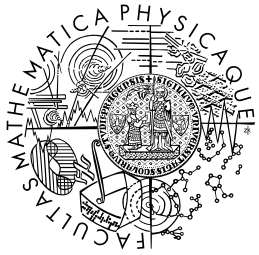

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Markéta Sauerová

Web Browser Recorder

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: Mgr. Kateřina Macková
Study programme: Computer Science
Study branch: Programming and Software Systems

Prague 2022

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

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

In ..... date ....

Author's signature

Foremost, I would like to thank my supervisor Mgr. Kateřina Macková for her time and advice in the course of this thesis. Also, my thanks goes to the whole Apify team for their tips and support, especially to Mgr. Jan Čurn Ph.D. for the opportunity to participate on such project and Bc. Milan Lepík for his oversight and guidance. And last but not least, I want to offer many thanks to Jindřich Bär for developing the Web Browser Robot Library and all his time spent helping me to understand it.

Title: Web Browser Recorder

Author: Markéta Sauerová

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Kateřina Macková, Department of Theoretical Computer Science and Mathematical Logic

Abstract: The goal of this thesis is to develop an intuitive RPA tool in the form of a client-server web application. This tool can be used for recording web automation workflows and their subsequent execution, modification and management. Even though there are many similar tools available, it is difficult to find one that can support a variety of web automation use cases and that is simple enough to be used by untrained users. This thesis aims to build such a solution, a tool producing recordings in the easy to understand Web Automation Workflow format and leveraging from the Web Browser Robot library Interpreter. The result is a powerful GUI recorder providing numerous useful features.

Keywords: browser recorder robotic process automation intelligent workflow files

Contents

Introduction

Web automation plays a vital role in the modern world. It has so far attracted a sizable audience, raging from developers automating the testing of web applications, business owners comparing their goods on the market or analysing the generated online advertisements, e-shops monitoring data extraction, to a common web browser user interaction like subscribing to or unsubscribing from a streaming service, automatically checking the status of a payment or waiting for a discount on a plane ticket. Web automation has numerous use cases. The most important ones are web testing, data scraping and workflow automation.

Software testing is an essential phase of the software development life cycle consuming an average of 40% to 70% of the development process [1]. Today many software applications are written as a web-based applications that run on an Internet Browser. Test automation helps automate repetitive actions like interacting with web elements or filling out long Hypertext Markup Language (HTML) forms, which significantly decreases time and effort and increases efficiency and focus on innovation.

The Internet is a vast repository of knowledge, including intellectual, social, financial, and security-related data. Web scraping can be described as a technique for extracting unstructured data from the web and converting them into structured data that can be stored and later analyzed. Because an enormous amount of data is constantly generated, web scraping is widely acknowledged as an efficient and powerful technique for collecting them [2][3]. There are various techniques for data extraction including regular expression matching, Document Object Model (DOM) parsing or gathering data from the site's Application Programming Interface (API) by Hypertext Transfer Protocol (HTTP) request. However, the most common technique is the use of web automation tools [4].

Each of these use cases focuses on a slightly different aspect of web automation. However, sharing the common way of production involves usually programmed, predetermined, organized series of actions and conditions. This poses a severe constraint. When it comes to websites changing their structure, blocking by anti-bot mechanisms or even A/B testing, an automation engineer needs to re-implement the original solution, which proves to be a tedious task or which could create a complex and hard-to-maintain solution. The ultimate goal of the recorder is to make the automation process easy, efficient and reliable, boosting the productivity of the development. Furthermore, the economic relevance of web automation increases the importance of improving its quality.

As of today, there are several web browser recording tools available. The majority of them are Chrome browser extensions, such as DeploySentinel Recorder and WildFire among others, as a browser extension is the easiest way to manipulate the browser. It has a better access to the state of the browser and its DOM. Because of the Graphical User Interface (GUI)'s usual complexity, it is difficult to navigate through a recording extension and provide a positive user experience, prolonging the time required to completely understand how to operate these tools. Additionally, they usually lack one of the following abilities: to edit or improve workflow that has already been recorded, manage recordings and their runs. Since management frequently rests in the user's hands, it is challenging to

maintain the recordings without programming experience.

The final objective of this work is therefore to produce a self-contained Robotic Process Automation (RPA) tool that can create a recording in the WAW format, granting the power of non-linear interpretation, and that is easy to use due to the simple GUI that adheres to appropriate User Experience (UX) principles. This tool should be helpful for various interactions with the recorded workflows, including editing, running, managing and visualizing them.

The web browser recorder completely removes the need of writing code to automate scenarios. Non-technical users, the automation community, and engineers can benefit greatly from this, making their job more reliable and efficient. However, difficulties come while designing such a tool. There are numerous user options and distinct requirements for many different use cases. This is the key justification for my decision to build the recorder as a web application with a simulated browser running on the server-side.

Even though, this approach comes with some browser interaction issues that require to be solved, a client-server application, in my perspective, offers the necessary functionality for simple yet powerful recording tool.

Thesis Overview

The first chapter 1 provides an overview of the key technologies used, including JavaScript programming language, TypeScript programming language, Node.js, React.js, Playwright and Web Browser Robot (WBR) library. Without their knowledge, it would be impossible to understand the proposed solution. The second chapter 2 provides a general overview of the requirements for the Web browser recorder. It also analyses existing solutions and describes the proposed one in detail. The third 3 and fourth 4 chapters offer an insight into the implementation process including various examples from the completed software. Furthermore, these chapters contain descriptions of some possible alternative approaches. Last but not least an attachment A with the project's repository description is present.

1. Used Technology

An overview of the used technologies is present in the first chapter. We describe JavaScript programming language, TypeScript programming language, React.js and various web automation tools including Playwright. Furthermore, a brief introduction of the WBR library with references to its documentation is offered.

This chapter should provide the reader with the knowledge necessary to understand the suggested solution.

1.1 JavaScript Language

JavaScript is a programming language that is one of the core technologies of the World Wide Web, alongside HTML and Cascading Style Sheets (CSS) [5]. Its association with the web browser makes it one of the most popular programming languages in the world [6]. As of 2022, 98% of websites use JavaScript on the client side[7], often incorporating third-party libraries [8]. A dedicated JavaScript engine can be found in all significant web browsers.

JavaScript may be defined as an ECMAScript standard compliant high-level, typically just-in-time compiled language. It has dynamic typing, prototype-based object-orientation, and first-class functions [5]. It supports event-driven, functional, and imperative programming styles [5] and has APIs for working with text, dates, regular expressions, standard data structures, and the DOM [6].

JavaScript Engines A JavaScript engine is a software component that executes JavaScript code. Brendan Eich developed the first JavaScript engine in 1995 for the Netscape Navigator web browser, which evolved into the SpiderMonkey engine, still used by the Firefox browser today [9]. Every major browser has a JavaScript engine, which is typically created by the browser's vendor. Google developed the V8 JavaScript engine for the Chrome browser. The V8 is considered the first modern engine. It was introduced as part of the Chrome browser in 2008, and its performance at the time was superior to any prior engine [10]. Initially only used in web browsers, they are now essential parts of some servers and many different applications. The most popular runtime is Node.js.

Node.js Node.js is an open-source, cross-platform, back-end JavaScript runtime environment that runs on the V8 engine, which was designed to build scalable web applications [11]. JavaScript can be used by developers to create command-line tools and for server-side scripting, which generates dynamic web page content before sending the page to the user's web browser. As a result, Node.js unifies the development of online applications around a single programming language as opposed to using separate languages for server-side and client-side. Node.js uses an event-driven architecture capable of asynchronous input/output operations and non-blocking models, making it altogether lightweight and efficient [11].

1.1.1 TypeScript Language

Nowadays, the fashion in most programming languages demands strong typing. The theory behind this is that strong typing allows the compiler to detect a large class of errors at compile time [6]. Detecting these errors rather sooner than later can lower their overall cost.

Due to the fact that JavaScript is a loosely typed language [6], type errors cannot be caught by the compiler. This is the reasoning behind the introduction of TypeScript. Built on JavaScript, TypeScript is a tightly typed programming language.

TypeScript Compiler The code is usually parsed by a compiler and transformed into an Abstract Syntax Tree (AST). An AST is a tree representation of the abstract syntactic structure of code written in a formal language. The compiler then converts the AST to bytecode, which is later evaluated by the runtime. The way that TypeScript works differs from other languages such as JavaScript or Java. TypeScript compiles the code not into bytecode, but into JavaScript code. Moreover, TypeScript verifies the code's type safety before producing JavaScript code using a type checker [12]. This is the main reason why TypeScript makes the code safer.

Type System A type checker utilizes a type system, which is a set of rules used to assign types to the program. Although the modern languages brought plenty of different type systems, there are generally two main ones: the types are either explicitly provided or they are inferred automatically[12]. As TypeScript supports both, it is possible to annotate the types in a *"value: type"* form, otherwise TypeScript will infer the types from assigned values.

TypeScript vs JavaScript In comparison to JavaScript, TypeScript does not automatically convert types at runtime. It validates the types' correctness at compile time, throwing most of the errors at compile time too. Furthermore, TypeScript's type inferring improves the IntelliSense experience in modern code editors. On one hand, TypeScript makes development longer and harder as the type conversions must be done explicitly. On the other hand, it avoids unexpected behaviour at runtime.

1.1.2 React.js Library

React is a declarative and component-based JavaScript library for building user interfaces. Its declarative and modular nature makes it easy for developers to create and maintain reusable, interactive, and complex user interfaces [13].

Large applications that display a lot of changing data can be fast and responsive if built with React, as it takes care of efficiently updating and rendering specific components when data changes[13]. React achieves this rendering with its implementation of a virtual DOM, setting React apart from other web User Interface (UI) libraries that handle page updates with expensive manipulations directly in the browser DOM [13].

React uses JavaScript Extensible Markup Language (JSX)[1] syntax. JSX can be compared to a template language, which produces React components utilizing the full power of JavaScript. React embraces the fact that rendering logic is inherently coupled with other UI logic: how events are handled, how the state changes over time, and how the data are prepared for display. In contrast to separating technologies by putting markup and logic in separate files, React separates concerns with units called components that contain both.

In the past, there have been various React component types, but with the introduction of React Hooks it is possible to write an entire application using only functions as React components. React Hooks were introduced at React Conf in October 2018 [14] as a way to use state and side-effects in React function components.

MUI Component Library Material User Interface (MUI) library provides ready to use and customizable React components that implement Google's material design [13]. First announced on June 25, 2014 at the Google I/O developer conference [15], Google's Material Design is a design language that seeks to unify the user experience across Google products and across platforms.

1.2 Web Automation

Automation in general refers to a broad variety of technologies that eliminate the need for human intervention in various operations. Web automation usually refers to browser automation or automation of certain actions performed on the internet. Around the year 2010, the web automation market began to expand rapidly [16]. Bringing several notable tools developed over the next years.

