klienta, které se načtou pouze v prohlížeči. Jelikož je podstatné spustit prohlížeč z kódu, je nutné použít knihovnu, která toto umožňuje. V této práci byla za tímto účelem použitá knihovna Selenium. Prohlížeče, které se používají k tomuto účelu poskytují možnost Headless, která znamená, že prohlížeč nebude mít grafické rozhraní a bude tak šetřit zdroje.

Popis řešení

Pro zpracování dynamických stránek byl vytvořen nástroj, který za pomocí konfiguračního souboru, obsahující sekvenci příkazů, zpracovává požadovanou stránku. Každý příkaz v tomto souboru je jeden příkaz z knihovny Selenium jako například příkaz, který simuluje kliknutí. Tato knihovna umožňuje jednoduché ovládaní webových stránek z kódu za pomocí rozhraní WebDriver.

Nástroj projde daný konfigurační soubor a vykoná každý příkaz, který je v něm obsažen. Pokud má příkaz nějakou návratovou hodnotu, jako například text elementu, je tato hodnota uložena do databáze. Pokud je v konfiguračním souboru specifikované cyklení, nástroj pokračuje, dokud není ukončen. Jinak se soubor projde pouze jednou a nástroj končí.

Nástroj je vytvořen jako konzolová aplikace s možností nástroj buď spustit z příkazové řádky, nebo nástroj zavolat z jiného projektu.

Webová aplikace umožňuje zobrazení všech dat z databáze. Dále také umožňuje vytvoření nebo editování konfiguračních souborů. Z hlavní stránky aplikace také lze spustit nástroj na zpracování stránek s vybraným konfiguračním souborem.

Testování

Testování bylo rozděleno do částí: První testovaná část byly vytvořené třídy pro Selenium příkazy. Tato část byla testována pomocí unit testů. Každý unit test testoval pouze jeden příkaz. Tato část byla zcela úspěšná.

Další částí bylo otestovat vytvořenou webovou aplikaci. Testovaná byla všechna funkcionalita každé stránky. Všechny testy proběhly úspěšně.

Poslední částí bylo otestování celkového projektu. Testováno bylo několik webových stránek s různými konfiguračními soubory, aby byla zajištěna funkčnost na většině stránek. Tyto testy přinesly očekávané výsledky jak v databázi, tak i ve webové aplikaci.

Zhrnutí a budoucí práce

Nástroj na zpracování dynamických webových stránek ukázal přijatelné výsledky. Výsledný celek spolu komunikuje správně a dodává správné výsledky. Výsledná práce splňuje všechny požadavky ze zadání.

Budoucím rozšířením by byl nástroj, který by dokázal konvertovat testovací kroky generované pomocí Selenium IDE do konfiguračního souboru a ulehčit tak tvorbu konfigurací.

Dalším rozšířením by mohla být stránka v již vytvořené webové aplikaci, která by vytvořila statistiku k datům z databáze.

Automated Processing of Websites Requiring JavaScript

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Daniel Dolejška. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Karel Norek
July 26, 2023

Acknowledgements

I want to thank my work supervisor, without whom it would not have been possible to create this work. Thank you, Ing. Daniel Dolejška, for your time, guidance, and advice.

Contents

Chapter 1

Introduction

Web scraping is a method to extract data from websites. The resulting data can then be used for several different purposes. For example, the goal can be monitoring prices and news or even researching trends within the user group of a specific website [19]. Scraping static websites is fairly easy since they use plain HTML that can be parsed with numerous tools. Dynamic websites are a more significant problem because JavaScript or a similar language is available only after a website is loaded in the web browser [10].

This thesis is focused on dynamic web scraping and aims to be able to extract data from websites that require JavaScript to run correctly. That is done by using headless browsers which can handle this kind of task. These browsers act as standard browsers but can be used in a command line. To be able to use it in an application, one must use libraries that provide methods to control those browsers.

Even though the focus of this thesis is dynamic web scraping, there are other parts that help with handling results received from the scraping. One of those parts is a database for storing the results. Another part is a web application that allows users to better keep track of the results and provides a way to run the scraper and also build a configuration for it.

The thesis is divided into chapters, where each chapter covers a specific part.

For the theoretical part of web scraping in general and more detailed information for dynamic web scraping is dedicated chapter 2. This chapter also contains information about headless browsers and libraries to control them.

The proposed solution for the scraper and all other parts is in chapter 3. This chapter explains what libraries and frameworks are used for each part and why precisely those were chosen.

A detailed description of an implementation of this thesis is provided in chapter 4. A section is dedicated to each part with a detailed description of how it is implemented.

Testing of the implementation is in chapter 5. Testing is divided into sections, each covering a specific part of the project. All test cases that were used are described with test steps, and their results are also provided.

Chapter 2

Web Scraping

Web data scraping provides a way to extract or monitor data from websites. Nowadays, it is one of the most important things in web development since it can be used to test websites, analyze them, or even monitor those websites. Generally, Web data scraping can be defined as the process of systematically extracting and combining contents of interest from the Web. Web scraper mimics the web servers' and humans' browsing interaction in a conventional Web traversal [4]. The basic structure of scrapers consists of three main parts.

2.1 Basic Web Scraper structure

The first part of the Web Scraper structure is site access. Communication between the scraper and the Website is usually done through the HTTP protocol. This protocol works by sending requests to the Web server and waiting for responses. The most frequent methods in this communication are the GET and POST methods. The GET method retrieves data from the Web server. The POST method is used for sending data to the Web server. One of the most important things about those requests is the 'User-Agent' part of the request header. This specifies what kind of program is accessing the contents of the Web server. It is important since some Web servers can specify which resources should not be accessed by automatic procedures. [4]

The second part is HTML parsing and content extraction. In this part, the scraper reads the retrieved HTML document and extracts its contents of interest. For this purpose exists several ways. HTML parsing libraries are one of those ways. Using HTML parsing libraries and manually retrieving data is one of those ways. Another way is through selector-based languages. Website data must still be parsed, but these languages provide a better way to get wanted results. Selector-based languages are more popular nowadays since most websites are dynamic and plain HTML parsing is insufficient. One of the selector-based languages is XPath[1]. XPath is an expression language that allows the processing of values conforming to the data model. This data model provides a tree representation of XML documents and atomic values such as integers, strings, booleans, and sequences that may contain references to nodes in an XML document and atomic values [17].

The last part of a scraper is output building. That is the most flexible part of the scraper since it solely depends on the purpose of the said scraper. The scraper should transform extracted data into a structured representation that can be further analyzed and stored [4].

[1] XPath - XML Path Language

2.2 Different approaches in Web Scraping

Web Scraping follows one of the following approaches.

2.2.1 Mimicry Approach

The mimicry approach is a category of scrapers with predefined data locations. That makes location data in DOM fairly efficient but has some major flaws. One of those flaws is that scraper can be used only on one specific website since even small changes in the DOM can make it impossible to retrieve the same data. It can be easily solved by using configuration files that store the location of wanted data for each website separately. [3]

2.2.2 Weight Measurement Approach

Scraper with this approach uses a generic weight algorithm that analyzes the DOM tree. The algorithm then chooses a starting node and retrieves all data from child nodes. This approach can produce a lot of unwanted data to be extracted. [3]

2.2.3 Differential Approach

This approach works with websites that differ only in content from the body of the page. That means the website will have identical menu bars and footers and only vary in the body. The scraper will then apply a masking algorithm to build data from two website pages consisting only of differences. That ensures that the scraper extracts only data from the body and ignores menu bars and footers. [3]

2.2.4 Machine Learning Approach

The machine learning approach uses the classic principles of machine learning. It uses a large sample of manually analyzed web pages. The scraper is then able to statistically deduce where the wanted data is located and can extract it. [3]

2.3 Static and Dynamic Web Scraping

Web scrapers can also be divided into two categories. Static and dynamic web scrapers Since dynamic websites are becoming more popular, dynamic Web scrapers are better when extracting data from multiple generic websites. Static scrapers still have their use since some websites can even nowadays be static, or when creating scrappers for websites located on the dark web, where websites must be static in order to work.

2.4 Static Web Scraping

Static web scraping focuses on plain HTML and CSS. It is fairly easy to extract data from websites that use plain HTML and CSS.

Firstly scraper needs to access a website that should be scraped for data. Communication between scraper and website is done using basic HTTP protocol as mentioned in section 2.1. In programming languages, libraries exist that handle this communication. For example, Python has `requests` library, and C# has `Windows.Web.Http` library and JavaScript users can use `Axios` library. Through the GET method scraper gets the source

code of a chosen website. After that, data can be stored or used in some other way. This process is straightforward but isn't optimal nowadays since most websites use some dynamic elements, and scraping them this way could prove difficult and, in some cases, even impossible. That is why a new way of scraping websites was created: dynamic web scraping.

2.5 Dynamic Web Scraping

Dynamic web scraping is very different from static web scraping. Dynamic websites use several client-side languages that make scraping more difficult. If the same approach as for static websites is used, the extracted data could be incomplete or nonexistent. That is because some programming languages on the client side can be used to display data a certain way or get data from another source, like ads. Because of that, plain HTML source code without those additional languages would be incomplete. Another problem can occur when the website to be scraped has a lot of pages and links, so if someone wants to scrape a page that is hidden behind a link on the main page, they would need to scrape the main page, get the required link and scrape the next page. Dynamic web scraping solves those problems thanks to headless browsers. The problem is solved by enabling the user to control a website and providing a way to click on a link without retrieving the required link manually.

Dynamic websites today use programming languages that can enhance the website by running code in the background. That can be helpful when rendering some data, creating counters, and others. One of those languages is JavaScript. JavaScript is one of the most popular ways to enhance websites [16]. For JavaScript to work correctly on a website, the website must be loaded by any web browser [5]. After the website with JavaScript is loaded, data can now be extracted using typical scraping methods as mentioned in 2.4. The main problem is getting JavaScript to load from code or the command line. Since JavaScript needs to be loaded by a browser, it is necessary to have some browser that can be run from code or command line. For this purpose, headless option in browsers exist.

2.5.1 Headless Browsers

A headless browser is a browser that can be launched from the source code or the command line. This browser doesn't have a GUI (graphical user interface), so it performs faster since it doesn't need to load visual images, render DOM, CSS animations, and other resources. These browsers are mainly used for automation, testing, and data extraction.

There are three main browsers that have a headless option: Chrome[2], Firefox[3], and Safari[4]. To use those browsers in code, libraries are needed. Libraries can be made for a single language or multiple languages. For a scraper to be as universal as it can be, it should be able to use all the main headless browsers. Since most libraries use one of these, as seen in table 2.1, choosing those that can use multiple is best. Requirements like this make it easier to select a suitable library. The proper library needs to support all required browsers and also have good support for a programming language that will be used for a scraper. [6]

[2] https://www.google.com/chrome/ - [cited 2023-07-24]
[3] https://www.mozilla.org/en-US/firefox/new/ - [cited 2023-07-24]
[4] https://www.apple.com/safari/ - [cited 2023-07-24]

Usually, headless browsers are a special mode of standard browsers that can be turned on when creating a new browser session. Browser sessions are run by a WebDriver 2.5.3. Each of the already mentioned browsers has its own driver based on the general WebDriver.

Some examples of libraries for headless browsers can be seen in table 2.1. [2]

Name	Driver	Language
Puppeteer[1]	Chromium	JavaScript
PuppeteerSharp[2]	Chromium	C#
PhantomJS[3]	WebKit	JavaScript, Python, C#
Ultralight[4]	Chromium	C++, C#
Selenium[5]	Chromium, Firefox, WebKit	JavaScript, Python, C#
Playwright[6]	Chromium, Firefox, WebKit	TypeScript
playwright-dotnet[7]	Chromium, Firefox, WebKit	.NET

Table 2.1: Headless browser examples

[1] https://pptr.dev/ - [cited 2023-07-25]
[2] https://www.puppeteersharp.com/ - [cited 2023-07-25]
[3] https://phantomjs.org/ - [cited 2023-07-25]
[4] https://ultralig.ht/ - [cited 2023-07-25]
[5] https://www.selenium.dev/ - [cited 2023-07-25]
[6] https://playwright.dev/ - [cited 2023-07-25]
[7] https://playwright.dev/dotnet/ - [cited 2023-07-25]

From the table can be seen that a lot of libraries use just one of the main headless browsers. So developers must choose what kind of browsers they want to support and pick a library accordingly. Playwright is an up-and-coming library that could be the best solution for dynamic web scraping in the future, but since it is still one of the newer libraries, Selenium still holds its value.

2.5.2 Selenium

Selenium is a suite of tools and libraries enabling web browser automation. Specifically, it provides an infrastructure for the W3C WebDriver specification — a platform and language-neutral coding interface compatible with all major web browsers [14]. Selenium also supports a wide range of programming languages, making it one of the most popular tools for web browser automation. Because it is one of the most popular tools, numerous ways exist to use Selenium. For example, the Selenium IDE extension for browsers allows users to create steps for web browser automation [15]. When the user has prepared all of the test steps on a selected website, automation can then be exported into a chosen language, making it easier to create test cases. Another way is to use the already mentioned Selenium WebDriver, which implements the individual browser-controlling code. More information about WebDriver can be read in section 2.5.3. These things combined make Selenium one of the easiest to use and the most accessible tools for web browser automation.

