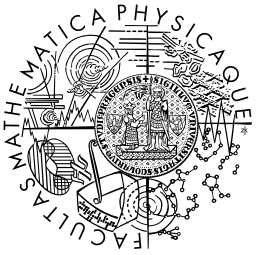

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Jindřich Bär

Declarative Web Automation Toolkit

Department of Software Engineering

Supervisor of the bachelor thesis: RNDr. Jakub Klímek, Ph.D.
Study programme: Computer Science
Study branch: Databases and Web

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

I would like to thank my supervisor RNDr. Jakub Klímek, Ph.D., who has been a great help in the development of this thesis. Special thanks go to my colleagues at *Apify* for their support and insigthful advice.

Title: Declarative Web Automation Toolkit

Author: Jindřich Bär

Department: Department of Software Engineering

Supervisor: RNDr. Jakub Klímek, Ph.D., Department of Software Engineering

Abstract: The goal of this thesis is to develop a declarative toolkit for developing web automations. Despite the great number of web automation tools and libraries on the market, it is difficult to find one powerful enough to meet the needs of complicated web automation use cases, yet simple enough to be used by untrained users. In this thesis, we research existing web automation tools, compare them based on their features and ease of use, and then develop our own text format for defining web automations. Following this, we also develop an interpreter and a validator for this format and a design and implement a GUI tool for creating and editing web automations in this format. The user testing in the last part of the thesis describes problems the users have encountered while using the tool. In the conclusion we try to come up with solutions to those problems and suggest ideas for further development.

Keywords: web, automation, scraper, crawler, declarative programming

Contents

Introduction

In the past few years, the web scraping and data extraction industry became much more prominent, as the need for data rises among all branches of science. The industry is expected to grow at 13.1% CAGR, reaching a market value of USD 948.60 Million [MaResFut20].

With web browser automation also being the leading technology for UI testing and robotic process automation (RPA) for web, the technology exceeds the data extraction needs by far. Despite the immense size of this evergrowing industry, there is still no standardized universal format for storing automated workflows. Most automation developers produce executable code in general purpose programming languages, with *Java*, *JavaScript*, and *Python* being the most prominent ones [Applit21].

This approach poses a certain security risk, as the user of such automation needs to run untrusted code. It also creates a barrier to entry for beginners without the required programming knowledge. Furthermore, the absence of a standardized format hinders the collaboration between developers.

Thesis goals

The goal of this thesis is to develop a human-readable, declarative format for storing and creating web automations, with an interpreter of this format and a visual editor, allowing less technical users to create and maintain automations in this format.

Such format should allow for a development of resilient, reusable, and comprehensive workflow definitions. It should also be machine-readable, parsable and editable, ultimately leading to a simpler adoption of the format by developers of third-party software.

This format should also be application-oblivious, i.e. not too oriented on automating only web-related workflows. The definition of the format should allow developers to create interpreters for this format for handling different automation tasks, still maintaining the same syntax.

The presented workflow interpreter should then be able to parse, validate and execute the defined web-related workflows. It should also implement a basic programmable interface to allow other developers to use the interpreter from their own software.

The workflow editor should be able to generate valid workflow files, allowing the user to create the workflow definitions without knowing the exact internal syntax of the definition format. The editor should implement a modern, user-friendly and intuitive graphical user interface (GUI) with a steep learning curve. The ultimate goal of the editor is to shield the user from the programming part of the automation task completely, leaving them with a simple yet powerful graphical tool.

1. Analysis

When assembling a multifunctional, reusable toolkit, it is crucial to map the exact user needs. The following chapter analyses the user requirements and project use cases. It also describes different user roles and states general functional and nonfunctional requirements for the software.

1.1 User roles

In the first section of the analysis, we describe the typical users of such a toolkit. Users may have different requirements based on their level of expertise and knowledge. While the toolkit should be accessible and user-friendly enough to allow beginners to create web automations with ease, it also should provide the more experienced users with advanced functionality required for handling specific use cases.

For clarity, we describe only two user roles with a significant difference in skill and knowledge. Please note that these roles are rather exemplatory and do not describe actual users the author has met. Their main purpose is to provide a clear dichotomy between two common groups of software users.

1.1.1 User

User has a fairly basic knowledge of using personal computers and web-related technologies - knowledge of e.g. *CSS* or *XPath* selectors is expected. A *User* wants to reach their goal without much additional knowledge and/or specialized tools.