PhantomJS[2] is one of them. It is a headless WebKit browser scriptable with JavaScript, originally released in 2011. *PhantomJS* was used to extract information from web pages, capture screenshots, monitor the network and run functional tests.

Another notable project is Selenium[3], first released in 2014. It currently includes a range of tools and libraries enabling web browser automation.

Another tool worth mentioning is Puppeteer [4] developed by Google. Puppeteer is a Node.js library that provides a high-level API for controlling Chromium and Chrome browsers over the DevTools Protocol.

1.2.1 Playwright

Playwright is a web testing and automation framework developed by Microsoft. Originally released in 2020, it is the successor of the Puppeteer automation library, see the respective commit[5].

Mostly everything that could be done by a user manually, including opening a browser and manipulating a page, could be done with Playwright. Unlike the

[1] https://reactjs.org/docs/introducing-jsx.html
[2] https://phantomjs.org/
[3] https://www.selenium.dev/
[4] https://developer.chrome.com/docs/puppeteer/
[5] github.com/microsoft/playwright/commit/53cdbc5688935810b8c51ec86e3037c24bbcfac1

Puppeteer library, it allows to control Firefox, WebKit and Chromium-based browsers. Playwright supports most popular programming languages, including Node.js, Python, Java and .NET. Not only it is cross-browser, cross-platform and cross-language, but also constantly updated, making Playwright probably the best choice when it comes to web automation projects.

1.2.2 Robotic Process Automation Tools

In conventional workflow automation technologies, a software developer creates a set of operations to automate a task and to connect to the back-end system using internal APIs or specialised scripting languages. On the other hand, RPA systems create the action list by observing the user carry out the task in the GUI of the program. They then execute the automation by repeating those actions directly in the GUI. For websites and programs, which lack the APIs for such purpose, this lowers the barrier of using an automation.

RPA tools and GUI testing tools are very similar in terms of technology. What makes the RPA tools different is that they allow to handle data within and between various applications.

1.3 Web Browser Robot

The WBR was developed by Jindřich Bär in collaboration with Apify and MFF UK, in the years 2021-2022, for the purpose of web automation. It consists of the Interpreter, which can read and execute workflows, and ”smart” workflow format definition, called the WAW format.

1.3.1 WAW format

The WAW format[6] is a declarative format for specifying web-related workflows. It enables the user to control the automation flow with conditional expressions, allowing to make decisions based on the websites content. It is also easily parsable, based on JSON, which greatly simplifies validation, visualization and third-party adoption[17].

A workflow consists of a list of actions paired with their respective conditions. This is a very simple yet powerful mechanism. It also allows the robot to act on its own, recovering from potential mistakes and being less dependent on the actual environment [18].

1.3.2 Interpreter

The Interpreter is the core part of the WBR library. Its purpose is to read the workflow, act upon it, handle the internal concurrency and provide the execution results[18]. See the Interpreter documentation on GitHub[7].

[6]https://github.com/apify/waw-file-specification

[7]https://github.com/barjin/wbr/blob/main/docs/wbr-interpret/interpreter.md

2. Problem Analysis

The second chapter describes the requirements for the Web browser recorder application. Moreover, we discuss and analyze the current competition of automated recorders. Also the solution to the assignment is proposed. Finally, the planning phase of the implementation is explained.

2.1 Requirements

For a variety of use case scenarios involving web automation and its subcategories, the user should be able to choose from many capabilities offered by the web browser recorder. The following can be classified as the main functionality requirements.

Generating Selectors The application server must have the ability to produce unique and correct selectors for specified HTML elements. This step is essential for producing the right conditions for the WAW format.

Web Automation Workflow generator The application server must understand how to create proper JSON data that adheres to the WAW specification format. A few heuristic methods based on applying the WAW format to some real use cases for web automation should be included in the generator.

Simulated Browser GUI The user should be able to interact with the remotely initialized browser. On the client side of the application, a method must be available for rendering the server-side browser view.

Interpretation The user should be able to interpret the recorded workflow. Interpretation makes sense in two different areas. The first one is inside of the recording session as a means to update the workflow, providing the possibility to pause the execution. The second one is dependent on the implementation of the workflow execution management and appears when the execution is initiated as a run. For recording interpretation the WBR's Interpreter is used.

Recordings Management All the previously recorded workflows should be accessible by the user. These recordings can be updated using the recorder's editing options. The user must have the ability to do both, add new recordings and remove existing ones. The recordings can be divided to two categories: those with parameters and those without. Both should be subject to interpretation and the user should have access to any data gathered throughout their execution.

Application Design Overall, the tool should be easy to use with a simple design. The interface should offer a good user experience. UX features linked to the complexity of the recording, including recording custom actions, choosing the right selector and configuring the interpreter, should be taken into consideration. Additionally, methods for displaying notifications and modals are required.

2.2 Competition

As stated in the Introduction part of this work, there are several web browser recorders available. In this part, a brief analysis of their functionality and comparison of their ideas are present.

2.2.1 Chrome Browser Extensions

There are many Chrome browser extensions dedicated to recording and automatically generating code. An extension is executed inside of the launched browser, having its context accessible. Interaction with the active page is therefore easier. Let us have a look at the two most notable, yet distinctive recorder extensions.

DeploySentinel Recorder Recorder from DeploySentinel[1] is a user-friendly and simply designed extension. Simply stepping through the website while recording makes the extension convert captured user interactions into generated Cypress, Puppeteer or Playwright scripts. The great part about these scripts is that they include human-readable comments explaining what every line do. It highlights and shows selector for elements on the page when hovering over them. This makes the selector choosing process straightforward. Moreover, the recorder prioritize stable selectors.

Two main issues have been encountered while using this extension. Firstly, it does not let a user edit the generated code while recording, making parametrization or usage of regular expressions harder. Secondly, pop-up use cases, like log in using Google account, are impossible as the recording happens only on one active page.

Wildfire The Wildfire[2] extension records actions and replays them using a simulator. When actions are recorded or simulated, it produces a log which can be reviewed. Additionally, it offers a Workflow Editor to manipulate the behavior of the simulation.

On one hand, Wildfire simulation abilities are impressive. On the other hand, the UI is complex and counter-intuitive. It generates CSS selectors as the full path from the page's root. Because of that understanding which recorded action does what from the workflow editor is almost impossible. It shows that the separation of the simulation and the editor makes the usage harder.

2.2.2 Other solutions

There are plenty of other recording solutions on the market. Notable mentions are UiPath's "app/web recorder"[3] that can attach the user's browser inside the recording application, Dexi.io[4], a web based GUI recorder.

Automated workflow recording seems to be modern and fast evolving trend, part of which many software companies want to be. Let us have a closer look at examples from Google and Microsoft.

[1] https://www.deploysentinel.com/recorder
[2] https://wildfire.ai/
[3] https://docs.uipath.com/activities/docs/app-web-recorder
[4] https://www.dexi.io/

Chrome DevTools Recorder Chrome DevTools developer team launched their recorder[5] as a preview feature in November 2021. Besides recording a workflow, it allows to generate a detailed performance trace or a Puppeteer script. It has a friendly and simple UI, but the recorder lacks in features. It operates only on one page, making automation use cases involving pop-ups impossible to record. For now, special actions like screenshot are not supported. Even though the recorder associates multiple selectors to the actions, there is a high probability that the selectors are going to be constructed as the full path selector sequence, starting from the root of the page. Such selectors capture dynamically generated classes, making them fail on replay. As a result, Devtools recorder, for example, does not handle a simple Google search by clicking on the auto completed search bar preview.

Power Automate Microsoft developed a recorder application, called Power automate[6]. This recorder differs from the others, because it can record both, web or desktop application workflows. Power automate allows execution of the workflows only inside of the application. It generates a JSON representation of the actions and triggers. This representation includes all inputs, such as the text and expressions used. The format of the representation as well as how to interpret it are private information.

2.3 Proposed solution

The web browser recorder is an easy-to-use RPA tool built in a form of a client-server web application. A user can record a workflow in the simulated browser GUI, using Playwright in the background. The workflow is being recorded in the WAW format. This automation format is unique in that it can be interpreted in a non-linear manner. As a result, a workflow may now be designed as a deterministic finite automaton that can match numerous states according to the conditions rather than just being a set of sequential rules. We achieve that by processing the WAW workflow through an Interpreter from the WBR library.

2.3.1 Client

The front-end or client-side of the application is built using the JavaScript React library. This library belongs among the most widely used technologies for creating websites. With approximately 4% of the market share, React easily surpasses the two other well-known frameworks, Vue.js and Angular, according to the statistics [8]. Together with the TypeScript support for JSX it becomes a powerful tool. TypeScript Extensible Markup Language (TSX) can correctly model the patterns used in React code-bases like *useState* [19]. This enables to obtain the advantages of strong typing even in the client side.

React is a client framework with a lot of benefits. The UI's interactivity, proficient data binding and component re-usability are a few of its outstanding

[5]https://developer.chrome.com/docs/devtools/recorder/
[6]https://docs.microsoft.com/en-us/power-automate/

features. React also enables significant data changes resulting in automatic alteration in the selected parts of the UI, making programmatic component reloading unnecessary.

As opposed to the class implementation of the components, which will become obsolete in the future, the recorder adopted the functional approach, using hooks for manipulation of the component's life cycle. When it comes to coding, the modern React approach is undoubtedly more understandable and easier to adopt.

User A user could be anybody. However, users can be separated into two main groups: developers and people with no development experience. Since the tool should be simple to use and have an intuitive and friendly UI, both groups should be able to easily interact with the recorder on the first try. Understanding the advanced features shouldn't need more work than reading the information that is provided in the WBR library documentation. The specification of the WAW format is the sole unusual concept that is needed to be understood.

Setup React application was created by *create-react-app* with the TypeScript template. This tool was developed to help with React web application setup. It eliminates the barrier between writing code and configuring a compilation environment. Create React App (CRA) makes use of Babel and Webpack even though they cannot be seen as dependencies in the produced *package.json* file. The configurations are hidden inside the *react-scripts* package. All the packages required to make React function in browser are listed in the *react-scripts package.json* file. There are 58 packages for that.

Customized scripts, described by Figure 2.1, enable simultaneous server and client start-up, including React's iconic hot reloading. Hot reloading allows to see the changes that have been made in the code without reloading the entire app. React Native tracks which files have changed since the last save and reloads only those.

```
"scripts": {
  "start": "concurrently -k \"npm run server\" \"npm
  run client\"",
  "server": "./node_modules/.bin/nodemon server/src/
  server.ts",
  "client": "react-app-rewired start",
  "build": "react-app-rewired build",
      ...
}
```

Figure 2.1: Scripts definition from *package.json* file

Page Structure And Design The goal is to plan a simple yet effective application design appealing to a wider audience. The controls should be easier to grasp with such a design, making the program more user-friendly.