2.5.3 WebDriver

WebDriver is a remote control interface that creates a way to control user agents and enables introspection. It provides a means to instruct the behavior of web browsers remotely [18]. It is a W3C recommendation, meaning that major browsers all support this interface, leading to a more uniform behavior across different browsers [14]. Using the WebDriver is relatively simple. All the user needs to do is install the Selenium library and create a new session. To create a new session, initialize a new Driver class from the code. The selected driver will open the browser depending on which driver class the user chooses. After creating a session, WebDriver provides a way to control the browser. That means the user can navigate to a chosen website, locate elements on that website, and perform actions like clicking or writing. WebDriver also provides a headless mode for some browsers, so it can also run on servers.

Chapter 3

Proposed solution for scraping dynamic websites

This chapter covers a proposed solution for dynamic web scraping, including what kind of frameworks will be used and reasons why.

A dynamic website, as mentioned in 2.5, uses plain HTML and CSS but can also use other languages like JavaScript. That creates a requirement for headless browsers and libraries that use them. This library must meet the required criteria.

Another problem that needs to be solved is which approach to choose. The most common approach is the mimicry approach because it is relatively easy to implement and has good results. Different approaches were covered in 2.2. This thesis will make use of the mimicry approach. The best way to ensure an easy scraping of multiple websites is to have some file that can serve as a guide for the scraper so it knows which elements to extract. Otherwise, the scraper would have to be hardcoded for a specific website, but the approach with a configuration file can be used to scrape any website of the user's choosing.

The configuration file needs to have a list of elements to extract. To be able to find those elements, some way to locate those elements needs to exist so the scraper knows how to get to the element. There are several options available for this purpose. One of the ways is to search elements by id, but that is not a reliable solution since some websites can use elements without an id, which could cause problems with extracting those elements. A better way is to use selector-based languages. There are two of those languages worth mentioning. First is the CSS selector. That is usually used when creating the CSS styling of websites, but it can also be used when wanting to extract elements. Another language is called `XPath`[1]. That is the ideal solution for dynamic scraping since it can locate almost anything on the website [17]. It is better to choose just one of these ways so there is clarity about which way to use and when.

The next thing to consider is handling received data from the scraper. Since the scraping provides results from the scraping, there needs to be some way to handle and store them. There are numerous ways to do this. The simplest way would be to create either XML or JSON files and fill that with the results. Even though this could work, it isn't an optimal solution because there is no easy way to edit those files afterward. It also prevents the usage of queries to create aggregations or transformations. So much better solution is to create an actual database for the data. Which database was chosen and why is described in section 3.4.

[1] XPath - XML Path Language

Now it comes to the actual scraper. There are multiple ways that the scraper can work. It can work as a one-time scrape, meaning it will scrape a website once, extract its data, store them, and end it. Another way is for the scraper to run continuously in an infinite loop at specified intervals and scrape a website in each loop. Since both methods are beneficial, it is better to include them both with an option to switch between them.

The next step is how to store the data. The scraping will most likely be done more than once. That means the same elements will be extracted from the website multiple times. Now either the received data can be overwritten, or they can be stored as a new record. Since the scraper can be used to monitor some data on a website, it is better not to overwrite the old data because then the new data wouldn't have any means to compare them to the old data.

The last thing is to propose a web application that will help users use the scraper. The initial thought for the web application is for the users to be able to view the data from a database and some way to create a configuration file so it doesn't have to be done manually. Another thing worth considering is to have some way to run the scraper from this web application to make it easier to use. A detailed description of which framework will be used and why is in section 3.6.

That concludes every significant decision that needs to be made when creating scrapers for dynamic websites. In the sections below are written selected frameworks and a more detailed description of a proposal. Actual details of the implementation are in the chapter 4.

3.1 Chosen library for headless browsers

This thesis will use the Selenium NuGet package available for C#[2]. Selenium was chosen for its compatibility with all main drivers so the users can choose whichever they deem the most appropriate for their chosen website to be scrapped. Selenium is one of the most popular for website automation. It also includes a way to locate elements and retrieve their values easily. That makes it a good choice for the scraper.

Another positive thing is that a Selenium IDE exists, which can help users create configuration files because it can write a sequence of steps that the user will transcribe into the configuration file with minor differences.

Selenium also provides easy control of a website with simple commands that can be easily executed.

3.2 Configuration file

This section covers the configuration file. The JSON format was chosen for the file format because it provides all necessary functionality like storing the file as a simple format, possible serialization, and deserialization into C# classes and others. The proposed solution for the file is to have fields for specific URLs, loop option, and driver types. After that, there will be an array of commands. Each command should contain its name and necessary arguments. That will help maintain a sequence of commands in their order, and each command will be identifiable by its name and have all required arguments present.

To quickly validate if the configuration file is valid, there exists a declarative language that allows to annotate and validate JSON documents called JSON schema [9]. JSON

[2]Selenium NuGet package - https://www.nuget.org/packages/Selenium.WebDriver

schema is a way to create templates for certain JSON files. For each field in the schema, there is a way to specify the field type. Another thing that can be done is to specify the format of a string. For example, for the field URL used in this project, it can be specified that the format will be URI. The next possible thing is to create an enumerable field. That can be used for the driver field. The enumerable field will help check correct values so the users don't type in a wrong value. Next, useful things are the `anyOf`, `allOf`, `oneOf` keywords. Those specify how many things can be present from the specification. This example can be used for better understanding: If the schema contains definitions for three different objects and they are inside a `oneOf` keyword, the final JSON file can only contain one of those objects. If the keyword is `anyOf`, the final file can contain any combination of them. [9]

The JSON library contains a method that takes a schema as an argument and uses it to validate the JSON file on which the method was used.

With these propositions, the configuration files should contain all needed information for the scraper to scrape any website.

3.3 Locating elements

Locating elements on a website can be tricky. As mentioned before, there are many ways to locate some elements, but for the scraper to work, it needs to be able to find any element on any given website. So that means the way of locating elements by ID is not optimal and will not be used.

Now what remains is to choose between the CSS locator and the XPath. The CSS locator can sometimes be better for the location of certain elements, and even the Selenium IDE usually offers this way first as can be seen in figure 3.1. Still, it could provide some difficulties when using different browsers. For example, the Chrome browser uses RGBa for colors in CSS, but the older versions of the Firefox browser don't support RGBa [1]. That means if the locator is used for the RGBA when using an older version of Firefox, it could provide problems.

That leaves the XPath way. It provides an easy way to select an element. Selenium has a method that takes an XPath string as an argument, and this method then locates the element using this argument. More about XPath and how to build the correct XPath string is in the section 3.3.1.

3.3.1 XPath

The XPath language provides a flexible way of addressing different parts of an XML document. Even though it is mainly used for XSLT documents, it can also be used as a much more powerful way of navigating through the DOM. [17]

XPath uses a path notation to identify and navigate nodes in an XML document. XPath has numerous built-in functions that help when creating a functional XPath. Here are some examples of how to create an XPath expression.

- **//a[@id='test']** - This XPath looks for element **a** anywhere on a website with an ID with the value "test".
- **//p[contains(text(), 'value')]** - This XPath looks for element **p** anywhere on a website that contains a text "value".

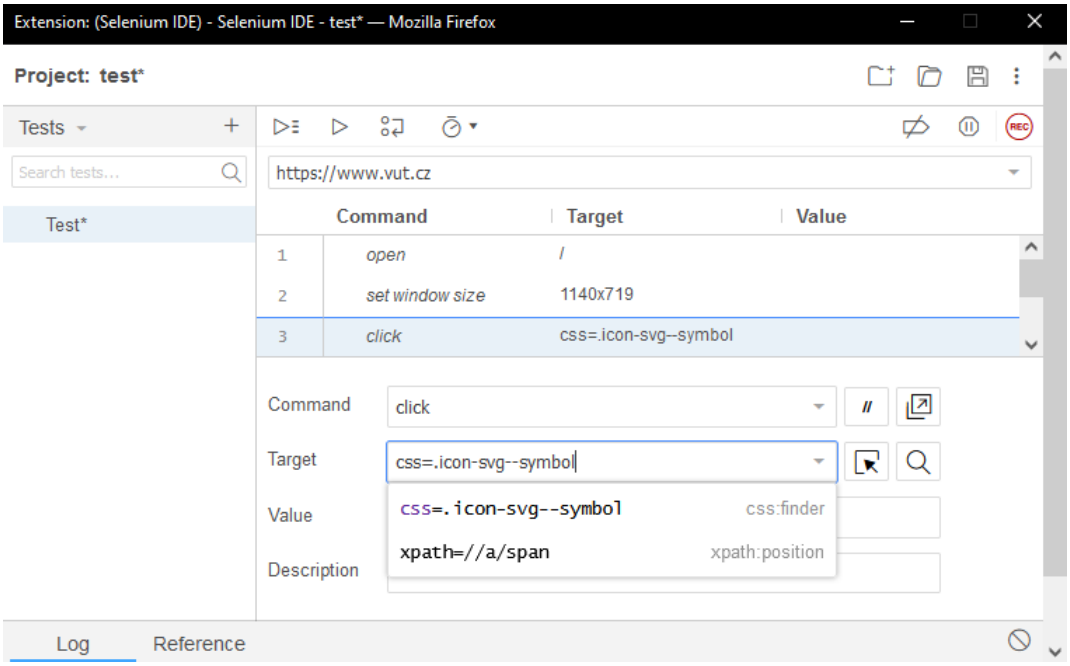

Figure 3.1: SeleniumIDE - CSS locator

For more details on XPath syntax, visit the page https://www.w3schools.com/xml/xpath_syntax.asp.

Using XPath will make it easy to locate any element specified in a configuration file so the scraper can work correctly.

3.4 Database

For the database will be used Entity Framework. It provides a way to build a clean high-level data access layer. Several supported databases like SQLite[3], MySQL[4], and PostgreSQL[5] for Entity Framework exist. Since this project needs only two tables, using a local SQLite database is more than enough. [8]

The database access layer will be implemented with Unit of Work and Repository design patterns [11]. This layer is going to be accessed by using a Facade design pattern [13].

There will be two tables, one for the websites and the other for all elements. The elements table will have a foreign key that will bind it to a specific website in the other table.

3.5 Scraper

The proposal for the actual scraper is simple. The whole application will be created as a hosted console service. That service will check the arguments with which the scraper was run. It will also create a JSON schema and validate a configuration file given to the

[3] https://www.sqlite.org/index.html
[4] https://www.mysql.com/
[5] https://www.postgresql.org/

scraper. After these tasks are done, the scraper will execute every command present in a configuration file and, if needed, save them to the database.

Functionality from Selenium will be implemented using the command behavioral design pattern [12]. This pattern was chosen because all commands will have the same execute method, so they can inherit it from the same interface and only change the implementation of the method.

Each different way to control a website will be made into a single command class which will serve as a wrapper for the actual Selenium command. Each class will have an execute method which will serve as a way to execute the command. Commands with return value will return its value as a string, while the others will return null.

3.6 Web application

Since all of the other parts of this project are implemented in C#, Blazor framework[6] provides a way to keep everything in one single language, not including the HTML for the actual website.

Blazor is a framework for building client-side web UI with .NET. It can create interactive UIs using C# instead of JavaScript. Since it is in C#, it can also use all of its libraries or even other projects implemented in the same language. That can prove helpful when binding the scraper to the website. [7]

Blazor website is made of components. The components are elements of UI like tables, dialogs, alerts, and others. These components make it easier to provide functionality for the users without a need to create that from scratch. [7]

This framework also allows the usage of JavaScript when needed, so the users don't lose any functionality when using it.

There is also a library called `MudBlazor`[7] that further deepens the functionality and makes even better components, so using the Blazor framework with this library is the best choice for the web application. The list of all possible components included in `MudBlazor` can be read in the documentation: `https://mudblazor.com/docs/overview`. Each of the `MudBlazor` components has its own styling, making a website look much better.

Now for the actual web application for this project. The initial plan is to have pages for each database table, a page for building a configuration file, and a simple main page.

The database pages will each have a table that will show the data. Since it is expected that there will be a lot of records in each table, there will be some filtering functionality to provide a better user experience. Another good thing would be to have a way to delete selected records because the scraper won't have a way to delete them. That means there must also be a way to choose which records should be deleted.

The configuration page will be the most complex one. It must provide a way to select values for each field that can exist in a configuration file. Since the fields can be divided into two parts, it will be better to make them separately and then put them together.

The first part is the fields that will be present every time. Those include URL, loop, and driver fields. This part is going to be done as simple form fields. Because only three drivers are available, it will be best to have them as a selection box so the users can avoid accidentally typing in the wrong driver.

[6] `https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor`
[7] `https://mudblazor.com`

The second part is a list of commands. This part must be flexible since there are no rules that specify the number of commands or even which commands will be used. So it would be best to have a button that will add a command. After a command is added, there must be a way to fill out all necessary properties. To be able to do this, the property fields are going to be generated after a name of a command is filled.

There must also exist a way to save the created configuration. Before saving a configuration, there is going to be a check that will validate the configuration.

To enhance the user experience, there will be a way to edit already existing configuration files.

Chapter 4

Implementation

This section is focused on the implementation of the scraper. It consists of three main parts: the scraper, the database, and the web application. Since all those parts are implemented in C# programming language, they are all part of the same solution divided into projects.

The web application is implemented in the Blazor framework that allows the creation of an interactive client-site web user interface. Thanks to this framework, the whole project uses the same programming language and can use methods from other parts like the database and the scraper. That means the web application can use facades from the database without creating a new one, and calling the scraper can be done by creating a task to run the scraper. More details can be read in 4.3.

The core part is the scraper project. This part handles the dynamic scraping of chosen website. This project also includes wrappers for Selenium commands, handling the configuration files, creating JSON schema for configuration files, and storing the scraping results in the database. A detailed description is in section 4.1.

The last part is the database project. This part handles working with an SQLite database using the Repository, Facades, and Unit of Work patterns 4.2.