Such a user wants to use the toolkit in the most basic way. While they might have some experience with the technologies used in the toolkit, they generally do not want to use the toolkit programmatically and rely on the GUI tools only.

Their automation use case is easily described, mostly as a linear sequence of well-defined, simple steps. Some examples of such use cases might be *automated data extraction* and simple *robotic process automation*.

1.1.2 Developer

The *Developer* user role describes an intermediate-to-expert computer specialist with deep knowledge of computer systems, programming and web-related technologies. This user role expects to take advantage of the advanced features of the toolkit, possibly spending some extra time learning how to use those properly.

They might not want not only to create and run automations but also to use the toolkit programmatically, install the toolkit components on their systems or edit parts of the toolkit.

When creating an automation, the *Developer* has more complicated use cases with possibly branching scenarios. Those might be more *elaborate data extraction* cases, *software testing*, *complicated RPA* and other.

1.2 Requirements

The following section describes functional and non-functional requirements for the toolkit project, based on the requirements of the user roles described in the section 1.1 User roles.

For clarity, let us divide the toolkit project into individual tools serving different purposes. As the main purposes of the toolkit are *creating*, *editing* and *running web automations*, we can talk about the *Editor* and the *Runner* parts separately.

1.2.1 Editor

The *Editor* is the part of the toolkit allowing the users to create and edit the web automations. It should provide a user-friendly way of doing so while not restricting the more advanced users.

The goal of the *Editor* is not to exhaustively support all the features of the workflow definition syntax, but to provide a simple and intuitive way of creating and editing web automations.

Functional Requirements

1.2.1.1.1 The *Editor* must allow the user to create a valid workflow file.

1.2.1.1.2 The *Editor* must enable the user to upload a valid workflow file into the *Editor*.

1.2.1.1.3 If the uploaded file is not a valid workflow file, the *Editor* must reject it.

1.2.1.1.4 The *Editor* must allow the user to edit the workflow file. No user-induced change to the file shall corrupt the valid file syntax.

1.2.1.1.5 The *Editor* must allow the user to export a valid workflow file. This exported file must be readable by the *Runner*.

1.2.1.1.6 The *Editor* must interface the *Runner*, allowing the user to test run the automations.

1.2.1.1.7 During the test run, the *Editor* must display the automation results in a human-readable way.

Nonfunctional requirements

1.2.1.2.1 The user interface of the *Editor* shall adhere to the best user interface (UI) practices.[UIDesign]

1.2.1.2.2 The *Editor* shall contain example workflow files for the user to study and to showcase the capabilities of the toolkit.

1.2.2 Runner

The *Runner* is the part of the toolkit providing support for executing the automations made with the *Editor*. It should provide a safe and optimized way for running the automations as well as a comprehensive user interface.

Functional Requirements

1.2.2.1.1 The *Runner* must allow the user to execute given valid automations.

1.2.2.1.2 If the provided automation is not valid, the *Runner* must refuse such automation, notifying the user.

1.2.2.1.3 If the automation provided to the *Runner* is not valid, the *Runner* must provide the user with detailed information about the errors.

1.2.2.1.4 The *Runner* must allow the user to observe the automation run.

1.2.2.1.5 The *Runner* must provide the user with additional information about the automation run.

1.2.2.1.6 The *Runner* must enable the user to interrupt the automation execution at an arbitrary moment.

1.2.2.1.7 The *Runner* must inform the user of any runtime errors. Furthermore, the *Runner* must also log all errors appropriately.

1.2.2.1.8 The *Runner* must expose a programmable interface to allow for a simple third-party adoption.

Nonfunctional requirements

1.2.2.2.1 The *Runner* shall implement the automation execution in an optimized way.

1.2.2.2.2 The installation of the *Runner* shall be simple, allowing for quick adoption of the software.

1.3 Use case analysis

The following section goes through multiple use cases for the individual parts of the toolkit. The described use cases should reflect the user requirements from the section 1.2 Requirements. Every use case is also accompanied by an example scenario describing a typical user flow.

For clarity, we again divide the use cases into parts corresponding to the main features of the toolkit, much like in the previous section.

1.3.1 Editor

The *Editor* is a part of the toolkit facilitating the creation of the automation files. The following section contains sample use cases, describing the standard user flow and exception handling.