A review of online editors and other remote browser applications was done before designing the wireframes for the recorder. The online form editor Typeform[7] served as the primary source of inspiration. Very little of the design was influenced by the competition that already exists. Apify Console[8] served as another source of inspiration, mainly because recordings are in line with the Apify Actor concept[9] as they can be run and have variable input.

Please note that the following images do not represent the final application look, they are only the UI wireframes. Wireframes are set of images displaying the functional elements of a page, typically used for planning an application's structure and functionality.

The mentioned Wireframes are also available at *Whimsical*[10] for further inspection.

The application's initial screen contains a few important features, see Figure 2.2. The user can view previously recorded workflows with a variety of extra information, including the count of *where-what pairs*, the date of the most recent update, and the number of runs. The rows of the recordings table act as an interface for interaction with different control buttons. Namely, the user can run, edit, delete or update recording parameters. Furthermore, the user has the option of starting a new recording session or switching to the runs or tasks. The original idea of changing views from list to grid was considered, but it lost its sense throughout implementation.

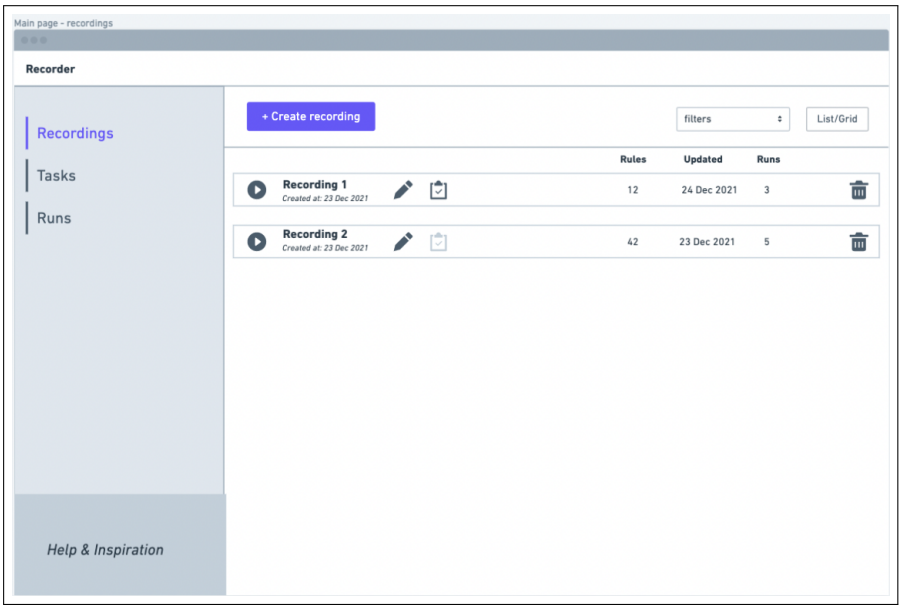

Figure 2.2: Wireframe - Main page with the table of recordings

The runs table page content is very similar to that of the initial page, see Figure 2.3. It consists of a table listing all interpretations of the recordings started in the application, together with extra information. Note that table of

[7]https://www.typeform.com/
[8]https://console.apify.com/
[9]https://apify.com/actors
[10]https://whimsical.com/browser-recorder-8ZogzyekRKFNZumg4EeS5j

runs in the finished recorder needed to be more complicated as individual runs contain logs and output data.

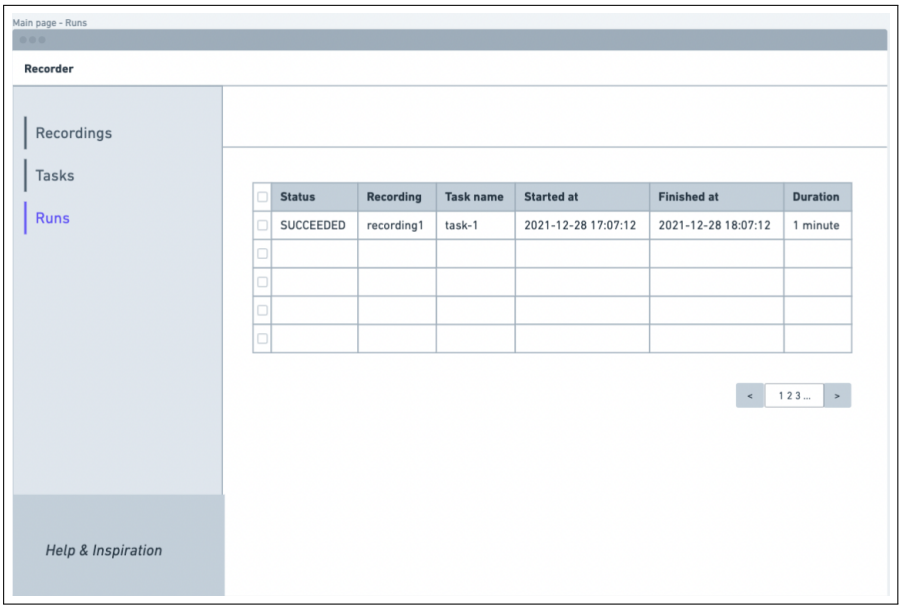

Figure 2.3: Wireframe - Main page with the table of runs

The recording page is divided into three main elements, see Figure 2.4. On the left side panel there are controls for interpretation and a recording preview. The simulated browser window is situated in the middle of the screen. It is composed of the browser tab manager, browser window and navigation bar. Lastly, the right side panel contains a settings form for every possible custom action and the logic tab, a feature presenting a visualized workflow interpretation route graph that has been given up during the development process. The navigation bar on top is always visible, enabling users to switch between pages and states.

As the implementation phase began, the overall structure and design underwent a lot of quick changes associated with a better understanding of the developed features.

Styled components library for CSS styling is used with the help of MUI library to achieve a minimalist design. MUI toolkit offers basic and customizable components for building React applications. They fundamentally improve the interface while reducing the amount of time needed for the atomic components creation and styling. For minor style adjustments, inline CSS is used.

Code Structure The top-level client directory structure is defined as follows.

- The **API** folder implementing the HTTP request methods communicating with the server endpoints.
- The **components** folder containing the reusable components of different complexity, starting from the least complex ones. The directory is divided into *"atoms, molecules and organisms"*[11] sub-directories, allowing for a hi-

[11] https://medium.com/@janelle.wg/atomic-design-pattern-how-to-structure-your-react-application-2bb4d9ca5f97

erarchical component structure. The *"atoms"* are isolated structures from which the *"molecules"* are built. The *"organisms"* are then constructed from the *"molecules"*. Both *"molecules"* and *"organisms"* can contain other components of their kind.

- The **constants** folder declares constant values needed throughout the client side.
- The **context** folder defines the context components which provide data to its children by using their custom hook method.
- The **helpers** folder contains the helper functions.
- The **pages** folder consists of pages, also called views, which are higher-level components that are using all elements necessary for assembling a specific view.
- The **shared** folder contains server-shared definitions of types and constants.

The source directory consists of these folders as well as the *index.tsx* file, which is the main entry point, and *App.tsx* file, which sets up the application.

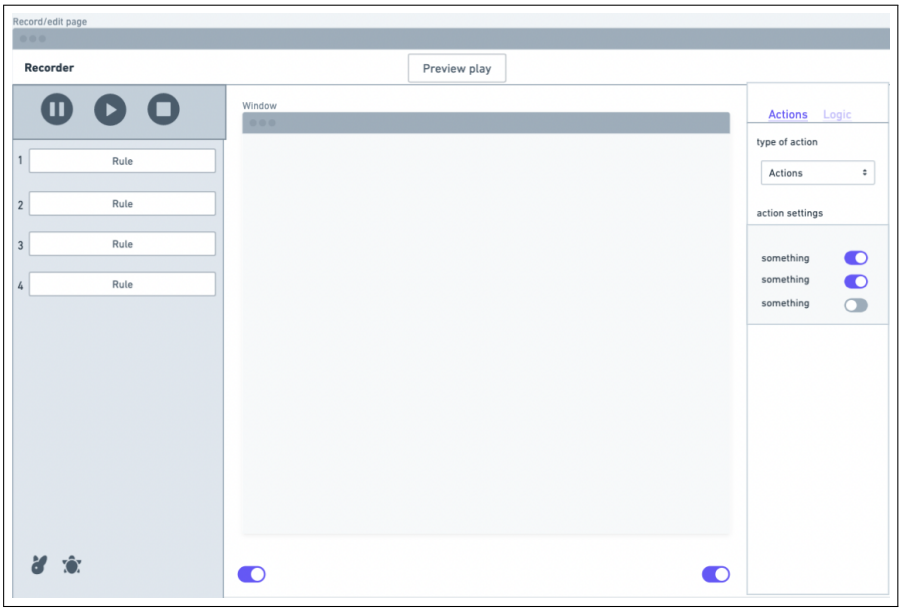

Figure 2.4: Wireframe - Recording page

2.3.2 Server

The back-end or server-side of the application is built using TypeScript and the Express framework. Express is a minimal and flexible Node.js web application framework that provides a robust set of features for implementing API. With a substantial user base, the Express framework has a supportive community [20].

In comparison to these technologies, the Playwright and Web Browser Robot libraries are both used on the back-end. The browser is started and interacted with using the Playwright library. The workflow in the WAW specification format is executed using the WBR's interpreting algorithm.

Architecture Two key categories cross over in the back-end functionality: the browser and the workflow management. The browser management section offers setup and maintenance of remote browser instances. Whereas the workflow management part is responsible for workflow generation in the WAW format and workflow interpretation.

There are four main classes covering these functionalities and their interaction, see Figure 2.5. The *RemoteBrowser* and the *BrowserPool* are classes in the browser management section. In the workflow management section, there are the *Generator* and the *Interpreter* classes included.

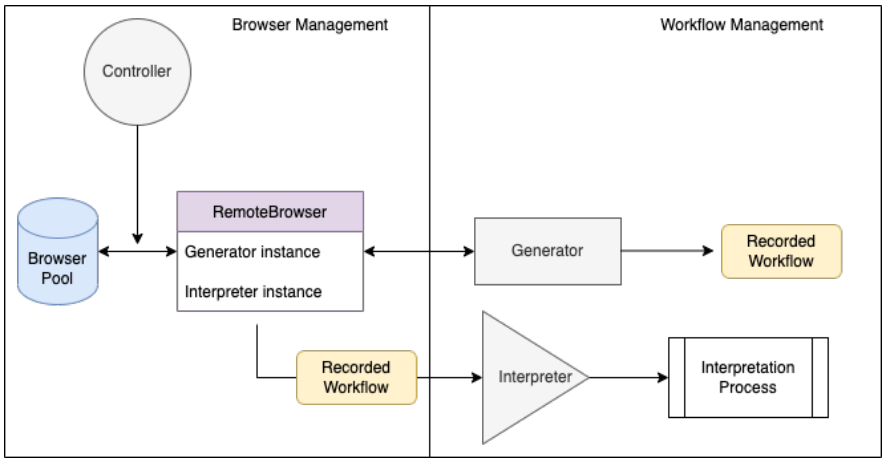

Figure 2.5: Server architecture diagram

A controller, consisting of helper functions used for setup and management of browser instances, must be included in the browser management section. The *RemoteBrowser* is also bound to the *BrowserPool* by the controller. The *BrowserPool* is a dictionary of all the active *RemoteBrowser* instances indexed by their id.