Additionally, there is a project in the solution for testing the Selenium wrappers. This project contains only one file **CommandsTests.cs**, where all the tests are present.

All chosen libraries and frameworks and reasons for choosing them were already mentioned in the proposal section 3. Correct behavior is tested for this implementation in section 5.

4.1 WebScraper

The scraper project is a standalone application that can be run from the command line, or it can be called as a method which can be seen later in web application implementation 4.3. This project can be divided into multiple small logical parts.

4.1.1 Selenium WebDriver and Selenium commands

The first part is the creation and handling of the Selenium WebDriver. Into this part can also be included the wrappers for Selenium commands since they interact with the created WebDriver. Selenium WebDriver has its own class that handles creating and deleting the WebDriver (later only driver). When creating the driver, a specific value that represents the expected type of the driver is taken from the configuration, and according to it, the driver uses either Chrome, Firefox, or Safari subclasses of the driver. For the Chrome and Firefox

drivers, it is possible to create it with specific options. Those options are the headless and the maximize options, but the maximize option is only available for the Chrome driver because the Firefox driver doesn't support it. Safari has yet to have an available class for making those options at the time of writing this thesis. After successfully creating the driver, its URL is set to the one taken from the configuration file. Deletion of the driver is done by calling method `Quit` for the driver, which disposes of it and doesn't cause memory leaks after the program ends.

Selenium commands have their own namespace `WebScraper.SeleniumCommands`. This allows library `System.Reflection` to filter all classes inside this namespace for generating JSON schema for the configuration files. Each command inherits the interface `ICommand`. This interface has abstract method `Execute`, which is later used in each command for execution. The method is of type *ValueTask* because *ValueTask* results are stored on the stack wherever the *Task* type results are stored on the heap. So when the results go out of scope, the memory is cleaned. This method takes only one parameter driver, specifying with which driver the method should be executed. Each command has its own class with all necessary properties. A list of all available commands is in section 4.1.4.

4.1.2 Arguments

Argument parsing is the next part. Parsing is done by using the library `CommandLine`. Available arguments for the scraper are **-f** or –**filename**, which specifies the path to the configuration file, **-h** or –**headless**, which turns on the headless mode, and **-m** or –**maximize**, which turns on the maximize mode (Works only with the Chrome driver). An additional argument is the –**help** which only prints a help message for the scraper, and then the program ends. To be able to access arguments in the whole program, they are stored in a static class called `Argument`. Arguments are made as private to minimize potential errors, and methods are implemented to retrieve their values.

4.1.3 Configuration file

The configuration files are written in the JSON[1] format. That is the founding block for this project since all actions that are going to control the website will be taken from these files. The configurations are built according to the JSON schema created for them. For the JSON schema, please read 4.1.5. For the configuration files, there are classes that are going to store the data deserialized from them. The main class is called `Config`, which includes a list of secondary class `CommandConfig`, which stores all required parameters of a command to be able to execute it correctly. That means a name of a command and a dictionary for arguments. To build a configuration file, the users can either do it manually or use the web application that provides an easier way.

4.1.4 Building a configuration file

This section will provide all the necessary information about available fields, their values, and their meaning. Fields are case-sensitive, so fields should be named as mentioned below when creating the configuration file. As mentioned before, these files can be created manually, but it is preferred to use the web application since it reduces the number of errors

[1] JSON - JavaScript Object Notation

that users can make. If the user still wishes to build it manually, it is preferred to use some IDE[2], which allows usage of a JSON schema for creating JSON files.

Required fields

This section covers all required fields for the configuration file. These are necessary for the scraper to work.

- **url** - This field specifies the URL of the website that will be scrapped. It is a string value in classic URL format. For example, https://google.com is a valid URL. It is necessary to have the prefix *https* or *http* because Selenium checks provided URL and will throw an exception if the prefix is missing.
- **loop** - The field "loo" works as a switch for the scraper, specifying if the scraper should loop or scrape once and end. This field takes bool values true and false.
- **waitTime** - Specifies the time in milliseconds the program should wait before starting another scraping loop. Allowed values are numbers beginning with 0. If the user doesn't want the scraper to loop, set it to 0.

Optional fields

These fields are only optional but it is recommended to fill them.

- **driver** - Option to choose which driver to use. Available values are **Chrome**, **Firefox** and **Safari**. If this field doesn't exist or it is set to null, the scraper will use the Firefox driver.
- **commands** - This is an array of commands in chronological order that the scraper will execute after loading the website.

Commands

The list below contains all of the implemented Selenium commands used to control the websites. Detailed description of all commands is in appendix B.

- AcceptAlert - Accepts an alert on a website.
- AddCookie - Adds cookie to a website.
- Back - Moves back to a previous page.
- Clear - Clears a form field.
- Click - Simulates a click action.
- DeleteAllCookie - Deletes all cookies.
- DismissAlert - Dismisses an alert on a website.
- ExecuteJavaScript - Executes a script.

[2]IDE - Integrated Development Environment

- Forward - Moves forward to a next page in history.
- ImplicitWait - Waits for all elements to load.
- Maximize - Maximizes browser window.
- MoveToElement - Simulates a mouse hover event (Browser can't be headless).
- Navigate - Navigates to a URL.
- SaveAttribute - Saves attribute value.
- SaveCssValue - Saves CSS value of an element.
- SaveHtml - Saves HTML source code.
- SaveTagName - Saves tag name of an element.
- SaveText - Saves text of an element.
- SaveTitle - Saves page title.
- SendKeys - Sends text into an element.
- SendKeysAlert - Sends text into an alert.
- SendReturn - Sends a return key.
- Submit - Submits a form.
- WaitUntilClickable - Waits until an element is clickable
- WaitUntilExists - Waits until an element exists.

How to use XPath to locate elements can be read in section 3.3.1

The example 4.1 provides a simple configuration file for saving the header's value on website `https://refactoring.guru/` after clicking on a menu item.

4.1.5 JSON schema

The JSON schema is used for validating the configuration files. Since it is created at run time, it is not possible to have an outdated JSON schema that would cause invalid validation.

The JSON schema is generated in **JsonGenerator.cs**. There the program creates dictionaries for each object that is present in the final schema. Dictionaries are filled from the most nested dictionary. Commands are generated from the assembly that takes all classes whose name contains *WebScraper.SeleniumCommands*. This happens in the method called `GenerateCommandConfigs`. This method also fills the commands with their properties by calling method `GenerateParameters`. The return value from these methods is the list of all command objects with their properties, and the list is ready to be added to the JSON schema. When all required data are included inside a single dictionary, this dictionary is serialized and saved into the file named `config.schema.json`.

```
{
  "url": "https://refactoring.guru/",
  "loop": false,
  "driver": "Firefox",
  "waitTime": 0,
  "commands": [
    {
      "Click": {
        "args":
        {
          "path": "//a[@href='/design-patterns']"
        }
      }
    },
    {
      "SaveText": {
        "args":
        {
          "path": "//h1",
          "name": "HeaderName"
        }
      }
    }
  ]
}
```

Listing 4.1: Example of a simple configuration file

4.1.6 Validation and serialization

Validation is implemented in the source file **JSON/JsonValidator.cs**. Validation first loads the JSON schema and the configuration file. The schema is parsed into the `JSchema` type, and the configuration file is parsed into the `JToken` type. After that, the method `IsValid` is called on the variable that holds the configuration file.

Serialization is done after successful validation. Firstly is called method `JsonConvert.DeserializeObject` on the configuration file. That fills all the fields in the `Config` class which stores all the information. But since the list of commands is stored in generic class `CommandConfig` it needs to be converted to the actual instances of the specified command. That is done by iterating through the command list. In every iteration, a class with a matching name of the property name in the class `CommandConfig` is taken from the assembly. That provides an actual class of the given command. This class is then instantiated. After that, each property from the class is filled with values from the dictionary contained in the `CommandConfig` class. The command is then ready and added to the list. The list into which the command is added serves as a guide for the scraper to know which command and in which order to execute.

Data are considered as prepared when all of this is done, and the scraping can begin.

4.1.7 Scraper service

The whole scraper runs as a console service. That enables better control of the application, the option to add singleton instances for the entire application, and provides a clean way to end the application so all resources can be freed without leaking.

The application is built by using the `Microsoft.Extensions.Hosting`. The app uses a builder that handles all services, like the database and the scraper service. After the builder builds the app, it runs as a console application.

The actual scraper service uses `IHostApplicationLifetime` to control the service's life. That means it ends the service when it successfully runs its course or if something goes wrong and it ends with an error. When the life of the service ends, the exit code is set accordingly.

Another part of the service is the logger. The logger logs all things that happen with the scraper so the user can check how the scraper progressed. Standard notices are written as debug traces and errors as error traces.

The flow of the service is as follows: Firstly, the file path for the configuration file is checked. Next is checking the project path to ensure it was run from the correct directory because the database could have trouble working as intended if it didn't. That is followed by handling everything for the configuration file - generating a JSON schema, validating the file, and its deserialization. All that is left to do is to run the actual scraper.

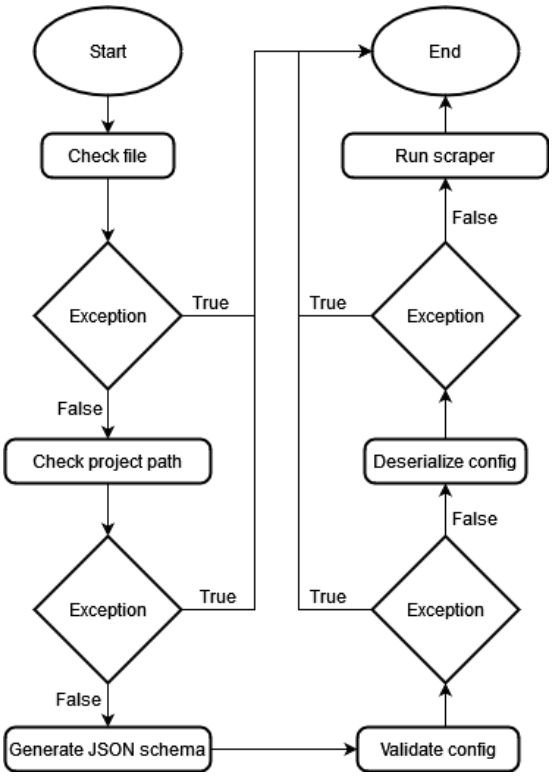

Figure 4.1: Flow of the scraper service

4.1.8 Scraper

Scraping is done in method `Run` inside static class `Scraper`. This method is run from the scraper service. To be able to run the scraper, there are some parameters needed. The first

two parameters are the instance of `Config` class that stores the data from a configuration file and the list of commands to be executed. These parameters serve as a guide for the scraper so it knows exactly what kind of website to scrape and with which commands. Another set of parameters is the facades for each database table, which means the website and the element facade. These facades are described closely in section 4.2.6. The last parameter that is needed is the logger. This logger handles all outputs to the terminal and serves as a way to let the user know what is happening. All these parameters are passed as arguments to the method.

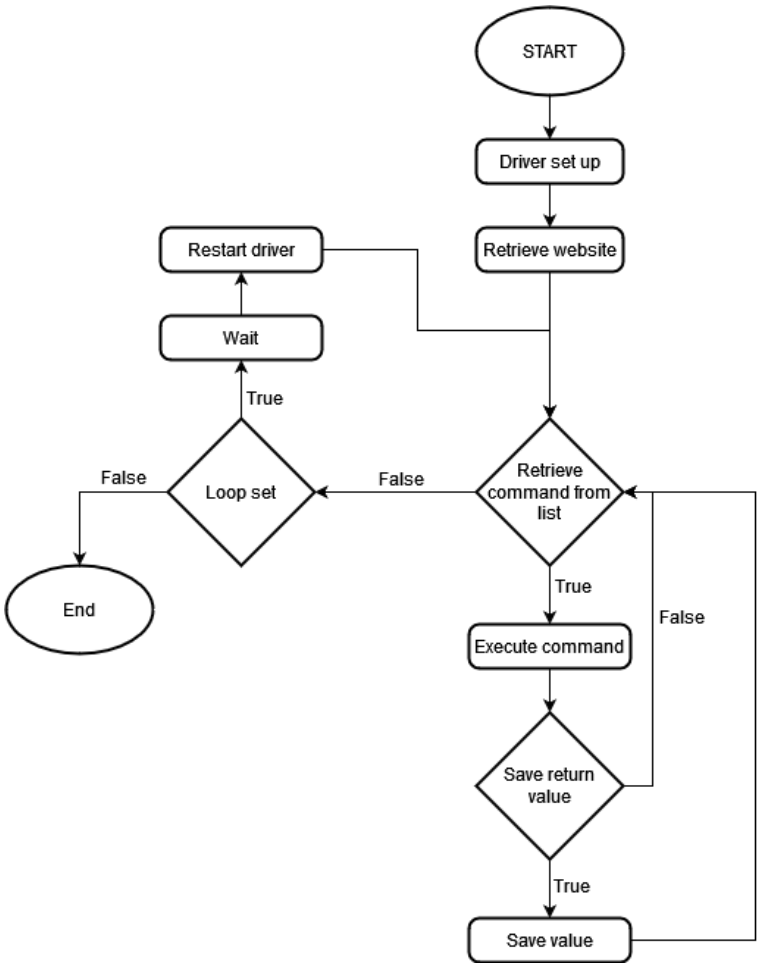

Figure 4.2: Flow of the scraper

The scraper first sets up the driver. That is done in method `DriverSetup`. That method creates an instance of the class `SeleniumWebDriver` and calls its setup method to create the driver correctly. If the setup doesn't succeed, the program ends with an error.

The next important thing is to get the website entity from the database. For this purpose serves the method `GetWebsite`. That method calls a method from the website facade that tries to retrieve the website entity from the database. If the entity doesn't exist, it creates one. Once that is done, the website entity is prepared and ready to be used in creating element entities that are used later.

After this, the driver is prepared, and the actual scraping can begin. Now the scraper iterates through the list of commands prepared earlier. Iteration through the list is inside

an `do while` loop because this enables the scraper to run at least once before ending. If the configuration has its loop setting set to true, the loop continues with another iteration, and the list of commands is iterated again until the entire program is terminated.

Iteration of the command list is done by `foreach` loop since it enables easy access to each instance of the command inside the list. The command is first executed by calling the method `ExecuteCommand`. Since the commands in the list are saved as instances and not the actual classes, it is needed to retrieve the `Execute` method, which executes the command, by using a `GetMethod` method on a retrieved instance that takes a name of a method as a parameter. Because all commands have the same method for executing called **Execute** there are no problems retrieving it. Using this way to retrieve the method presents another problem. The problem is that since the method cannot be called directly but only with method `Invoke` called on the variable that stores the `Execute` method. That means that the driver parameter needs to be wrapped into an object type since the `Invoke` method takes the parameters for the execution as an object. The prepared command is then executed. The execution can throw an exception if the command fails. If that happens, the program ends with an error. The result from the execution is then returned. If the command doesn't have any return value, null is returned.

The last part of every iteration is to check if the result of the command is supposed to be saved. That is done by checking the command name because all commands that should be saved have the substring "Save" in their name except for the command `ExecuteJavaScript`. Method `SaveCommand` handles the saving to the database. Once the method is called, a new element entity is created with all of the necessary information, and then the method from the element facade is invoked. After this, the result is successfully saved in the database.

When the whole list is completely processed, and the loop setting is set to true, the task is delayed according to a configuration file's value `waitTime`. After that, the driver is restarted, and the `do while` loop continues with the next iteration. If the loop setting is set to false, the scraper ends.

That concludes the functionality of the scraper.

4.1.9 Error codes

There are a few return codes that the scraper can have. If everything is finished successfully, the return code is 0. The program returns 1 if something goes wrong while executing the commands or setting up the driver. Errors contributed to the arguments and their parsing return 2. If the validation of the configuration file is unsuccessful, the returned value is 4. Problems with deserialization return a value of 5. The last unique value that can be returned is the value 3. That happens if the project is run from the wrong directory.

4.2 Database

The database is an important part of the scraper since it stores the data from the scraping and allows them to be viewed from the web application. This project uses the `Microsoft.EntityFrameworkCore`. The database uses the Repository and Unit of Work Patterns. That creates an abstraction layer between the data access layer and the business logic layer of the application. This database uses an SQLite local database because it has only two tables, and the SQLite is more than enough for this project. The two tables in the database are the Websites and the Elements tables. Each record in the Elements table has a foreign key `WebsiteId` that represents to which website the element belongs. A facade for each of

the two tables exists that handles the creation, modification, or deletion of records in the database by using repositories. There are also present entity mappers that map existing entities to new ones when updating records in the database. Each part of the database is added as a service to the WebSraper service as a singleton to ensure that the application doesn't create more than one database context.

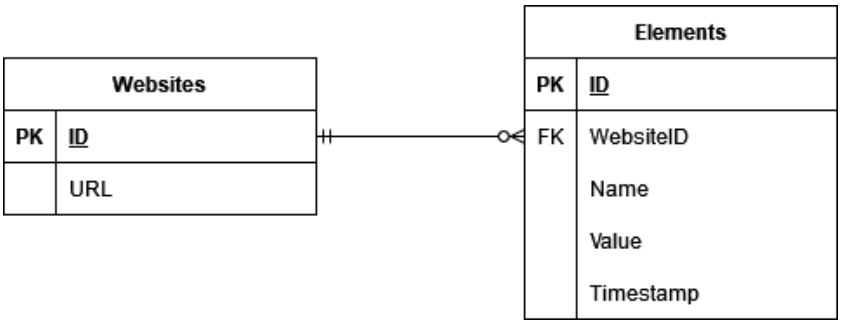

Figure 4.3: ER diagram

4.2.1 DbContext

This is the most essential building block for the database. It provides a way to communicate with the database, whether to retrieve data or create or update them. When setting up the database context, it is important to specify which entities will be used and their relations. In this project, it is the Element and the Website entity. Since the Website entity can have multiple Elements, it must be specified in the builder.

When the context is prepared, the factory for creating a new context is used. That enables more accessible and more secure access to the context.

4.2.2 Entities

There are two entities present in the database. Both entities use the same interface `IEntity`, which has only the ID since it is the only constant in both entities.

The first one is the Website entity. This entity contains only the URL of the website and a unique ID. This entity serves as a way to bind the Element entities for this website. That way, when users want to delete all scraped elements from a specific website, they only need to delete the Website entity. All of the Element entities will also be deleted thanks to the cascade on delete option.

The other entity is the Element entity. The entity contains a unique ID, a name that serves as an identifier for the element, so the users don't need to use ID for identification, a value that contains the results from the scraping, the timestamp at which the entity was created, and the website ID to which this entity belongs to. This entity also contains a whole Website entity to which it belongs. But the Website entity is used only in the web application to make it easier to show the website URL instead of ID because ID isn't helpful to the user.

4.2.3 Unit of Work

Unit of Work encapsulated every object affected by a business transaction. That ensures every change to the database objects will be registered and committed.

This project has class `UnitOfWork` that inherits the interface `IUnitOfWork`. This class handles the commits to the database or disposes of the changes. It also contains a method for creating a repository. More about repositories is in the section 4.2.4.

For better creation of a `UnitOfWork` class, there is a `UnitOfWorkFactory`. When creating this class, it is necessary to use a database context for correct creation.

4.2.4 Repository

The repository is a point between the database context and the facade. It handles all the actions with the database. Methods that can be used from this repository are the `ExistsAsync` which checks if an entity exists in the database, `InsertAsync`, which inserts a new entity to the database, `DeleteAsync`, which deletes an entity from the database, and `UpdateAsync` which updates an entity in the database. The method `UpdateAsync` uses mappers that maps a new updated entity to an old one.

The repository needs to be created for a specific entity because it executes its methods with a certain type of entity. That means that a repository created for an element entity won't work correctly or even not at all on a website entity.

4.2.5 Mappers

Mappers are used when updating already existing entities in a database. A mapper for each entity takes the old entity and sets all properties to the new values.

4.2.6 Facades

Facades are the layer that handles the business logic before calling the repository to interact with the database. This layer is the part called from the scraper application when retrieving or inserting data into the database. Since the database only contains two tables, two facades are present. Both of those inherit the core interface `IFacade`. This interface includes virtual methods for deleting and inserting an entity into the database.

Deleting is done by calling method `Delete` from a repository. The insertion is a bit more complicated because there are two ways to insert an entity. Either update the already existing one or insert a completely new entity. So to choose which way to use, the method `SaveAsync` first checks if an entity already exists in the database and updates it if it does. If it does not, the method inserts a new one.

For the Element entity, there is the `ElementFacade`. This facade inherits the interface `IElementFacade`. This facade has one additional method to the generic facade: the `GetAsync` method. This method returns all elements stored on the website. The results are used in the web application so the user can view the database.

Another facade is for the Website entity. Similar to the `ElementFacade`, it inherits an interface called `IWebsiteFacade`. The facade implements three new methods that are not present in the base facade. Those methods are `SaveWebsiteAsync` which works similarly to the insertion method from the base facade but with the difference that a Website entity is either retrieved from the database or a new one is created since a website doesn't ever need to be updated. The next new method is the `GetAll` method which works the same as the `GetAsync` method from the `ElementFacade` but with websites. The last method is the `GetWebsiteAsync`, which only retrieves a website from the database.

4.3 Web application

The web application serves as a way for the user to either build a new configuration file, update an already existing one, view data from the database with the option to delete some records and a way to run configuration files with a disabled loop option.

The application is written using the Blazor framework. That allows users to call the scraper from the website without any extra problems. There are two choices when building a web application using Blazor. The first is the Blazor WebAssembly. That means the web application is built on the client side. The other way is the Blazor Server which builds the web application on the server, and the client can access it. Since the project needs to access the database and run the scraper, using the Blazor Server choice made sense. That way, there are no problems with the database and other things on the server side.

In addition to the basic Blazor components, there is the `MudBlazor` library. The library contains a lot of valuable components when building a website. For example, better text fields, drag-and-drop fields, and others.

Blazor even allows the usage of JavaScript, so when some functionality not present in the Blazor is needed, it can be added as a JavaScript script and called from the page. To do this, the script needs to be included in file **_Host.cshtml**. After that, all that is needed is to inject IJSRuntime into the web page and invoke the script. This is used in this project for saving files on the client side.

There are four pages present in the web application. The main page is the WebScraper page 4.3.1. The next pages are the Elements 4.3.2 and the Websites 4.3.3 pages. The last page is the Configuration file builder page 4.3.4. These pages create a united web application for controlling the scraper, creating configuration files, and viewing the scraper's results.

All of these pages have the same fixed navigation bar that handles navigation between the pages.

The application uses the same procedure when adding the database into the service as the scraper service (By adding all required things for the database as a singleton service). That enables injecting the facades into the pages so they can access the database.

4.3.1 WebScraper page

This is the index page of the web application. It serves as a way to run the scraper instead of running it from a command line. There are also components that allow uploading configuration files to the server; choose which configuration file should be run and with which options. The page also has a written status of the state of the scraper.

The upload is handled by the button on the right side of the page. It reads a specified file's content and saves it onto the server inside a **WebScraper/Configs** folder. If the upload is successful, an alert that shows success is shown. If it wasn't successful, an alert that shows what went wrong is shown.

The next part of the website is the selection box. This box contains a list of all configuration files present on the server. The files are taken from the **WebScraper/Configs** folder. The scraper then runs the selected configuration.

Under the selection box, there are two check boxes. These represent the options with which the scraper can be run. The first is the headless option, and the second is the maximize option.

The last thing on the website is the run button with status text next to it. When this button is clicked, the page checks if the provided configuration file has the loop setting set

to false. If not, an alert explains to the user that loop configurations must be run from a command line. If the check returns success, the scraper is called as an asynchronous task. The status text changes according to the state of the scraper.

4.3.2 Elements page

The elements page contains one big table for viewing all the records from the database's Elements table. This table is made with `MudTable` from the `MudBlazor` library. That provides numerous possible options on how to make the user experience better.

The table contains four columns, each for one property of the element entity, except for the ID, since it doesn't have any value for the user. Also, the website ID property is translated into the URL so that the user can better understand to which website the element belongs.

One of the options for the user to better enhance his experience is the table's pages. Because the table will most likely have a lot of records, to list them all without means to reduce that isn't great. So the user can choose how many records are shown at once. If there are more elements than are shown, the user can switch to a different page number of the table.

The next option is the filtering option. This enables filtering of the records with any of the presented columns. For example, if the user wants to look at only records from a specific website, he must type its URL in the search column and press enter. Not only is it better for checking the data, but also for deleting them, which is described later.

There is also a way to select any number of records with check boxes next to them. That tells the page which records are considered for deletion.

The last part of the page is the Delete button. The button deletes selected records from the database. A dialog appears for the user if the button is clicked and at least one record is chosen. That dialog alerts the user that he is deleting a record and asks if he wants to proceed. If the user selects multiple records, the dialog is shown for each of those records unless the user chooses the dialog option "For all".

The records are taken from the database by using the `ElementsFacade` that is injected into this page.

4.3.3 Websites page

This page is almost a copy of the Elements page. The difference is that there is only one column, the website URL, because, as previously mentioned, the user doesn't benefit from having an ID that won't do anything but confuse the user. Also, this page injects the `WebsiteFacade` instead of `ElementsFacade` for apparent reasons.

4.3.4 Configuration file builder page

This is a more complicated page. It handles the creation of the configuration file. There is also an option to upload an already existing configuration file and edit it. This page also enables the saving of the file to the server or the client.

The whole thing is made as one big form. All of the form fields are identical to those available in the configuration files, with one additional form field specifying the configuration file's name. That field is optional; if empty, the configuration file will be named "config.json".

The entire form can be divided into two parts. The first part is the fields that will always be present no matter what. Those are the `url, loop, waitTime,` and `driver` fields. Each

of these fields has a different input style because every one of those fields needs different input. The `url` field is a simple text field, the `loop` field is made as a check box, `waitTime` is a number field, and the `driver` field is a selection box with three values. The values are hard-coded because they will always be the same. The values are Firefox, Chrome, and Safari.

These fields are present whenever the user tries to create a configuration because they are considered core fields.

The second part is the commands zone. This zone handles all commands. In the beginning, this zone is empty. A button with the text "Add command" handles adding a new command to that zone. The zone is made as a `MudDropContainer`. That way, the user can rearrange the order of the commands easily by drag-and-drop method. When a new command is added, it is rendered as an empty command. This command has a selection box from which the user can choose the name of a command. All of the commands' names are once again taken from the assembly of the scraper application.

After a command name is chosen, the web page generates additional fields for each property of the selected command. When those fields are generated, the entire command is ready. If the users no longer want a command in the container, they can delete it by clicking the button with an icon of a trash can.

Each command in the container is made in a specific `MudPaper` that helps the users see exactly where a command's area begins and ends. Knowing that, it is easier to drag a command into a different place to remake the order of the commands.