Storage To comply with the recordings management requirements, the server must store recordings as well as runs. Both are saved with meta information, including the recording's name, the most recent update date, and others.

This application will store files on the file system. The database can be used in the future for a production solution ready to be deployed on the Internet.

2.3.3 Communication

An important part for a client-server application to work is the mutual communication.

Communication between the server and the client works in two ways. The first method is having a Representational State Transfer (REST) API, defined by the server and used by the client. The second method uses a dedicated web socket connection for data transfer.

REST API The back-end offers an API that conforms to the constraints of REST architectural style.

An API is a set of definitions and protocols. It could be referred to as a contract between an information provider and an information user, establishing the content required from the consumer (the call) and the content required by the producer (the response). In other words, if an interaction with a computer or system is required to retrieve information or perform a function, an API helps to communicate what needs to be done for that system, so it can understand and fulfil the request. An API is a mediator between the clients and the resources held by the server.

REST is defined as a specific software architectural style characterized by a group of selected constraints. Beneficial system properties and sound engineering principles are evident in a system designed to conform to these constraints [21]. HTTP 1.1 defines a set of request methods that indicate the action taken on a resource. Of those available, the ones of greatest significance using a RESTful approach are GET, POST, PUT, and DELETE [21].

The recorder's API consists mainly of methods needed to initiate and maintain a recording session, interact with the generated workflow and helpers for storage. The endpoints are described in the chapters 3 and 4. Technology used to define API is router from Express.js described in the introductory part of the Server section.

Socket.IO For web socket communication, the application uses the *Socket.IO* library. This library enables low-latency, bidirectional and event-based communication between a client and a server [22]. It is built on top of the WebSocket protocol and provides additional guarantees like a fallback to HTTP long-polling or automatic reconnection [22].

WebSocket is a computer communications protocol providing a full-duplex communication channels over a single Transmission Control Protocol (TCP) connection.

WebSocket communication plays an important role in the project. Due to its great properties, it is used as the main communication channel for frequent data transfer between the server and the client. This communication is used in both ways to achieve the desired effect.

3. Active Recording View

In this chapter, the recording page and all its associated features are described in more detail.

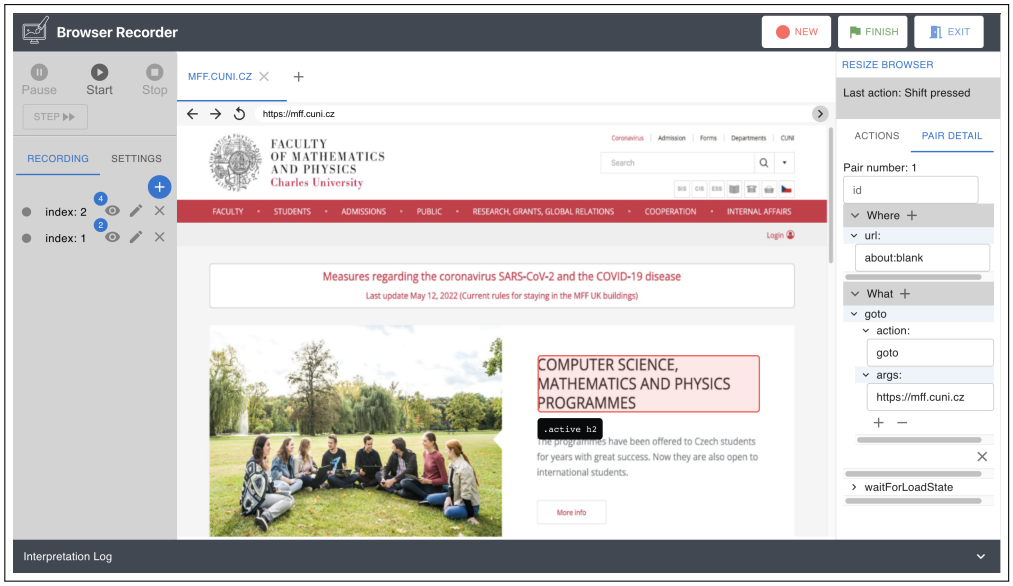

Figure 3.1: Example of the active recording view

The recording page consists of the left side panel, the browser window and the right side panel, see Figure 3.1.

The Left side panel includes the controlled interpretation buttons group, interactive preview of the recorded pairs and interpretation settings. The browser window consists of the tab manager, the navigation bar and the browser window, rendering images from the remote browser instance. Finally, the helper button for browser window resize, the last action noted by the recorder and recorded pair detail and custom action settings tabs can be found on the right side panel.

On top of that, the interpretation log appears at the bottom. The log is needed when recording interpretation is performed. The user can review the output data or debug information concerning the current execution in the log.

Besides the recorder's name and logo, the navigation bar contains the main controls. Namely, the *record* button, allowing to start a new recording session from the active recording view, the *finish* button for storing the recorded workflow and referencing the user back to the recordings management view, and the *exit* button for deliberately going back to the recordings management view.

3.1 Browser Simulation

A simulation must be made to provide a genuine impression of interaction with a browser. Unlike an emulation, a simulation does not mimic all of the software features. As a result, the simulation lacks the conviction of a real browser, yet it nonetheless improves the UX.

The simulated browser has an easy-to-understand structure, an interactive rectangular window, a navigational bar and a tab manager, see Figure 3.2. All of these features are required for supporting a wide range of automation scenarios.

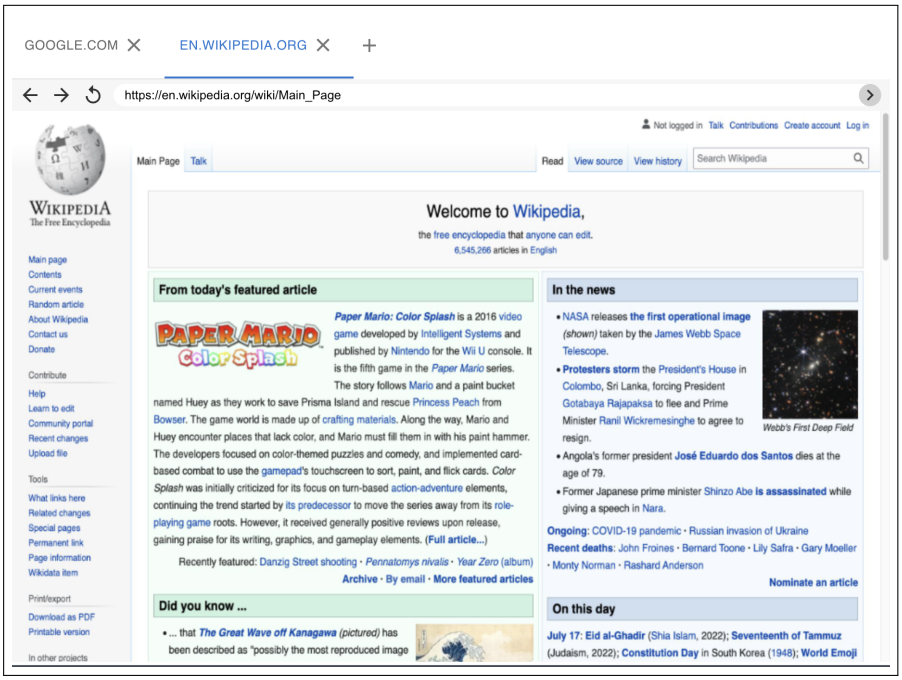

Figure 3.2: Simulated browser

The browser simulation for the recording session can be controlled by using one of the following API endpoints.

1. `GET /record/start`
2. `POST /record/start`
3. `GET /record/stop/:browserId`

These endpoints make a call to the *controller* component, which either initializes a new *RemoteBrowser* instance, assigns it a generated id, or stops an already initialized one, freeing up the resources. The id is referred to as the *browserId* which is used for retrieving the correct instance from the *BrowserPool*.

While initializing, a new dedicated namespace[1] is added to the existing socket connection. A namespace is a communication channel that allows to split the communication logic over a single shared connection, also called multiplexing. This means that multiple *RemoteBrowser* instances could be spawned and interacted with. This interaction and many other events are mapped to the right socket namespace, using the *browserId* as a dynamic namespace name.

When using the first endpoint a remote browser starts with the default parameters. The browser running on the server is a headless Chromium browser

[1]https://socket.io/docs/v4/namespaces/

by default. This can be changed through the second endpoint, defining the Playwright launch options[2] in the request body.

Because on the front-end the page structure is important, the browser is situated inside of the dedicated grid space, changing in size accordingly and dynamically to the user's screen size. Due to the properties of the canvas HTML element, the width and height of the browser window must be computed on page load. This creates an issue when changing the application's size by minimizing or maximizing the user's browser. As a result, the simulated browser is able to recompute the height and width. The re-computation takes place after reload or after a *browser resize* button have been clicked on.

3.1.1 Browser Window

For setting up an interactive window simulating the browser, which runs on the server, two key points have to be solved. The first point is what communication channel should be used. The channel should have low-latency, and a frequent data transfer is expected. The second point is how and where to render the obtained data on the front-end.

As stated in the Proposed solution part, there are two main communication channels available, web sockets and HTTP API calls. The full-duplex websocket protocol, with unique properties, is an obvious choice for solving the first point.

For solving the second point, there are two main candidates. The *canvas* HTML element is used to draw graphics, like images, by JavaScript on the website, and the *iframe* element, which specifies an inline frame. An inline frame is used to embed another document within the current HTML document. At first glance, *iframe* can seem like a better choice in contrast to displaying images. However, turns out that sending images through the network is less resource heavy compared to sending the optimized page's DOM to the *iframe*. As there is no way to transfer the whole browser application, the *canvas* element was chosen.

The process of browser simulation is simple. The *canvas* element draws an images in the Joint Photographic Experts Group (JPEG) format to a dedicated space. The remote browser's page is subscribed to a screencasting session on the Chrome DevTools Protocol (CDP)[3] level. A screenshot is then sent through the socket channel every time the active page updates, encoded in the Base64 form.

In addition to the visual effect, the browser window must know how to detect events from the user. This is done by listening to various types of events on the *canvas*. Information associated with the events, like coordinates or event type, is communicated to the server, where event simulation and workflow generation is performed using these data. Currently supported events are: *mousedown*, *mousemove*, *wheel*, *keydown* and *keyup*.