The commands in the drop zone are stored in a list. Each member of that list contains a command's name, command order, and selector for choosing a drop zone, but since there is only one, it is always set to that zone. The last property is the class for properties of each command.

Because the order of the commands can be rearranged, with every action within the zone, the command list is remade with the new order so the state of the list is saved, and there are no problems when the container is refreshed.

When the user is satisfied with the configuration file, he can choose to save it. There are two possible choices when saving. The first choice is to save it to the server. That choice is made by having a checked check box next to the save button. The second choice is to save the configuration on the client side, and this choice uses JavaScript to enable saving the file. No matter which way is used, the configuration is first validated.

Validation consists of validating the URL field and each command property field. If the validation is successful, the configuration file is built and saved.

The last thing remaining on this page is the upload functionality. That is done by simply reading the contents of the chosen file to be uploaded. Before anything more is done, the contents are validated against the JSON schema to ensure that it's valid. After that, the contents are loaded into necessary variables. Upload is done by clicking the upload button next to the save button.

4.4 Setup

This section covers all the necessary information on setting up the whole project to ensure it runs. It is essential to have .NET 7.0 installed. Using it on Windows OS is recommended, but using Linux and macOS is also possible. Next is important to have installed all required NuGet packages. They don't have to be installed manually because they are included as package references and will be installed when building the application.

Two commands need to be run to create a database. The first is to create an initial migration, and the second is to create the actual database. Commands below are called from directory **WebScraper.Database**. It is possible to call them from a different directory, but then it must have `-project` argument specifying a path to the database project.

```
dotnet ef migrations add InitialCreate --context WebScraperDbContext
dotnet ef database update --context WebScraperDbContext
```

Deletion of an already existing database can be done by the command below.

```
dotnet ef database drop --context WebScraperDbContext
```

The scraper can be run from the command line when the database is created. Commands below expects that the user is in the **WebScraper** project directory.

```
dotnet run -- -f path/to/configuration
dotnet run -- -f path/to/configuration -h -m
```

The web application can be run by calling `dotnet run` in the **WebScraper.WebApp** project directory

Chapter 5

Testing

Testing is vital to this thesis since many components must work together for the scraper to work correctly. This means that the Web application needs to communicate with the database and scraper itself, wrappers around selenium commands need to provide correct results, and the scraper has to store the results in the database without fault.

Testing is divided into simpler parts to test all of these requirements so each part can be evaluated individually. For selenium commands are created unit tests that assess the correct acting of those commands. Scraper is tested by running a number of configuration files ranging from simple to more complicated ones. These configuration files include testing of Chrome and Firefox driver's behavior on different websites. Lastly, web application include testing of configuration file builder, viewing database entities, and running scraper with a configuration that doesn't have loop parameter set to true. Websites to be tested are in list below.

- https://www.selenium.dev/
- https://refactoring.guru/
- https://testpages.herokuapp.com/styled/alerts/alert-test.html
- https://www.alza.cz/
- https://www.vut.cz/

Each configuration file contains a different sequence of commands to be used. All configurations can be found in folder **WebScraper/Configs**.

Testing was done on Windows 8.1 operating system.

5.1 Unit tests

As mentioned above, unit tests cover testing of wrappers of selenium commands to ensure their correct behavior. Since those wrappers are the core functionality of the scraper, they need to work precisely as selenium commands so the scraper can use them to control dynamic websites.

Each unit test cover only one command, but not every command can be easily tested. For example commands that can't be easily tested are wait commands which wait for some element to load and nothing else. Other commands have a simple testing method that

consists of creating a command wrapper, setting up needed parameters, and executing this command wrapper. After that, the assertion is used to validate a successful test. For example, `Click` command compares the new URL of the driver with the expected URL.

5.1.1 Save commands

Save commands all have a return value that is going to be saved in the database. That means testing these is pretty straightforward since all that needs to be done is to compare value from the wrapper and selenium command.

5.1.2 Navigation commands

This group of commands handles the navigation of a website. That means commands like `Click, Forward, Back` and others. Testing of this group is done by comparing the URL of the tested website after executing those commands and checking if the resulting URL is the same as the desired URL.

5.1.3 Alert commands

Handling alerts in selenium differs from handling other elements because alerts cannot be accessed as standard elements, but the driver needs to switch focus to the alert. Since alerts can be either accepted or dismissed, it can be challenging to check which action actually happened. So to properly test it, the tested website needs to have straightforward results of alerts.

Simple website `https://testpages.herokuapp.com/styled/alerts/alert-test.html` provides an easy way of creating alerts and checking their result.

5.1.4 Other commands

The remaining commands cannot be sorted into groups since each requires different testing method.

Commands like `AddCookie` and `DeleteAllCookies` checked the website's cookies and provided results accordingly. `ExecuteJavaScript` was tested by simply calling return for the page title and comparing it.

5.2 Web application

The web application consists of four pages. The first page is for executing the scraper and testing the correct functionality. All that needs to be done is to choose the configuration file with provided selector, check available options for the driver, and click on the run. To ensure everything works smoothly, a combination of all options must be tested. Since maximized option works only with the chrome driver, it will serve as the main driver for testing this page.

The first test was running a valid configuration file in headless mode and maximized. The maximized option is optional since there is a command to maximize the driver, but it can save some time. This test will also make sure all options for drivers work.

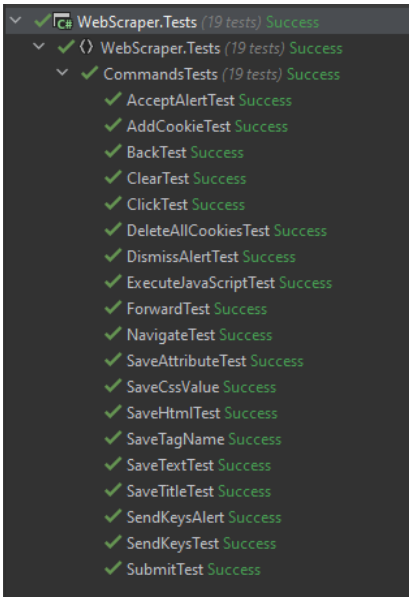

Figure 5.1: Unit tests

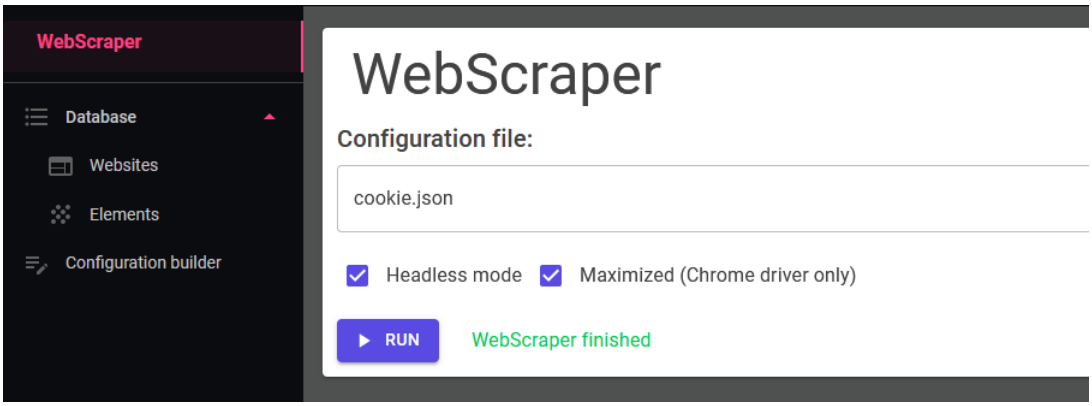

Figure 5.2: WebScraper page - successful run

The second test was running a configuration file that would produce some error.

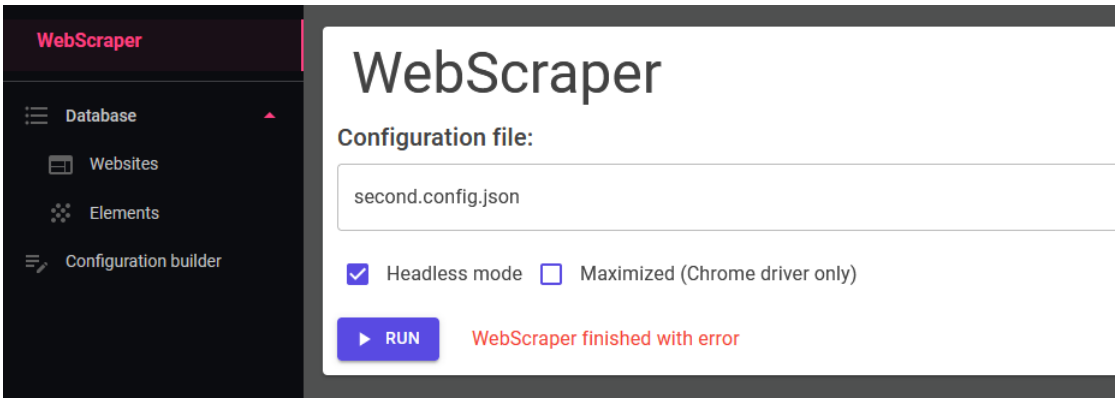

Figure 5.3: WebScraper page - unsuccessful run

The last test was to run a configuration file with a loop set to true because that configuration needs to be run from the command line. After **Run** button is clicked, a dialog warning the user about this was supposed to be shown.

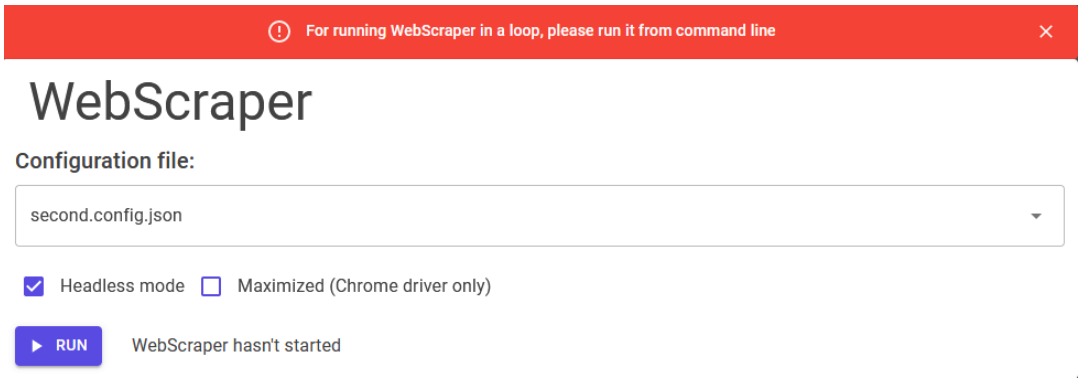

Figure 5.4: WebScraper page - Loop configuration file

The next pages that were tested are **Elements** and **Websites.** There are not many things to test since these pages are solely for viewing the database, and the only thing user can do is delete some data. Users can also filter records so that they can be tested too. Below can be seen the initial state of page **Elements** before testing.

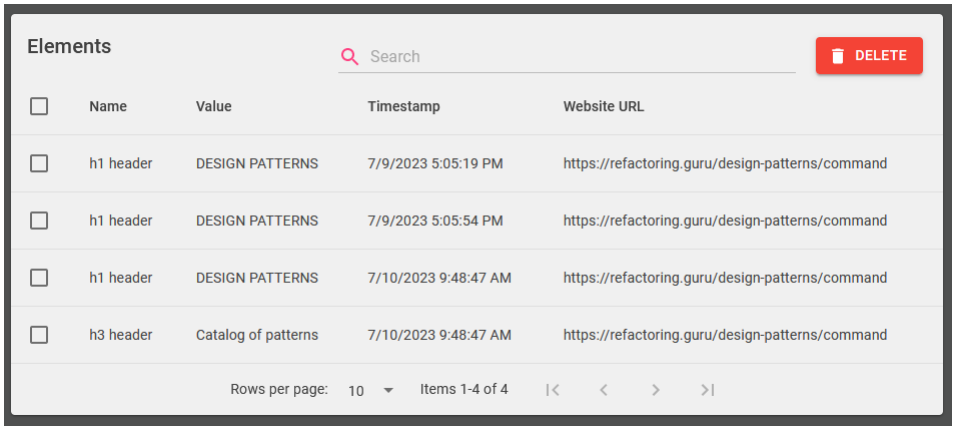

Figure 5.5: Elements page - Initial Elements page

After filtering data to show only records with a timestamp of 5:05, there were only two records. When clicking on both records and clicking on **Delete** button, a dialog was shown. After accepting this dialog, records were deleted, and only records that didn't match the timestamp of 5:05 are present, as shown below.

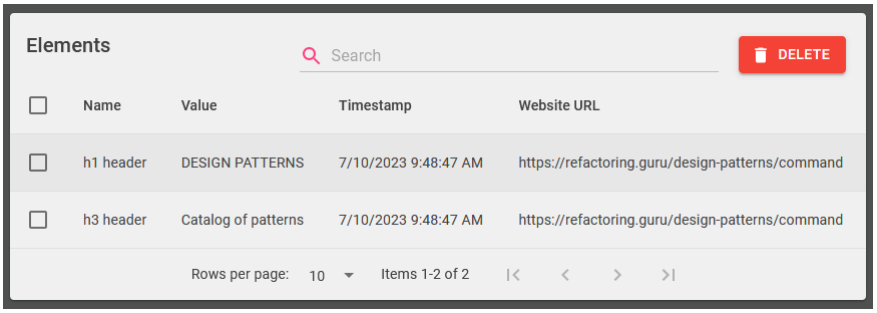

Figure 5.6: Elements page - Final Elements page

The same procedure was done with **Websites** page with the difference that when the user deletes the website record, all child elements will be deleted too. So for this test, the website with URL in figures 5.5 and 5.6 was deleted, and **Elements** page should be empty.