The *keydown* event is handled in a special way, which supports most of the keyboard shortcuts. Especially the copy-pasting and upper case letters. The *mouseclick* event tracks if navigation or new tab was the result of the click. The other events are not reflected in the result of the recording but they are simulated in the remote browser achieving their intended effects.

[2]https://playwright.dev/docs/api/class-browsertype

[3]https://chromedevtools.github.io/devtools-protocol/

3.1.2 Navigation Bar

A simulated browser navigation bar includes the *reload*, *back* and *forward* buttons, as well as the location bar where Uniform Resource Locator (URL)s are entered. This component is essential for web automation. It allows users to navigate to other pages.

A new tab always starts on the blank page *about:blank*. The URL can be inputted in multiple ways, including or not the protocol and the hostname part. On the other hand, the URL must always contain the full domain name. Other parts, such as the query parameters and fragments are not optimized.

The navigation buttons work as expected. The *back* button navigates to the previously visited website from the history and the *forward* button reverses this change. Navigation buttons are not controlled, meaning they are clickable at any point. The navigation history management is done on the back-end with the use of the Playwright's *goBack* and *goForward*[4] functions. These functions are also reflected on the generated recording, whereas the *reload* button is not.

The important feature of the location bar is that it keeps track of navigation and makes the bar synchronized. When navigation happens for a different reason than deliberate URL input, like when a hyperlink is clicked, the bar needs to update the displayed data. The synchronization is applied even when the workflow is interpreted, leaving the user in the right state for further recording.

3.1.3 Tab Manager

Even though the tab manager is not as important for automation as the navigation bar, it is a useful feature for the management of pages in other tabs. The main reason for its implementation is to support automation use cases involving pop-ups.

A pop-up is a new page opened in the context of another browser tab. They are mainly used for authentication through a different service provider. For example, one can log in to a website using a Google, GitHub or a Facebook profile. These options work by opening a pop-up window requesting the *username* and *password* combination. This window automatically closes after successful authentication, so the recorder also needs to listen for the *close* event on the page associated with the specific tab.

The front-end implementation is very simple, using the tab component from the MUI library with a *close* button for each tab. There is also an *add* button available, adding a new page to the browser context.

On the back-end, a set of events is responsible for the tab management behaviour. These events are utilizing the Playwright methods. Because every page has a browser context property, only the active page is needed for the tab management.

3.2 Recording

Recording is done automatically after a user interacts with the simulated browser. It is implemented as a group of event handlers calling the appropriate *Generator*

[4]https://playwright.dev/docs/api/class-page

functions. The recording interpretation is the only time when user interactions are ignored.

Otherwise, the *Generator* is constantly notifying the client, allowing to update the recording preview, see Figure 3.3.

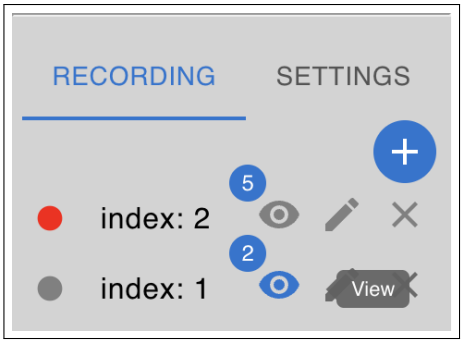

Figure 3.3: Recording preview

The preview is a list of *where-what pairs* located in the recorded workflow. Each preview item consists of a possible *breakpoint* button, provided for the debugging or editing purpose, a clickable index of the pair, reflecting the order of their addition to the workflow, a badge, showing the current number of *what conditions* and a group of three controls. By using these controls, the user can view the whole pair definition, edit the pair in a raw editor or delete the pair from the workflow.

Additionally, the preview is updated with the current version from the *Generator* every 15 minutes by an API call. This behaviour is implemented to minimize the possibility of an error in the preview.

3.2.1 Selector Generation

While persisting some of the previously generated data for a better workflow conditions, the *Generator* must be able to create unique and correct CSS selectors for HTML elements defined by the coordinates of the user interaction.

It does that by executing the *finder*[5] code directly inside the page's context. The *finder* is a CSS selector generator developed by a Google employee.

This selector generator outputs an object containing different selector variants, such as class selector or text selector. The *finder* is wrapped inside of an algorithm used to select the best selector from the *finder*'s output according to the user's action. The algorithm implements basic heuristics, for example if the element has only text, the text selector is going to be used or when an element has a unique id, it is automatically chosen as a selector. This approach leaves room for future improvements.

In addition, the recorder provides a highlighting feature on the front-end. The *highlighter* component gets the generated selector, always best suitable for the *click* action, together with an element bounding rectangle coordinates on *mousemove* event. From this data a highlighting rectangle with the selector is composed and displayed on the browser's emulated window, see Figure 3.4.

[5]https://github.com/antonmedv/finder/blob/master/finder.ts

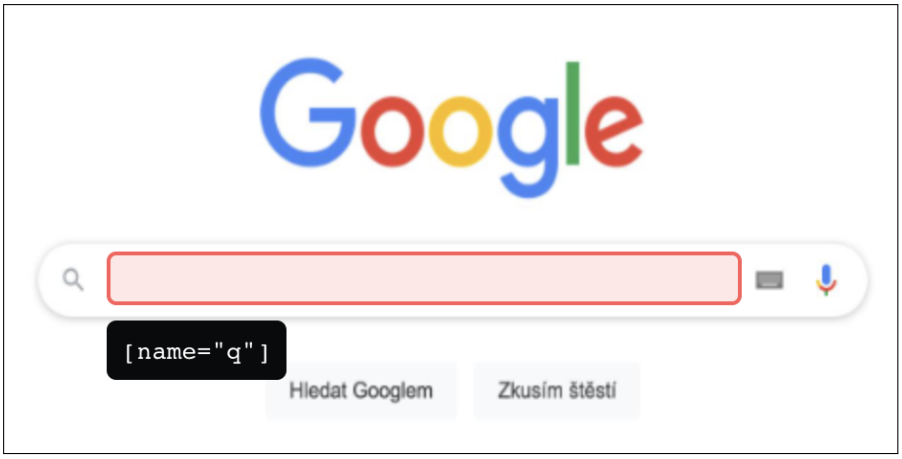

Figure 3.4: Highlighter component example

By using the *highlighter* component a user is able to choose which element with what selector is the most convenient for the action he wants to execute. If no generated selector suffice, the user can edit the generated pair manually.

Selector Dilemma When a user interacts with an element, a problem offering two possible solutions how to generate its selector, appears.

Let us explain it with a simple example. The user navigates to the news website, he clicks on the first article and extracts data from it. Was his intention to always refer to the specific article or rather to the article displayed on the first place by the website? The problem does not have an unambiguously acceptable or preferable solution.

In the recorder, this situation is solved by having the user see the selector before he executes his action or by manually updating the affected part of the recording.

Nevertheless, this poses a constraint on the user's knowledge. He is required to understand the meaning behind the CSS selectors for correct recording generation.

3.2.2 Workflow Generation

The recording is stored inside of the *Generator* instance, initialized for the active *RemoteBrowser*. The *Generator* class also stores the recording's metadata, including the name, create date, update date, number of pairs and parameters array. Metadata are useful while storing the recording.

When a new recording session is started an already existing workflow can be loaded from the storage by using the dedicated endpoint, providing the name of the recording inside of the storage.

Page state An active page can be described by its state. The state can include different conditions, starting from url or cookies to the structure description, including selectors for existing elements. State conditions are similar to element selectors. There can be different conditions or groups of conditions describing

the current page. These descriptions could be ambiguous, and therefore they could be applied to different pages. The description specificity grows quantitatively, meaning the more conditions are describing the state of the page, the more likely this description is going to be unique for that page. The specificity of the individual conditions also matters.

Workflow Pair A workflow pair is a JavaScript object consisting of the *where* and the *what* parts, see Figure 3.5.

The *where* part is an object describing conditions that a page state needs to meet to execute the *what* part. The pair can be matched from *where conditions* by the interpreting algorithm. The workflow pairs are generated automatically using the current URL and selectors necessary for execution of the *what conditions* as the *where* part of the pair. The *what* part of the pair is an array of objects representing user's actions and the necessary arguments for them. The actions are mapped to the corresponding Playwright page methods. There are several actions that can be executed on the page by the Playwright. The complete list of these actions is available in the documentation[6].

The actions that are supported by the recorder automatically as *what conditions* are *click*, *goBack*, *goForward*, *goto* and *press*.

```
{
"where": {
  "url": "https://www.google.com/",
  "selectors": [
    "[name=\"q\"]"
  ]
},
"what": [
  {
    "action": "click",
    "args": [
      "[name=\"q\"]"
    ]
  },
  {
    "action": "waitForLoadState",
    "args": [
      "networkidle"
    ]
}
```

Figure 3.5: Generated workflow pair

The example from Figure 3.5 shows, how could an automatically generated workflow pair look. This pair in matched by the interpreting algorithm when the page reaches the state identified by *"https://www.google.com/"* URL navigation

[6]https://playwright.dev/docs/api/class-page

and the HTML element satisfying the selector *"[name="q"]"* is present. After the successful match the interpretation continues by clicking on the element and waiting until it reaches a *"networkidle"* load state. More about the WAW format is mentioned in the WAW format part.

Generation Post-processing The *Generator* adds *waitForLoadState* action after every *what condition* except the *press* action. Waiting for *"networkidle"* load state ensures that the current action has enough time to change the page state before the next action is executed. The *press* action does not require this pause.

Each pair is validated by the two main criteria before it is added into the workflow. First check focuses on the *where conditions*. If a match, usually URL and the whole group of selectors, is found in the workflow, the *what conditions* are concatenated with the *what conditions* of the matched pair. Otherwise, only one pair would be matched during the interpretation.

The second validation criteria inspects pairs with the matching URL *where condition*. A test for the new *where* selectors is then made. Even though the selectors are different, they can point to the same element. If that is the case, the new *what conditions* are concatenated with the previously generated pair's *what conditions* and the selectors are added to the *where conditions*.

Else the selectors can point to a different element, but both elements can be visible on the page at the same time. Because the order in which pairs are organized matters during the interpretation, the earlier recorded pair would get over-shadowed by the later one. Even now, the solution is to concatenate their *what conditions* and to add all the selectors from the over-shadowing pair to the *where conditions* of the over-shadowed pair. The execution part now includes multiple elements.

A problem can occur, if the earlier pair was from a significantly further place in the workflow. The recording would probably not meet the user's expectations, therefore it is a good practice to notify the user in this case and let him resolve the problem.

After the recording is finished optimization takes place. The workflow is analyzed and sequences of *press* actions are transformed into only one action, using the Playwright *type* method. This algorithm takes only letters, numbers and space characters into consideration. Special keyboard characters, like *Enter* or *Arrow down*, except the *Backspace* character, are omitted and left as *press* actions inside of the workflow. Each *Backspace* character situated inside of the *press* action sequence is executed on the resulting string, making the input valid.

Custom Actions The recorder offers a possibility to generate custom action pairs. Custom actions are special methods, that cannot be performed by interaction with the simulated browser. Supported custom actions are *click on coordinates*, *enqueue links*, *scrape*, *scrape schema*, *screenshot*, *scroll* and *script*. The meaning behind these actions is described in the WAW format documentation[7].

The recorder provides a straightforward custom action settings editor on the right side panel, see Figure 3.6. The conditions of custom actions' pairs are constructed mainly from previously generated data and user decisions.

[7]https://github.com/apify/waw-file-specification

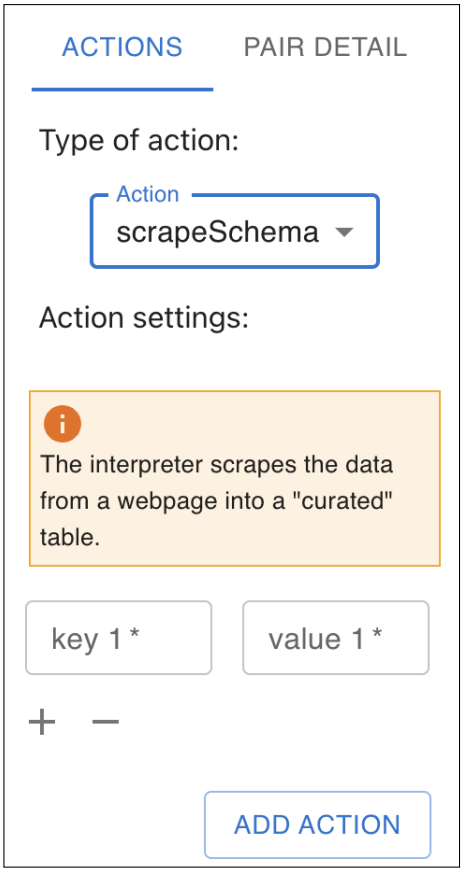

Figure 3.6: Custom action settings editor

3.3 Recording Interpretation

The recorder allows users to execute, stop and pause the active recording workflow. This process is generally denoted as the interpretation. The interpretation is performed by the WBR Interpreter. The recording interpretation is controlled on the back-end by the following API endpoints.

- `GET /record/interpret`
- `POST /record/interpret/stop`

Every *RemoteBrowser* instance has its own *Interpreter* class instance. The *Interpreter* class persists the current interpretation states, such as when the execution is paused, how to resume it, log messages and execution output data, including binary images or extracted JavaScript Object Notation (JSON) data. It also registers multiple events from the client and emits information about the execution. The main entry point for controlling the execution is located inside the *RemoteBrowser* class.

On the client-side, the interpretation is achieved by a group of controlled buttons, meaning that the label and the fact that a button is clickable are changed according to their usage, see Figure 3.7.

When interpretation is started a new page with the initial state is initialized. This state then changes until the interpretation finishes. If the interpretation is

Figure 3.7: Interpretation buttons

stopped a new page is also initialized because the *Interpreter* causes the active page to close.

Settings *Interpreter* settings are available in the settings tab on the left side panel. If they are changed, the *Interpreter* will start the next interpretation with those arguments. Therefore *maxConcurrency*, *maxRepeats* and *debug* options[8] are available. By default, the interpretation starts with *maxConcurrency* set to 1, *maxRepeats* set to 1 and *debug* set to false.

Parameters As recordings can have parameters, the user can change their value in the settings tab on the left side panel as well. Parameters are required for successful interpretation. If the user does not specify the parameters, they default to an empty string.

3.3.1 Interpretation Pause

The interpretation can be paused by two different actions, clicking the *pause* button or assigning a breakpoint to a pair. Throughout the workflow preview, many breakpoints can be assigned. The interpretation will pause every time it matches the breakpoint assigned pair. It will not execute the *what* actions of that pair before the interpretation is resumed.

After the interpretation starts, the client is highlighting the currently matched pair in the workflow preview. The user can watch and wait until the pair he wants to pause is highlighted. He can also decide to pause when a specific action is performed in the browser window.

The technique of pausing is possible due to the *flag* Interpreter action. This action executes a callback which listens to the user interactions on the front-end. The callback also provides the function for resuming the interpretation.

The pausing is achieved by adding this *flag* action as a first *what condition* to every pair inside of the workflow before the interpretation starts. The generated flag actions are not reflected anyhow in the recording.

Resuming Options After the interpretation pauses a user can resume it by the *resume* button. There is also the *step* button available. A step means resuming execution of the whole *what pair* part and trying to match another pair. If the match is defined, the interpretation will pause again, else the interpretation successfully finishes.

[8] https://github.com/barjin/wbr/blob/main/docs/wbr-interpret/interpreter.md

3.3.2 Interpretation Log

The interpretation log is displaying the execution progress messages, debug and other outputs, see Figure 3.8. It can be opened or closed by expanding the interpretation log component at the bottom of the application. The log is necessary for tracking the progress and debugging recordings. All log related communication is done through the web socket connection.

The front-end implementation scrolls down with new messages to mimic the terminal.

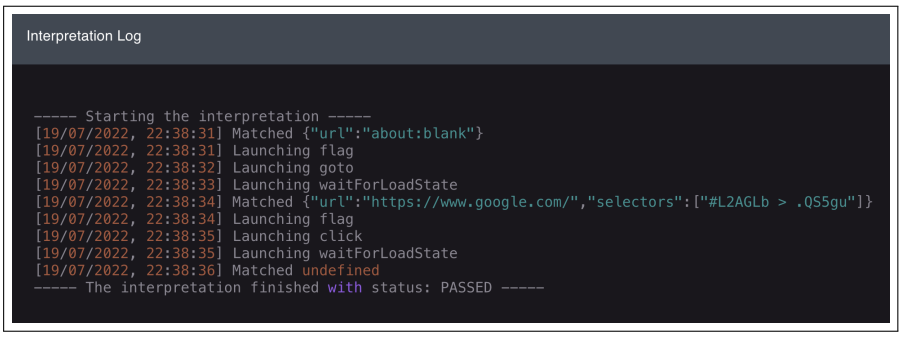

Figure 3.8: Interpretation log example

3.4 Recording Modification

One of the main features of the recorder is the possibility to edit a recording. This can be achieved in two ways: either editing the recording manually or automatically.

3.4.1 Automated Edit

Automated editing is performed by interpreting the recording. The workflow can be interpreted either fully, to continue recording from the end, or it can be paused at some specific part. This will take the browser to the desired state and when the interpretation finishes or pauses, the user can continue to record by interacting with the simulated browser or by using other methods. These pairs will be generated and added to the correct part of the recorded workflow.

When the recording is edited automatically by pausing the interpretation, it is called the block recording method. By using this method, the user can record different paths that the Interpreter can choose during the interpretation process.

The following example demonstrates why is this useful. Consider having a recording, which navigates to a specific website and performs a search there. This recording stops working because the website implemented new cookie consent banner, which makes the search in the background unreachable. We can go back to the recording session and pause the interpretation before the search is executed. The banner is visible in the browser window. We continue by recording a new path by clicking the banner's *agree* button. When the recording is run after this change, the Interpreter will make the recording succeed in both cases, when the cookie consent banner is present or when it is not.

3.4.2 Manual Edit

For updating, deleting and inserting pairs in the workflow manually from the client, there are endpoints available in the workflow route. These endpoints take the index of the pair inside of the recorded workflow as a parameter. When updating or inserting a new pair, its definition is expected in the request's body.

- `PUT /workflow/pair/:index`
- `POST /workflow/pair/:index`
- `DELETE /workflow/pair/:index`

Furthermore, the user can manually edit the recording in the UI by using the raw form editor or the pair detail edit option.

Raw Editor The raw editor provides a simple pair edit modal window, see Figure 3.9. The user can reorder the pair to a different index, set an id or edit the JSON string of the *where* or the *what pair* part directly. On save, the editor provides validation of the pair and can output specific error messages. These messages act as helpful insight for the user.

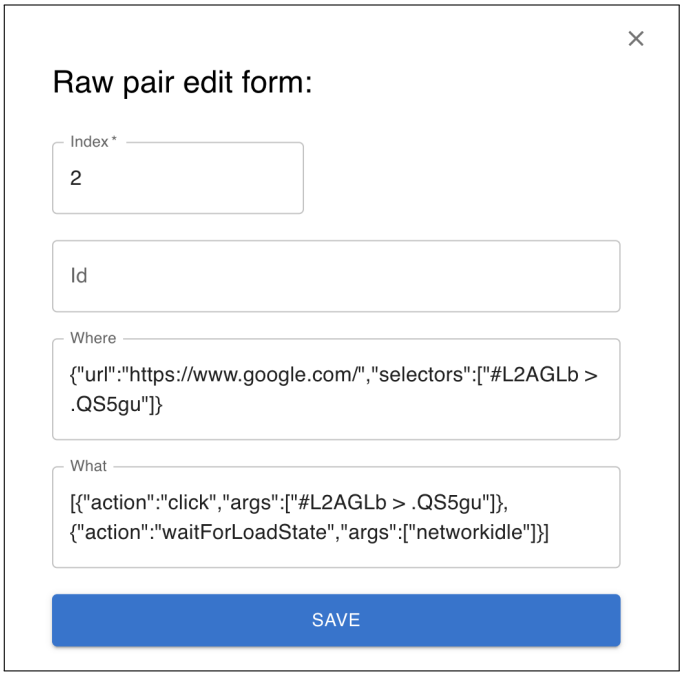

Figure 3.9: Raw editor example

Pair Detail Edit The right side panel provides a pair detail view, see Figure 3.10. The pair needs to be selected first by clicking on the *index* button in the recording preview, see Figure 3.3.

The pair detail component is generated recursively and dynamically. Rules and components defined for displaying various types of input are used. Objects are shown in a tree view, items can be added or removed from arrays and text

areas are smart, parsing numbers or objects if needed. A whole *what condition* can be deleted by the *X* button. On the other hand, *where conditions* can be only changed or added. Both, *where* and *what pair* parts are collapsible for well-arranged look. Therefore, the pair detail can be viewed as an intelligent "edit and display" helper for the workflow pairs.

If a value is changed, React updates every component involved in a cascading wave. The change also propagates to the recording preview, see Figure 3.3. Web sockets update the workflow inside the *Generator* on the back-end.