When the user clicks on the **Delete** button, a dialog should be shown warning them about deleting all child elements. After accepting, **Elements** page was empty, so this test passed.

That concludes the testing of **Elements** and **Websites** page.

Last page to be tested was the page **Configuration builder**. This page has a lot of possible choices on how to create a configuration file.

Firstly users can choose to name files to their liking. If the user doesn't choose, it will be named simply *config.json* so this can be tested simultaneously with other tests.

Next are the main parts of the configuration, meaning URL, loop, wait time, and selected driver. URL is required, so when users try to create a file without it, the page should let them know. Other fields are not required, so fill in their values and save the file to test if they are working correctly. Testing empty URL passed since the error was shown 5.7.

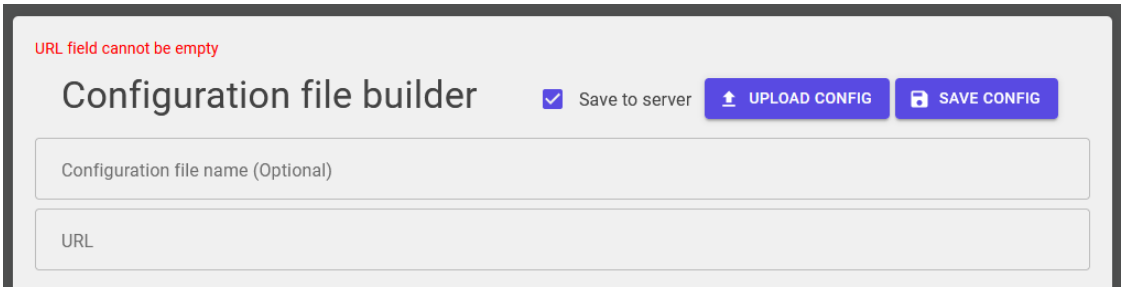

Figure 5.7: Configuration builder page - Empty URL

A significant part is the command part. To begin creating commands, button **Add command** is clicked, and a first command should show. When the command name is selected from the selection box, command properties should be generated correctly. All parameters must be filled in, so if users don't type in values, the same thing as with an empty URL should happen. Commands also have a delete button that deletes the command. When multiple commands are present, users should be able to drag them to different positions. To test these requirements, three commands were created, and their values were selected. This test doesn't include testing of empty parameters since it will be included in later tests. In figure 5.8 is starting state of commands.

Figure 5.8: Configuration builder page - Starting state of commands

Testing steps to include all of the functionality mentioned above were as follows: `Back` command was moved to the top, and `Clear` command was deleted. The final state after the test is in figure 5.9.

Figure 5.9: Configuration builder page - End state of commands

Users can edit already existing configuration files using the **Upload config** button. For testing purposes will be used one simple file.

Firstly, the driver is set to Chrome and loop to false in the simple file with one command `AddCookie`.

Figure 5.10: Configuration builder page - Upload result

In 5.10, we can see that the configuration contains one command `AddCookie`, the driver is set to Chrome and loop to false, so this test passed.

When configuration is made, it is needed to be saved. There are two choices on how to save the file. The first one is when the checkbox with label **Save to server** is checked. This will ensure that the configuration file is stored on the server in folder `WebScraper/Configs` and can be easily run from **WebScraper** page. Another choice is to save the file on the client side, and this will happen when the checkbox is not checked. The file is then downloaded according to the user's browser settings.

After the button to save is clicked, checks are made to make sure everything is in order for the configuration to be valid. That means a valid URL, all parameters are set, and correct values of those parameters. For example, if the parameter is `Time`, its value must be a valid unsigned integer. When the parameter is `Path` or `Attribute`, its value is checked if it's valid XPath. If any of those checks fail, the file is not saved, and an error message is displayed. To test this occurrence, command `Click` had an invalid XPath when clicking the save button. The result of this test is shown in 5.11.

This procedure was repeated on every available command and its parameters to ensure all were checked correctly.

If the checks succeed, the file is saved. Saving to the server shows an alert depending on the success of the saving. To test the saving of files, a simple file that was already used for earlier tests was enough. Both ways of saving should provide the same configuration files.

As can be seen in 5.12, the file was saved to the server successfully and can be compared to the configuration that was uploaded first.

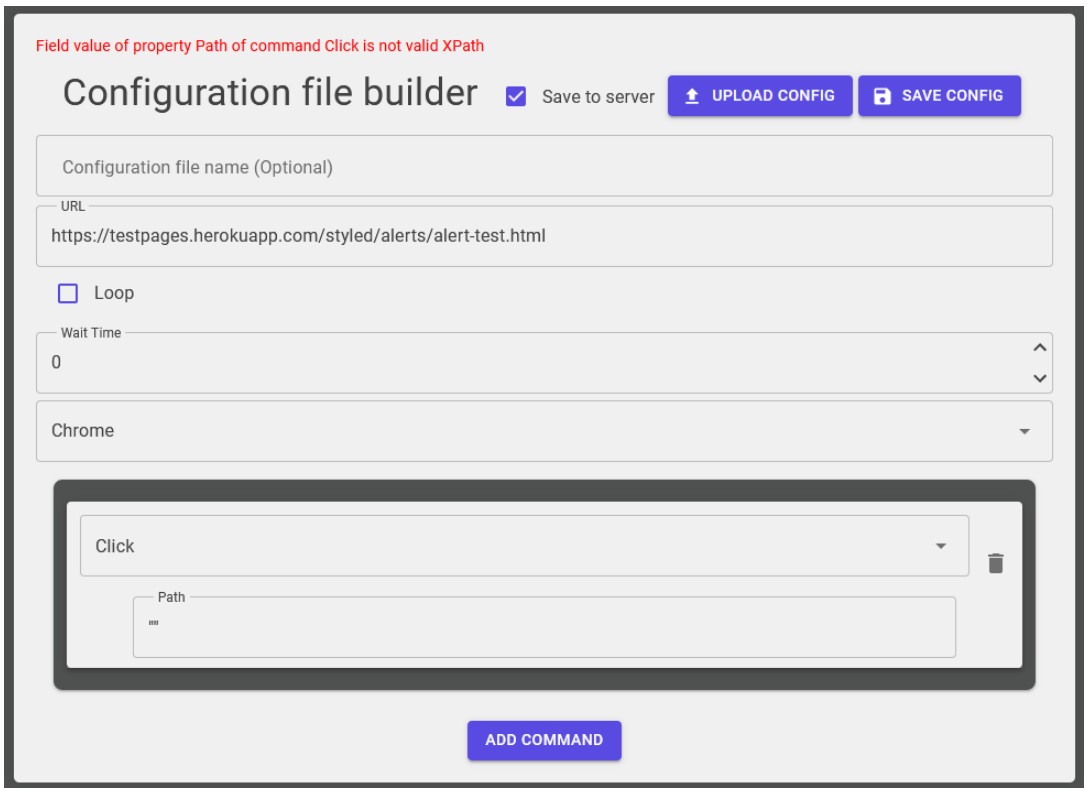

Figure 5.11: Configuration builder page - Not valid config

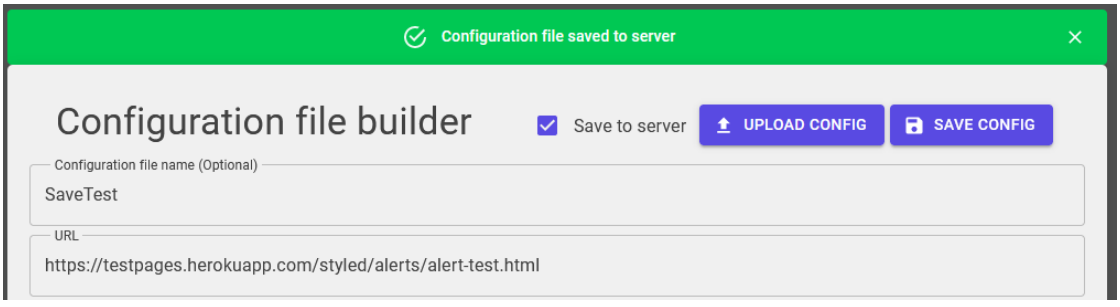

Figure 5.12: Configuration builder page - Successfully saved

```
{
  "url": "https://testpages.herokuapp.com/styled/alerts/alert-test.html",
  "loop": false,
  "driver": "Chrome",
  "waitTime": 0,
  "commands": [
    {
      "name": "AddCookie",
      "args": {
        "key": "Key",
        "value": "Value"
      }
    }
  ]
}
```

Listing 5.1: Saved config

When comparing this configuration with the uploaded one, we can see they are the same, so this test passed.

That concludes the testing of the Web application for WebScraper. The functionality works as intended.

5.3 Simple configuration files

This section will test the scraper with configurations that contain a few commands to determine if the scraper works correctly as a whole project. Configuration files for this part will be created using the web application's configuration builder, and they will also be run from the application. This part of testing will determine if the whole project communicates correctly together. To ensure correct behavior, configurations will be run with both Chrome and Firefox drivers and on numerous websites.

The first website that was tested is the website `https://testpages.herokuapp.com/styled/alerts/alert-test.html`, which served for alert testing. This configuration included commands `Click`, `AccceptAlert`, `DismissAlert` and `SaveText`. Firstly button was clicked that showed the alert, and then the alert was accepted. After that, the website showed text with true or false depending if the alert was accepted or not. The first iteration was to accept the alert and save the text. The second iteration was made by declining the alert and saving the text value. Both iterations were in the same configuration. Tested were both Firefox and Chrome drivers. Results from the test can be seen in 5.13. Records with earlier timestamps are from the Chrome driver. Others are from the Firefox driver.

The next testing website was `https://www.vut.cz/`. In this configuration were tested commands `Click`, `SendKeys`, `Submit`, `WaitUntilExists`, `SaveTitle`, and `SaveText`. In the beginning, by clicking on a specific element on the website, a search bar was shown. After that, text *IZP* was sent into the search bar and submitted. The next step was to click an element that contained specific text. The last steps were to save the web page title and the header's text. The Chrome driver finished successfully, but the Firefox driver threw an exception when trying to execute the last `Click` command. This was due to the driver

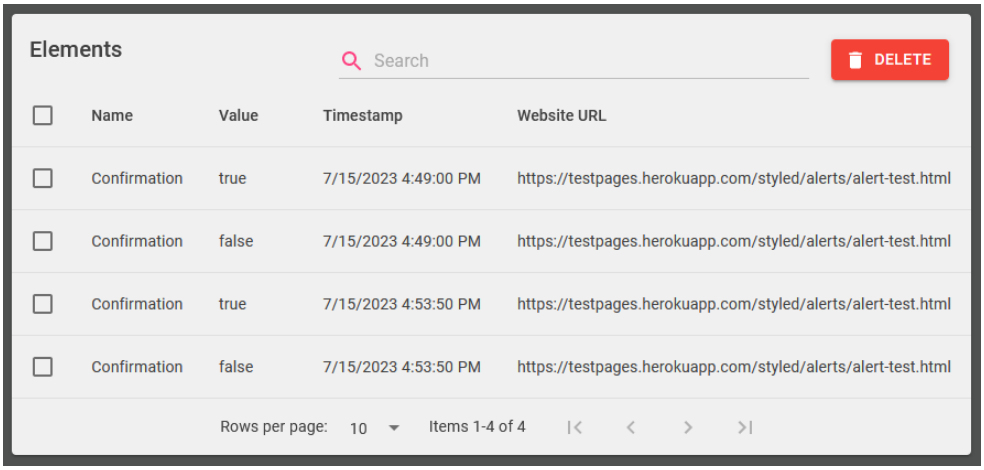

Figure 5.13: WebScraper - Configuration file for alerts

executing the command before the expected element existed. This problem was solved by adding command `WaitUntilExists` before `Click` command. After fixing that, both drivers worked as intended. Test results are in figure 5.14.

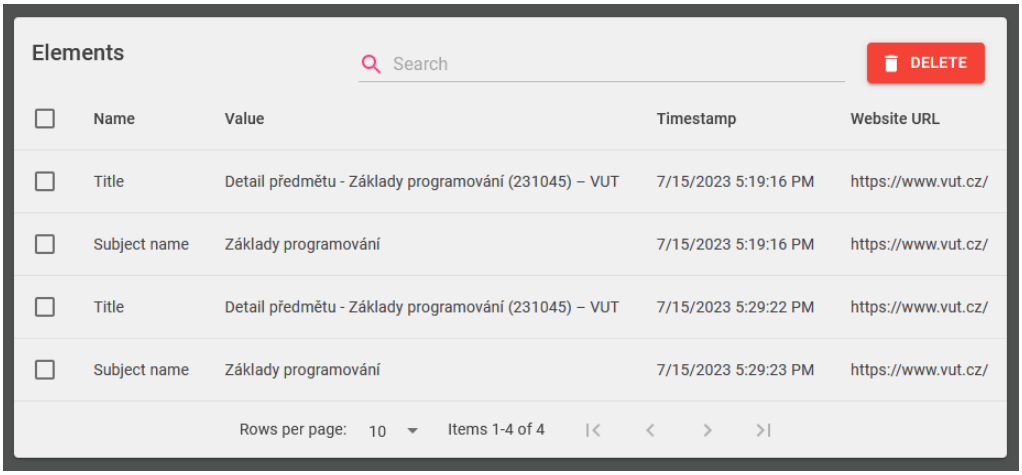

Figure 5.14: WebScraper - Configuration file for VUT.cz

Another testing website was `https://www.alza.cz/` because this website is quite cluttered with popup windows and cookie confirmations. It also has a lot of dynamic components that can be tested. Commands tested on this website were `Maximize`, `WaitUntilClickable`, `WaitUntilExists`, `Click`, `Navigate`, `Back`, `Forward`, and `SaveAttribute`. Since this configuration tested command `Maximize`, the scraper was run without the headless option, so it was obvious whether the driver maximized. The first thing that the website does is to ask for cookie confirmation. Because it isn't present immediately, `WaitUntilClickable` command has been used to wait for the confirmation to appear. After a few clicks on the website, the `Navigate` command followed by `Back` and `Forward` commands was used to test the navigation of the driver. Lastly, command `SaveAttribute` was used to save attribute *href* from an element that redirected the user to another page. As mentioned above, the problem with the Firefox driver occurred again when the driver

wanted to save the attribute before it was present, so the command `WaitUntilExists` was used. The Chrome driver passed successfully, and the Firefox driver also passed after adding the wait command. Results are shown in figure 5.15.

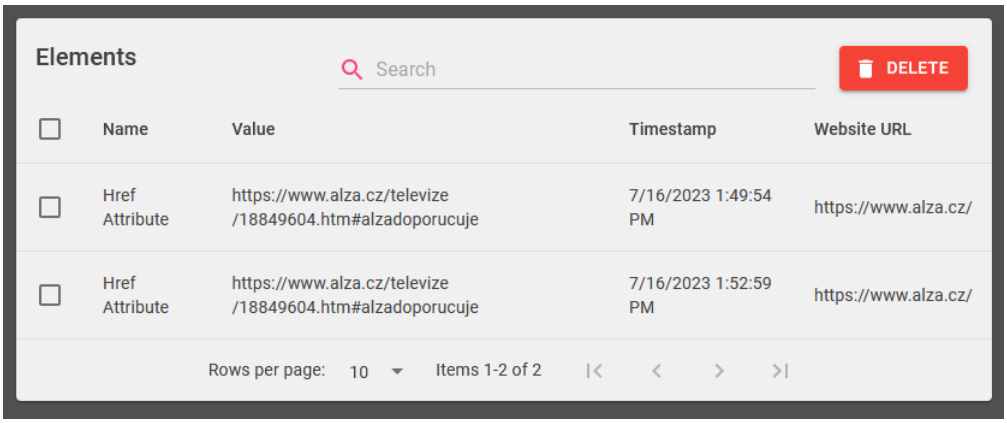

Figure 5.15: WebScraper - Configuration file for alza.cz

The following test was done on website https://refactoring.guru/. This test was mainly done for testing remaining save commands. To ensure that at least some navigation commands method correctly, one `Click` command was used to navigate to different pages. After that were consecutively called commands `SaveCssValue`, `SaveTagName`, and `ExecuteJavaScript`. The first command to save CSS value requires CSS property to be saved. For this purpose was chosen simply the color of the text. After the execution of this command can be seen that the Firefox driver produces slightly different results than the Chrome driver since each driver can have a different way of interpreting the CSS. The Firefox driver saved simple RGB values, but the Chrome driver saved RGBA values. The rest of the commands provided the same results. All results can be viewed in figure 5.16.

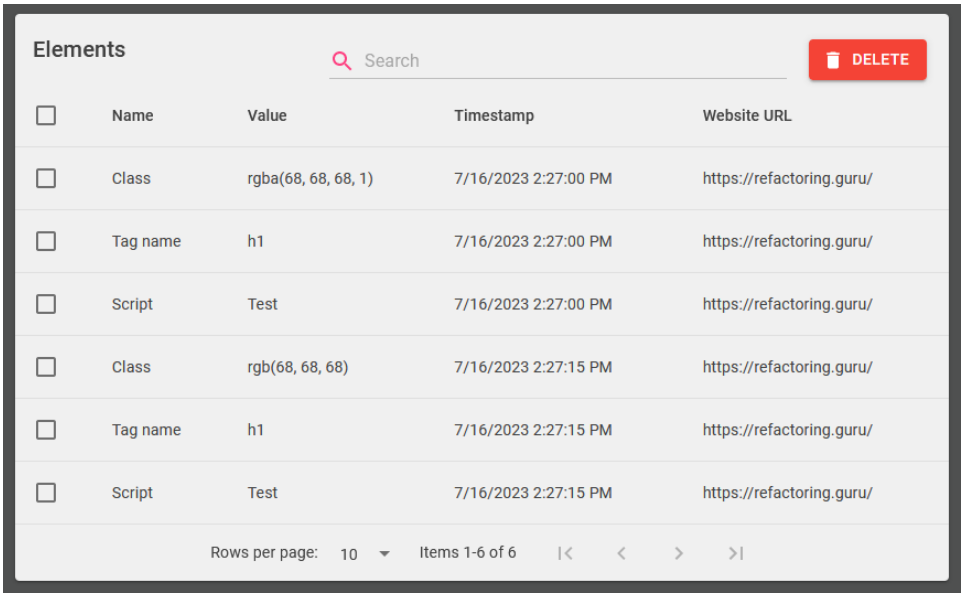

Figure 5.16: WebScraper - Configuration file for Refactoring guru

The last simple configuration file testing website was https://www.selenium.dev/. This page was tested sorely for the purpose of testing different websites to make sure that the scraper works on various websites other than those that were already tested. Once again were tested the navigation and save commands. This test saved the header of the documentation page, the title of the documentation page, and the header of the main page. The result of this configuration should be three records in the database for each driver with values "*The Selenium Browser Automation Project*" for the first header, "*The Selenium Browser Automation Project | Selenium*" for the title, and "*Selenium automates browsers. That's it!*" for the second header. Results in database 5.17 were as expected, so this test also passed.

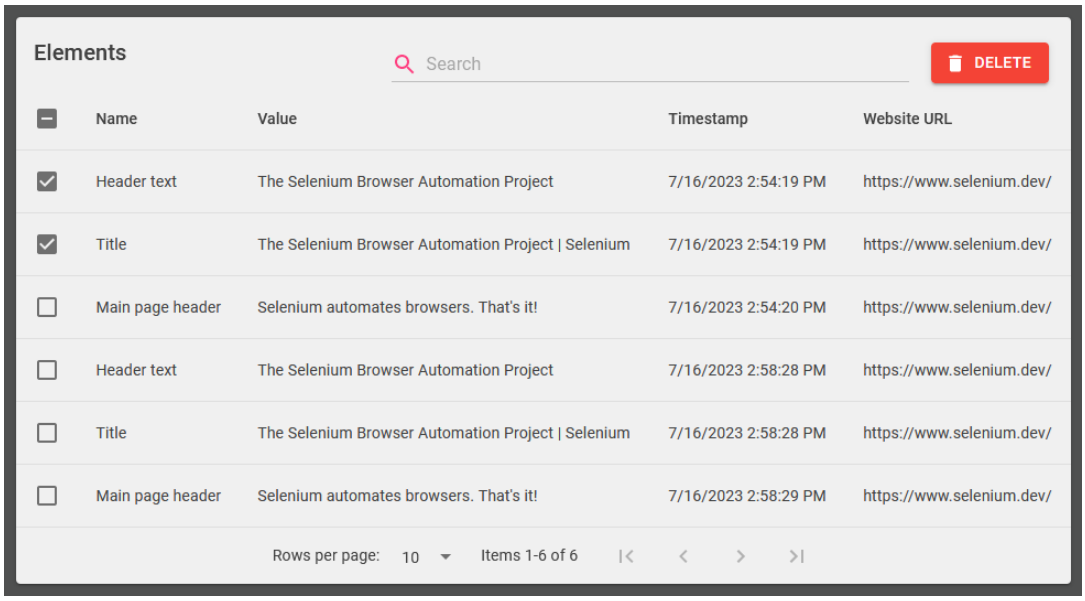

Figure 5.17: WebScraper - Configuration file for Selenium website

Since all of these configurations were only called once, and their loop setting was set to false, it is better to also test the loop option with them. As mentioned before, looping of configuration must be run from the command line. For this test, configuration for the website https://www.vut.cz/ was used. For the purpose of testing, the loop ran three times. The final results can be compared to the previous test 5.14 with the difference that the results now should be present three times 5.18.

These tests conclude the testing of simple configuration files. All tests were finished successfully. From these tests, the difference between the Firefox and the Chrome driver can be seen, where the Firefox driver needs more commands that wait for certain elements to appear. Another conclusion from these tests is that the Chrome driver is faster. Tested were both headless and normal version of the drivers. All configuration files can be found in the folder **WebScraper/Configs**, where file names contain the word simple.

5.4 Complex configuration files

The next phase for testing the scraper is running more complicated configuration files. This means that each configuration will have at least twenty commands, including a combination

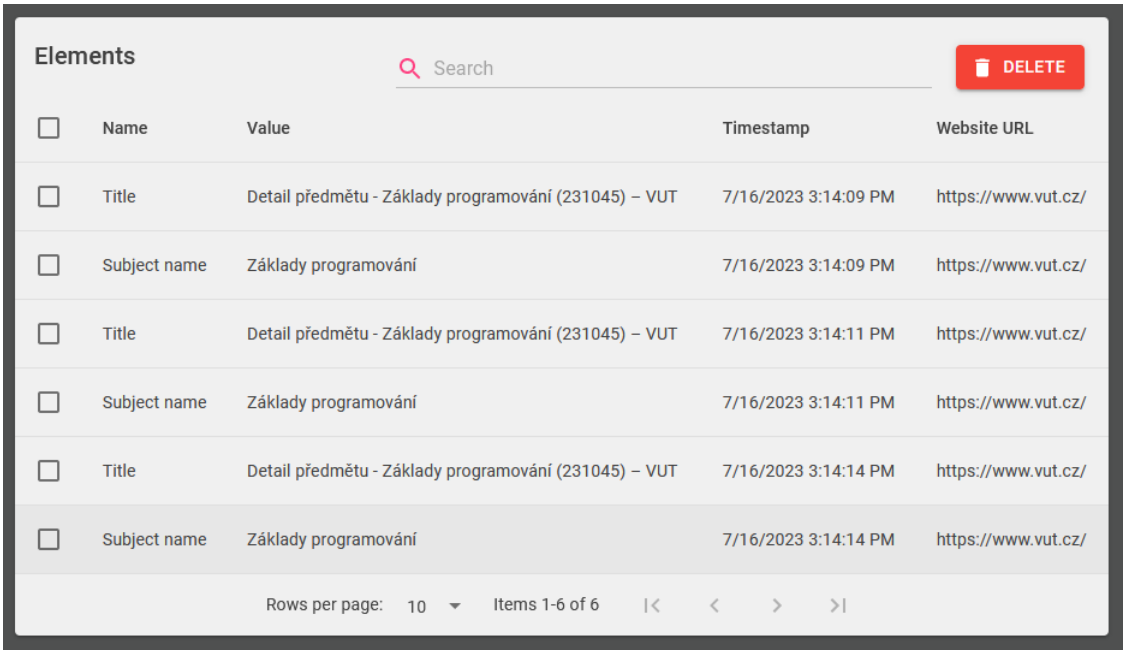

Figure 5.18: WebScraper - Configuration file for VUT website with loop

of different commands for navigating the website and saving its elements. Once again, all configurations will be run with the Firefox and the Chrome drivers.

The first tested website was `https://www.vut.cz/`. This configuration extended the one used for simple the configuration testing. This test used the command `MoveToElement`, which works only if the driver is not in headless mode, so this test didn't include headless option testing. Other commands included in this test were `Click`, `SendKeys`, `Submit`, `Maximize`, `WaitUntilExists`, `SaveText`, `Navigate`, `Back`, `Forward`, `DeleteAllCookies`, `Refresh`, and `SaveCssValue`. The exact flow of these commands can be seen in file **WebScraper/Configs/vut.config.json**. The scraper finished without error, and the return values of the save commands were as expected. Both drivers produced the same results except for the CSS value, but the reason for this was already explained above. This test proved that the scraper can handle bigger configuration files with almost every command and still provide correct results and finish successfully. Results are in figure 5.19.

For the next test was used the website `https://www.selenium.dev/`. This test aimed to check if the scraper can handle bigger configurations on different websites than the one tested before. Commands included in this test were `Click`, `Maximize`, `Back`, `Forward`, `Navigate`, `ImplicitWait`, `Refresh`, `WaitUntilClickable`, `WaitUntilExists`, `SaveText`, `AddCookie`, `DeleteAllCookies`, and `SaveTitle`. Since this configuration is quite long, all steps in sequence can be seen in file **WebScraper/Configs/selenium.config.json**. Both drivers passed this test without any problems, and their results also match 5.20. These results are as they were expected when creating this test configuration.

Like simple configuration tests, these must also be tested in a loop setting. Since the configurations are more complicated, testing them both and with both drivers was better. The results provided by these tests were the same as the result from the tests without the loop. The scraper works correctly on different websites with varied command combinations. Configurations can stop working if the tested website is changed since it relies on certain elements that need to be present.