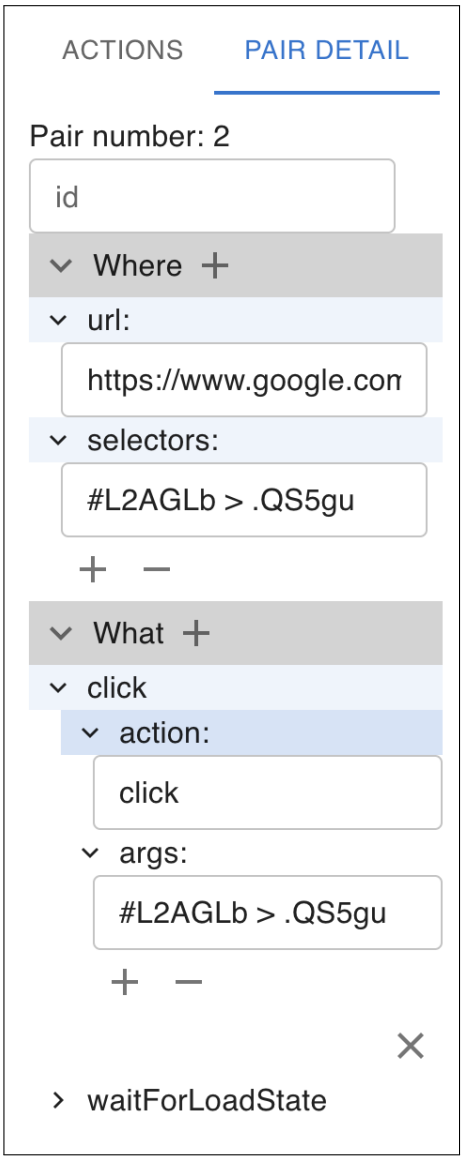

Figure 3.10: Pair detail edit example

The pair detail editor offers to add a new *where or what condition*. This addition triggers a helpful modal, tailored for every possible option according to the WAW format specification, see section 1.3.1. Examples of these modals are shown by Figure 3.11 and Figure 3.12.

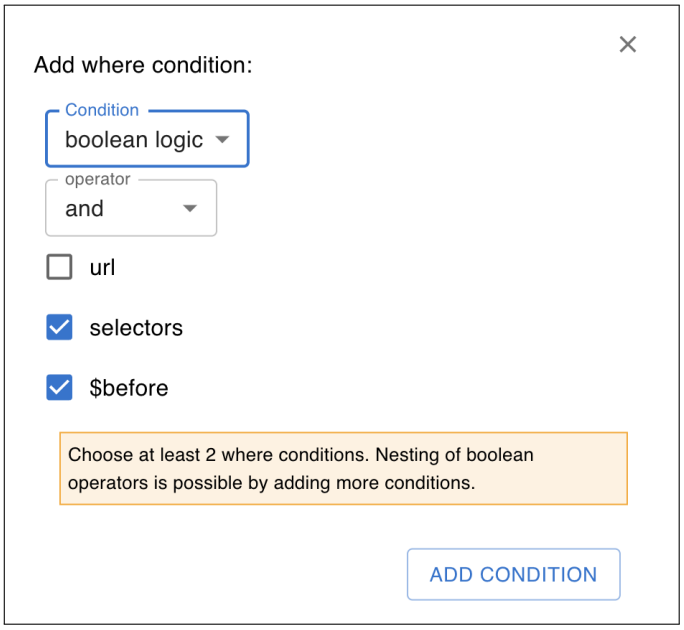

Figure 3.11: Add *where condition* example

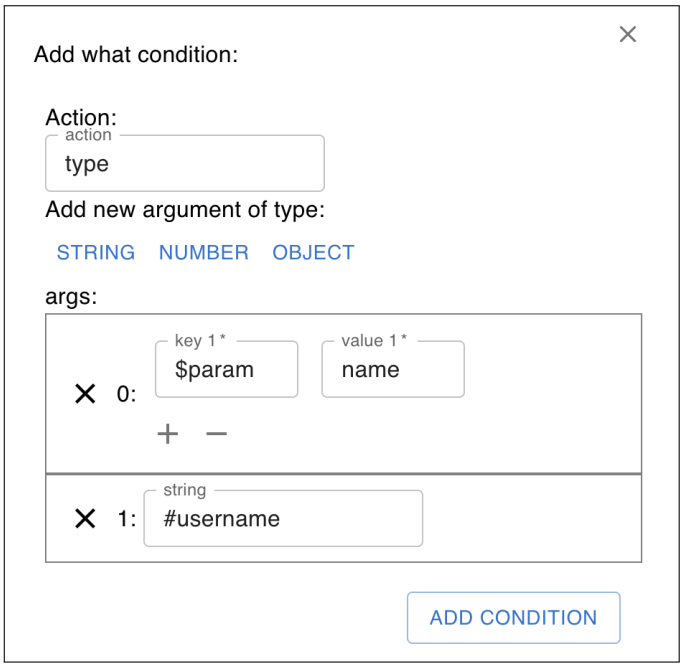

Figure 3.12: Add *what condition* example

4. Recordings Management View

This chapter describes the management section of this project in detail. User must be able to manage his recordings as well as interpretations of the recordings, also called runs. Recorder's main page serves this purpose.

The initial page points to the recordings table with a left sidebar menu, see Figure 4.1. The runs table is accessible from that menu. The top navigation bar is used to start a new recording session.

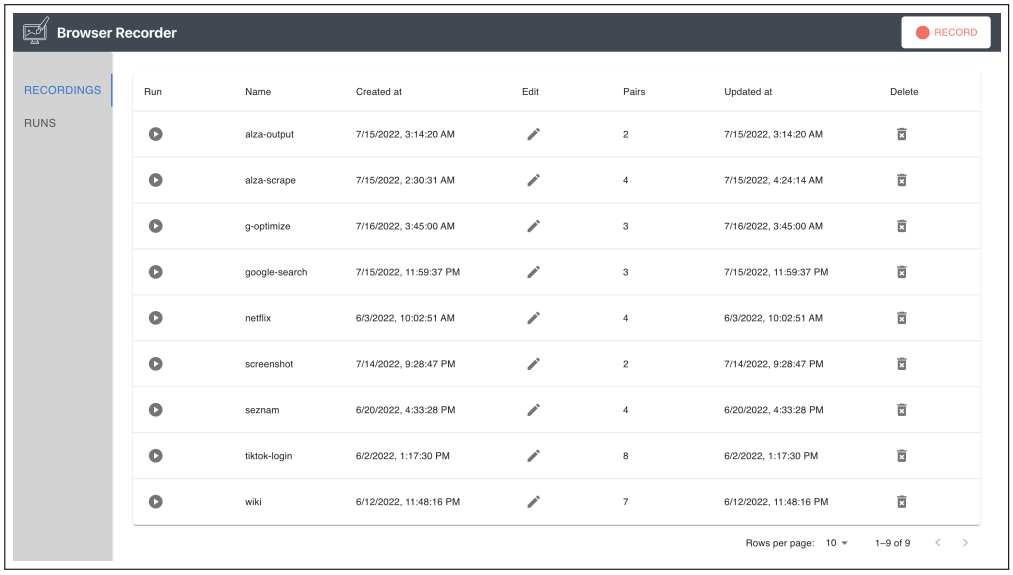

Figure 4.1: Recordings table view

4.1 Recordings Table

The recordings table offers an overview of the recorded workflows. Recordings are saved and read from *./../storage/recordings* folder. They are stored together with metadata, see Figure 4.2. The recordings metadata contains an array of parameters which is used when the recording is executed to achieve a correct interpretation.

```
"recording_meta": {
    "name": "alza-output",
    "create_date": "7/15/2022, 3:14:20 AM",
    "pairs": 2,
    "update_date": "7/15/2022, 3:14:20 AM",
    "params": []
}
```

Figure 4.2: Recording metadata example

Moreover, the table acts as a means for recording manipulation. A recording can be executed, edited and deleted from the table. The execution opens

the *Interpreter settings and recording parameters* modal. An edit starts a new recording session with a loaded recording workflow.

Server storage API provides endpoints for obtaining and deleting recordings.

- `GET /storage/recordings`
- `DELETE /storage/recordings/:recordingName`

4.2 Recording interpretation

Running a recording requires two API calls.

1. `PUT /storage/runs/:recordingName`
2. `POST /storage/runs/run/:runName/:runId`

The first request creates a *RemoteBrowser* instance with a dedicated socket connection for the interpretation, generates *runId* and saves created run's metadata in the file named *recordingName_runId* to the *./../storage/runs* folder. This way a new run entry is created for every interpretation. This entry is storing the Interpreter settings, filled parameters, interpretation log, run's metadata and output extracted during the run. Serializable and binary outputs are stored in a JSON format, see Figure 4.3. The second request controls the execution and stores its result to the run's file in the storage.

After initiating a recording run on the client, the view of the page changes automatically to the runs table, displaying contents of the collapsible row which is currently running.

4.2.1 Runs Table

The runs table offers an overview of the executed recordings. Every interpretation creates a new record in the table. The records are displayed as collapsible rows with hidden content described in Collapsible row contents section, see Figure 4.4. The runs are saved and read from *./../storage/runs* folder. A run's file content has a form of metadata in the JSON format, see Figure 4.3.

A run duration is computed after the interpretation finishes and it is transformed into a friendly format. When a task field contains a *task* label it means that the recording's parameter array is not empty and the inputted parameter values could be found in the record's hidden content. This method is not intended to be used with passwords and private data, and therefore a good candidate for improvement.

Server storage API provides endpoints for obtaining and deleting runs. The run's name in the second endpoint refers to *recordingName_runId* format.

1. `GET /storage/runs`
2. `DELETE /storage/runs/:runName`

```
{
  "status": "PASSED",
  "name": "alza-output",
  "startedAt": "7/15/2022, 4:27:14 AM",
  "finishedAt": "7/15/2022, 4:27:21 AM",
  "duration": "7 s",
  "task": "",
  "browserId": null,
  "interpreterSettings": {
    "maxConcurrency": 1,
    "maxRepeats": 1,
    "debug": false
  },
  "log": "...",
  "runId": "945e7678-0c10-47c0-aaba-2b00bafc1b6a",
  "serializableOutput": {
    "item-0": "...",
    ...
  },
  "binaryOutput": {
    "item-0": {...},
    "item-1": {...}
  }
}
```


Figure 4.3: Run meta data example

	Status	Name	Started at	Finished at	Duration	Run id	Task	Delete
⌄	PASSED	alza-output	7/15/2022, 3:19:21 AM	7/15/2022, 3:19:29 AM	8 s	6005e098-b0c6-488c-94ef-cd78e9f8f385		
⌄	PASSED	alza-output	7/15/2022, 4:27:14 AM	7/15/2022, 4:27:21 AM	7 s	945e7678-0c10-47c0-aaba-2b00bafc1b6a		
⌄	PASSED	alza-scrape	7/15/2022, 2:31:33 AM	7/15/2022, 2:31:40 AM	7 s	28785bc6-f918-4eec-abb3-255e1da201e1		
⌄	PASSED	cockatiel	7/15/2022, 4:29:21 AM	7/15/2022, 4:29:34 AM	13 s	1a76b8aa-04d3-41de-8ce6-588bd5e95968	task	
⌄	PASSED	cockatiel	7/15/2022, 4:27:49 AM	7/15/2022, 4:28:01 AM	12 s	a684b626-e458-4f65-9e85-124b4efab7f3	task	
⌄	STOPPED	cockatiel	7/15/2022, 3:34:30 AM	7/15/2022, 3:34:37 AM	7 s	c309c397-2d3c-4f02-9d7f-f9caf20a12bb	task	

Figure 4.4: Runs table

Collapsible row contents A collapsible row component consist of the interpretation log, Interpreter settings, inputted parameter values and output data preview, see Figure 4.5.