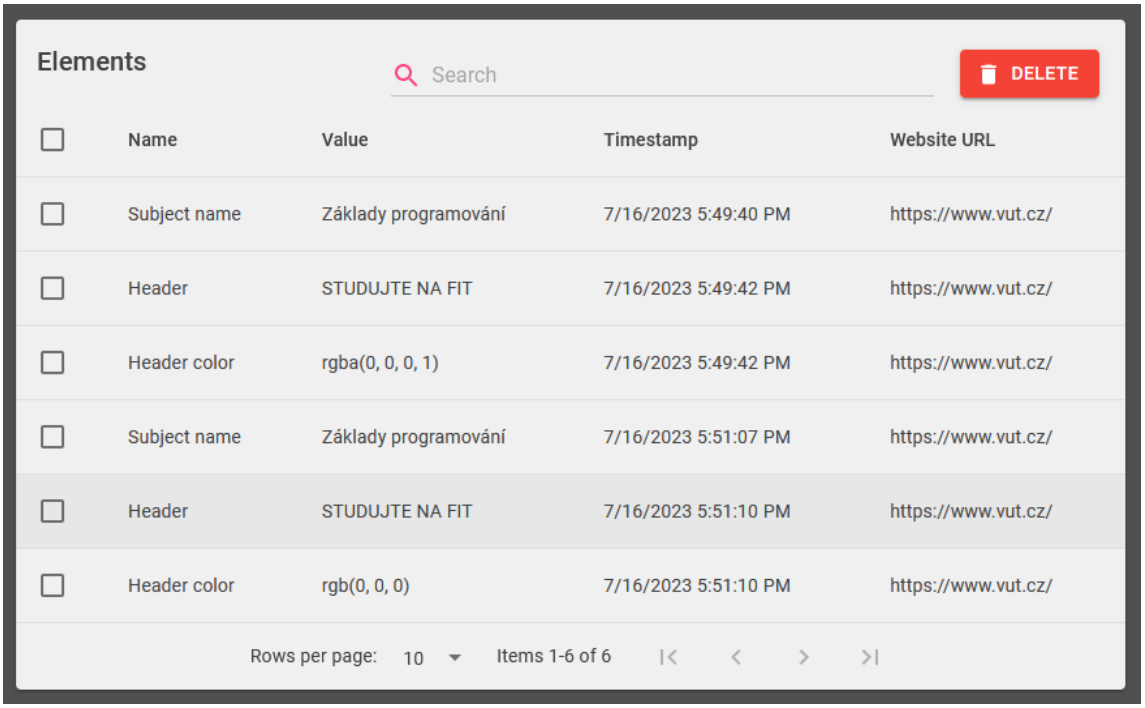

Figure 5.19: WebScraper - Extended configuration file for VUT website

That concludes the testing of this thesis. All required tests passed without exception. The web application works as intended. The whole project communicates together well. Results from the scraping are correctly stored in the database, and the web application displays them perfectly. Selenium commands behave as expected.

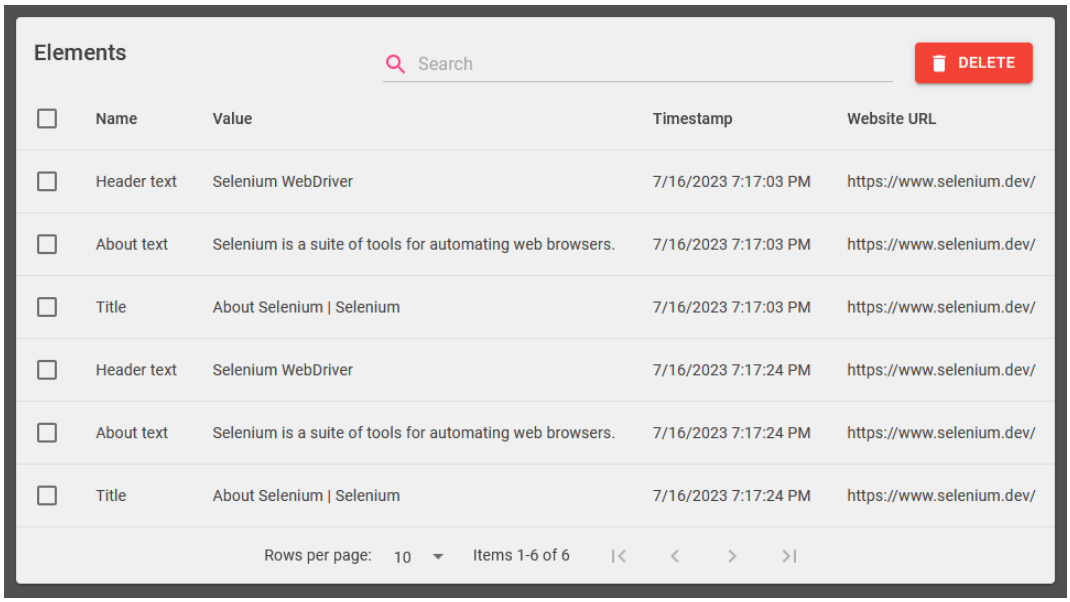

Figure 5.20: WebScraper - Extended configuration file for Selenium website

Chapter 6

Conclusion

The goal of this thesis was to create a web scraper that can scrape dynamic websites with the usage of a configuration file, which navigates the scraper to a website and its elements. This thesis met all requirements of the specification.

This goal was achieved by creating a .NET application for the scraping, SQLite as a database that holds scraped data, and a web application in Blazor which serves as a way for the user to control the scraper, view its results and create configuration files for it. All those parts work together seamlessly, and each part works as intended based on executed tests.

The most important results are the actual scraping results. Those results correspond correctly with the expected results.

My work could be expanded by creating a tool that could transcribe steps taken from the Selenium IDE used in browsers as an extension. That way, the user could create steps using the Selenium IDE, use the developed tool, and easily use the resulting configuration. That would help reduce the time it takes to make a configuration file for the scraper.

Another possible addition to this thesis could be a page within the web application that would create statistics about a scraped website, like which elements were frequently changed or how many were changed.

Bibliography

[1] Coyier, C. *RGBa Browser Support* [online]. September 2018 [cit. 2023-07-24]. Available at: https://css-tricks.com/rgba-browser-support/.

[2] Dhamani, A. *List of headless browsers* [online]. [cit. 2023-07-24]. Available at: https://github.com/dhamaniasad/HeadlessBrowsers.

[3] Diouf, R., Sarr, E. N., Sall, O., Birregah, B., Bousso, M. et al. Web Scraping: State-of-the-Art and Areas of Application. In: *2019 IEEE International Conference on Big Data (Big Data)*. 2019, p. 6040–6042 [cit. 2023-04-02]. DOI: 10.1109/BigData47090.2019.9005594. Available at: https://ieeexplore.ieee.org/document/9005594.

[4] Glez Peña, D., Lourenço, A., López Fernández, H., Reboiro Jato, M. and Fdez Riverola, F. Web scraping technologies in an API world. *Briefings in Bioinformatics*. april 2013, vol. 15, no. 5, p. 788–797, [cit. 2023-04-02]. DOI: 10.1093/bib/bbt026. ISSN 1467-5463. Available at: https://doi.org/10.1093/bib/bbt026.

[5] JavaTpoint. *How does JavaScript Work?* [online]. [cit. 2023-07-24]. Available at: https://www.javatpoint.com/how-does-javascript-work.

[6] Kiskyte, A. *Headless Browsers* [online]. 26. march 2021 [cit. 2023-04-02]. Available at: https://oxylabs.io/blog/what-is-headless-browser.

[7] Microsoft. *Blazor* [online]. 2023 [cit. 2023-07-20]. Available at: https://learn.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-7.0&WT.mc_id=dotnet-35129-website.

[8] Microsoft. *Entity Framework* [online]. 2023 [cit. 2023-07-20]. Available at: https://learn.microsoft.com/en-us/ef/.

[9] OpenJS Foundation. *JSON Schema* [online]. 2020 [cit. 2023-07-20]. Available at: https://json-schema.org/.

[10] Petersen, H. From Static and Dynamic Websites to Static Site Generators. [online]. 2016, [cit. 2023-07-24]. Available at: https://core.ac.uk/display/83597655?utm_source=pdf&utm_medium=banner&utm_campaign=pdf-decoration-v1.

[11] Pluskal, J. Data Persistence in .NET Applications. In:. ICS - Seminář C, 2023, chap. Repository, UnitOfWork, Facade and Mapper design patterns [cit. 2023-07-24]. Available at: https://github.com/nesfit/ICS/blob/master/Lectures/Lecture_04/Lecture.md#repository-unitofwork-facade-and-mapper-design-patterns.

[12] Refactoring Guru. *Command* [online]. [cit. 2023-07-24]. Available at: https://refactoring.guru/design-patterns/command.

[13] Refactoring Guru. *Facade* [online]. [cit. 2023-07-24]. Available at: https://refactoring.guru/design-patterns/facade.

[14] Selenium. *Selenium* [online]. [cit. 2023-04-02]. Available at: https://www.selenium.dev/documentation/webdriver/.

[15] Selenium. *Selenium IDE* [online]. [cit. 2023-07-24]. Available at: https://www.selenium.dev/selenium-ide/.

[16] Stack Overflow. *Most popular technologies* [online]. 2023 [cit. 2023-07-24]. Available at: https://survey.stackoverflow.co/2023/#technology-most-popular-technologies.

[17] W3C. *XML Path Language* [online]. 2010 [cit. 2023-04-02]. Available at: https://www.w3.org/TR/xpath20/.

[18] W3C. *WebDriver* [online]. March 2023 [cit. 2023-04-02]. Available at: https://www.w3.org/TR/webdriver/.

[19] Zhao, B. Web Scraping. In: *Encyclopedia of Big Data*. May 2017, p. 1–3 [cit. 2023-07-24]. DOI: $10.1007/978\text{-}3\text{-}319\text{-}32001\text{-}4_483 - 1.ISBN978 - 3 - 319 - 32001 - 4$.

Appendix A

Contents of the Included Storage Media

- src/ - Folder with source files.
- src/WebScraper/Configs - Folder with example configuration files.
- latex/ - Folder with LaTeXsource files.
- README.md - README file for the project.
- thesis.pdf - Thesis file.
- thesis-print.pdf - Thesis file for print.

Appendix B

Detailed description of commands

- **AcceptAlert** - This command accepts an alert of the website.
- **AddCookie** - Stores a cookie with specified key and value.

  Arguments:

  - `key` - Key of the cookie.
  - `value` - Value of the cookie.
- **Back** - Returns back to the previously visited page.
- **Clear** - This command is used for clearing out the text field of a form.

  Arguments:

  - `path` - XPath path to a text field of a form that will be cleared.
- **Click** - This command simulates a left mouse click.

  Arguments:

  - `path` - XPath path to an element that should be clicked.
- **DeleteAllCookie** - Deletes all cookies of the website.
- **DismissAlert** - This command dismisses an alert of the website.
- **ExecuteJavaScript** - Executes a provided script.

  Arguments:

  - `name` - Name of the record in the database that is going to store the result of the script.
  - `script` - Script to be executed.
- **Forward** - Moves forward to the previously visited page.
- **ImplicitWait** - This command is useful when loading slow websites. It waits for all elements to load or for a specified number of seconds, after which an exception is thrown. Since there is no way to specify for which element to wait, it is better to use other wait commands.

Arguments:

- `time` - A number of seconds to wait before throwing an exception. The value of this argument can be 0 or bigger.

- **Maximize** - Maximizes the driver. Useful when using commands like `MoveToElement`.
- **MoveToElement** - Simulates a mouse hover over an element. This work only when the driver isn't in the headless mode because then the mouse simulation doesn't work.

  Arguments:

  - `path` - XPath path to an element.

- **Navigate** - Provides a way to navigate to other URLs. It can be used when the user wants to skip some steps to get to the desired page.

  Arguments:

  - `path` - URL, which the scraper will navigate to.

- **SaveAttribute** - Provides a way to save the attribute value of a specified element into a database.

  Arguments:

  - `path` - XPath path to an element of which attribute value will be saved.
  - `attribute` - Name of the attribute in the specified element whose value will be saved into the database
  - `name` - Name in the database that will store the value from the attribute of an element.

- **SaveCssValue** - This command saves a value of specified CSS property from some element.

  Arguments:

  - `path` - XPath path to an element whose CSS property value will be saved.
  - `property` - CSS property whose value will be saved.
  - `name` - Name in the database that will store the value.

- **SaveHtml** - Saves the whole HTML source code to the database

  Arguments:

  - `name` - Name in the database that will store the source code.

- **SaveTagName** - Saves the tag name of a specified element.

  Arguments:

  - `path` - XPath path to an element whose tag should be saved.
  - `name` - Name in the database that will store the tag name.

- **SaveText** - This provides a way to save the text of a specified element into the database.

  Arguments:

  - `path` - XPath path to an element with wanted text.
  - `name` - Name in the database that will store the value from the element.

- **SaveTitle** - Saves the title of the website

  Arguments:

  - `name` - Name in the database that will store the title.

- **SendKeys** - A way to send text into a form field

  Arguments:

  - `path` - XPath path to a form field
  - `text` - Text to be sent into the form field

- **SendKeysAlert** - A way to send text into a form field of an alert

  Arguments:

  - `text` - Text to be sent into the form field

- **SendReturn** - Sends a return key to some element.

  Arguments:

  - `path` - XPath to the element that should recieve the return key.

- **Submit** - Submit command provides a more straightforward way to submit a form. It can be used on any field in a form to submit its value. Another way is to use the Click command on submit button somewhere on a website if it exists.

  Arguments:

  - `path` - XPath path to a form field.

- **WaitUntilClickable** - When executing this command, the scraper waits for a specified element to be clickable before continuing or throws an exception when the time limit is reached.

  Arguments:

  - `path` - XPath path to an element for which the scraper waits.
  - `time` - A number of seconds to wait before throwing an exception. The value of this argument can be 0 or bigger.

- **WaitUntilExists** - When executing this command, the scraper waits for a specified element to exist before continuing or throws an exception when the time limit is reached.

  Arguments:

  - `path` - XPath path to an element for which the scraper waits.
  - `time` - A number of seconds to wait before throwing an exception. The value of this argument can be 0 or bigger.