While the run is in progress an *abort* button is available to stop the interpretation, situated under the log's window. Log drags down, trying to mimic a terminal, with every new message the same way as Interpretation Log does.

After the run finishes with one of the possible statuses PASSED, FAILED or STOPPED, the *abort* button disappears and log data are loaded from the storage.

The output preview displays the serializable and binary data extracted while running the recording. The data are shown in the correct format, see Figure 4.5. For example, binary data with *image/png* mime type will appear inside an *img* tag, whereas JSON data will be parsed for a pretty print inside a *textarea* element. A link for downloading the data is provided as well and is located above the preview of each output item.

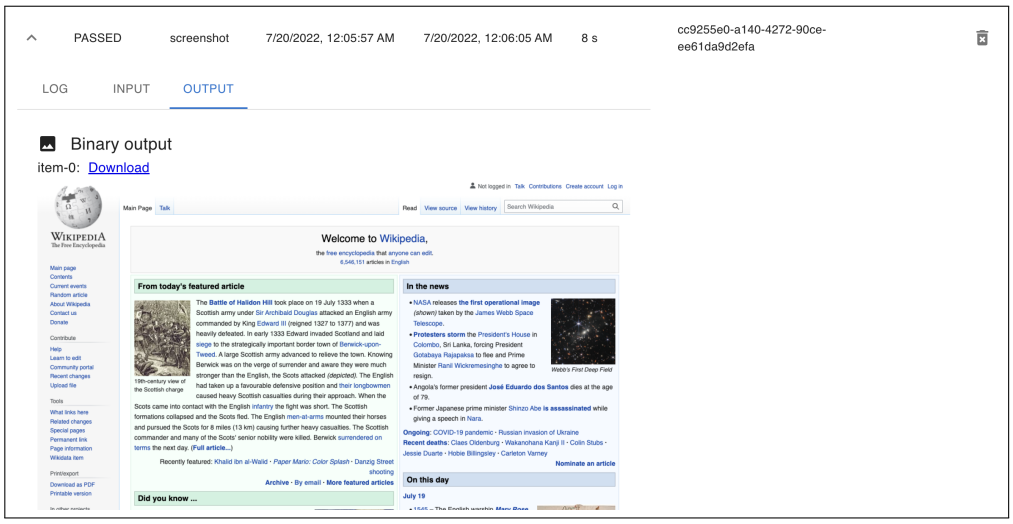

Figure 4.5: Run's output preview example

Conclusion

The web browser recorder application was developed according to requests and cooperation with the Apify company. Apify specilizes in web automation technologies and data extraction on the Internet. Apart from this project, a WBR library was created at Apify. For more details see section 1.3. The recorder was intended as a user-friendly UI, operating on top of the library's interpretation algorithm and WAW format specification, see chapter 1. Furthermore, it handles the element selectors and valid WAW data generation.

As mentioned in the Introduction, the recorder strives to completely remove the necessity for writing code when automating processes on the web. It is able to create web automation workflows, called recordings. The ability to execute these recording in an intelligent nonlinear way is the key for making web automation more efficient and error proof. The unique characteristics of WAW format and the Interpreter from the WBR library are very useful for that goal.

From the UX perspective having a server-client application is advantageous. The reasons are described in the chapter 2. Another advantage is the interchangeability of different software parts. The GUI could be easily replaced by a different implementation still profiting from the back-end functionality. The interpretation algorithm can be changed allowing further experimenting and finding the best algorithm for recording interpretation.

This project is a great starting point. From here, the recorder can be improved while still maintaining a balanced and user-friendly interface. Let us discuss some of the ideas for improvement. First of all, the solution is not optimized for inputting a secret information, like passwords. Because website log in use cases are common, this feature would make a great addition.

Adding support for a wider range of interactions, such as fill, drag and drop, move mouse for triggering hover effect on an element, will be helpful. With every new interaction the solution will get more complex, but also more useful. A simple support for other actions is already integrated, but making it work fully would require an examination of how the page reacts to these interactions and finding out how this should be integrated in the generated workflow.

As to the constant need of sending a big quantity of various data over the communication channels, there is a possibility for optimization, lowering the necessary bandwidth and making the application more responsive. For example, data can be emitted in a compressed form and decompressed after they are received.

Web automation tools must reflect the increasing complexity of browser manipulation. One of the issues is browser fingerprinting. A browser fingerprint is information collected specifically by interaction with the web browser of the device. Fingerprints can be used to fully or partially identify individual users or devices. Out of the box, the recorder does not support complicated browser configuration. As a result, the simulated browser is easily detectable as automation, leading to blocking its access either completely or partly. Therefore, it would be wise to add support for browser fingerprint configuration.

One last interesting extension could be an integration of mobile application emulator. This would enable the UI and back-end functionality to be re-used with a few alterations for recording mobile application workflows. However, we

would need to use a different technology than Playwright as this tool-set does not support emulating mobile applications as well as design a new interpreting algorithm over such technology.

Finally, it is important to mention that the recorder application is very useful for web automation community and will be soon published on the Apify website. The idea is to integrate the recorder inside of a public actor[1], taking the advantage of the user management and consumption cost computation systems, which are already available on the platform, making anyone able to use the web browser recorder.

[1]https://apify.com/store

Bibliography

[1] Angelo Gargantini and Elvinia Riccobene. Asm-based testing: Coverage criteria and automatic test sequence generation. *Journal of Universal Computer Science*, 7(11):1050–1067, 2001.

[2] Judit Bar-Ilan. Data collection methods on the web for infometric purposes—a review and analysis. *Scientometrics*, 50(1):7–32, 2001.

[3] Stephen J Mooney, Daniel J Westreich, and Abdulrahman M El-Sayed. Epidemiology in the era of big data. *Epidemiology (Cambridge, Mass.)*, 26(3):390, 2015.

[4] De S Sirisuriya et al. A comparative study on web scraping. 2015.

[5] David Flanagan. *JavaScript: the definitive guide*, volume 1018. O'reilly, 2006.

[6] Douglas Crockford. *JavaScript: The Good Parts: The Good Parts.* " O'Reilly Media, Inc.", 2008.

[7] W3Techs. Usage statistics of JavaScript as client-side programming language on websites. `https://w3techs.com/technologies/overview/javascript_library/`, 2022. [Online; accessed July-2022].

[8] W3Techs. Usage statistics of JavaScript libraries for websites. `https://w3techs.com/technologies/overview/javascript_library`, 2022. [Online; accessed July-2022].

[9] Sebastian Peyrott. A Brief History of JavaScript. `https://auth0.com/blog/a-brief-history-of-javascript/`, 2017. [Online; accessed July-2022].

[10] PC Games Hardware. Big browser comparison test: Internet Explorer vs. Firefox, Opera, Safari and Chrome. `https://www.pcgameshardware.de/Tools-Software-156186/Tests/Big-browser-comparison-test-Internet-Explorer-vs-Firefox/Opera-Safari-and-Chrome-Update-Firefox-35-Final-687738/`, 2009. [Online; accessed July-2022].

[11] Lauren Orsini. What You Need To Know About Node.js. `https://readwrite.com/what-you-need-to-know-about-nodejs/`, 2013. [Online; accessed July-2022].

[12] Boris Cherny. *Programming TypeScript: making your JavaScript applications scale*. O'Reilly Media, 2019.

[13] Shama Hoque. *Full-Stack React Projects: Learn MERN stack development by building modern web apps using MongoDB, Express, React, and Node. js.* Packt Publishing Ltd, 2020.

[14] React Conf. React today and tomorrow and 90https://www.youtube.com/watch?v=dpw9EHDh2bM&ab_channel=ReactConf. [Online; accessed July-2022].

[15] Roman Nurik. Material design in the 2014 Google I/O app. https://android-developers.googleblog.com/2014/08/material-design-in-2014-google-io-app.html, 2014. [Online; accessed July-2022].

[16] Andrew J. Davison. A brief history of web app automation. https://capiche.com/e/software-automation-history, 2020. [Online; accessed July-2022].

[17] Jindřich Bär. WAW documentation. https://github.com/apify/waw-file-specification, 2022. [Online; accessed July-2022].

[18] Jindřich Bär. WBR documentation. https://github.com/barjin/wbr/tree/main/docs, 2022. [Online; accessed July-2022].

[19] TypeScript Docs. React. https://www.typescriptlang.org/docs/handbook/react.html, 2022. [Online; accessed July-2022].

[20] Trends Built With. Express Usage Statistics. https://trends.builtwith.com/framework/Express, 2022. [Online; accessed July-2022].

[21] Casimir Saternos. *Client-Server Web Apps with JavaScript and Java: Rich, Scalable, and RESTful.* " O'Reilly Media, Inc.", 2014.

[22] Socket.IO developers. Socket.IO - Introduction. https://socket.io/docs/v4/, 2022. [Online; accessed July-2022].

List of Figures

List of Abbreviations

HTML Hypertext Markup Language

DOM Document Object Model

API Application Programming Interface

HTTP Hypertext Transfer Protocol

GUI Graphical User Interface

UI User Interface

RPA Robotic Process Automation

WAW Web Automation Workflow

UX User Experience

CSS Cascading Style Sheets

WBR Web Browser Robot

JSON JavaScript Object Notation

JSX JavaScript Extensible Markup Language

TSX TypeScript Extensible Markup Language

MUI Material User Interface

CRA Create React App

REST Representational State Transfer

TCP Transmission Control Protocol

AST Abstract Syntax Tree

JPEG Joint Photographic Experts Group

CDP Chrome DevTools Protocol

URL Uniform Resource Locator

A. Attachments

A.1 Project Repository

Content and structure of project repository attached to this thesis:

- *Project*
  - *config-overrides.json* — A *react-app-rewired* configuration
  - *package.json* — Holds important information and records dependencies
  - *README.md* — Repository's readme file.
  - *tsconfig.json* — Configures TypeScript compiler options.
  - *docs* — The generated server documentation.
    - *index.html* — Documentation entry point.
  - *examples* — The examples folder.
    - *recordings* — The examples of recorded workflows.
    - *runs* — An example of run file.
  - *public* — Contains static files such as index.html, javascript library files, images, and other assets
    - *img* — Images folder.
    - *index.html*
  - *server* — Back-end implementation.
    - *src* — Contains all back-end source code files.
  - *src* — Contains all front-end source code files.