The user in all the following use cases corresponds to the *User* user role.

Use Case 1 : First steps

- **Goal**: User wants to learn how to operate the *Editor*.
- **Scenario**:
  1. User accesses the *Editor* application for the first time.
  2. A comprehensive welcome message is shown. The *Editor* application provides a step by step explanation of its interface.
  3. During this showcase, the *Editor* interface walks the user through the process of creating simple automation.
  4. After the tutorial phase, the *Editor* interface returns to the default state.
- **Note:** At any time, the user can decide to stop the tutorial and access the full version of the *Editor*. On the other hand, the user must be allowed to start the tutorial manually, even repeatedly.

Use Case 2 : Create an Automation

- **Goal**: User wants to create a new automation.
- **Scenario**:
  1. User accesses the *Editor* application.
  2. Using the *Editor* interface, the user creates a new blank automation file.
  3. User edits the newly created automation using the *Editor* interface.
  4. After editing the automation using the *Editor*, the user can export the automation. The *Editor* generates a valid automation file and presents it to the user.

Use Case 3 : Edit an existing automation

- **Goal**: User wants to edit an existing automation stored on their device.
- **Scenario**:
  1. User accesses the *Editor* application.
  2. Using the *Editor* interface, the user passes the existing automation to the *Editor*.
  3. The *Editor* validates the passed automation.
  4. User edits the uploaded automation using the *Editor* interface.
  5. After editing the automation using the *Editor*, the user can export the automation. The *Editor* generates a valid automation file and presents this to the user.
- **Exception**: The file provided by the user in step 2 is not a valid automation file.
  - **Exception flow**: The *Editor* rejects such file with a comprehensive error message. The *Editor* interface returns to the initial state. The user can pass another automation file.

Here follows the UML diagram specifying the relations between steps of the use cases and their relation to the end-user.

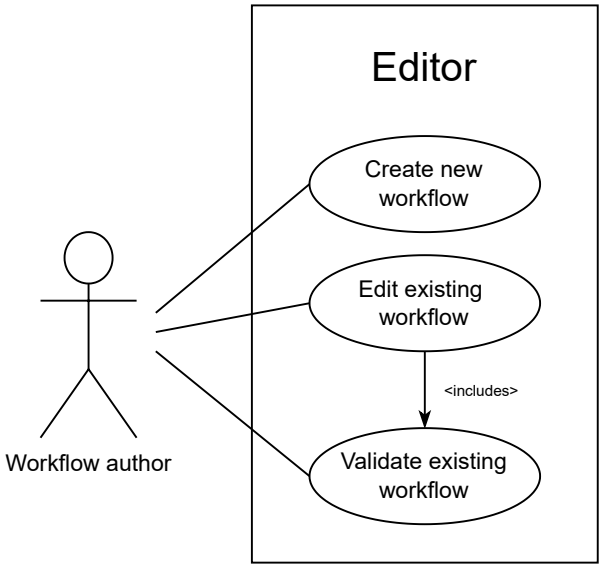

Figure 1.1: Editor - Use Case UML diagram

1.3.2 Runner

The *Runner* is the part of the toolkit responsible for the automation execution. The following section contains sample use cases, describing the standard user flow and exception handling. If not stated otherwise, the user in the following examples corresponds to the *User* user role.

Use Case 1 : Running an automation

- **Goal**: User wants to run an automation.
- **Scenario**:
  1. User accesses the *Runner* application.
  2. Using the *Runner* interface, the user passes the automation to the *Runner*.
  3. The *Runner* validates the passed automation.
  4. The *Runner* runs the automation, sharing the progress with the user.
  5. After the automation is done, the *Runner* notifies the user, eventually presenting the results of the automation run.
- **Exception**: The file provided by the user in step 2 is not a valid automation file.
  - **Exception flow**: The *Runner* rejects such file with a detailed error message. The *Runner* interface returns to the initial state. The user can pass another automation file.

Use Case 2 : Stopping the execution early

- **Goal**: After submitting the automation to the *Runner*, the user wants to stop the automation execution prematurely.
- **Scenario**:
  1. User accesses the *Runner* application.
  2. Using the *Runner* interface, the user passes the automation to the *Runner*.
  3. The *Runner* validates the passed automation.
  4. The *Runner* runs the automation, sharing the progress with the user.
  5. Using the *Runner* interface, the user orders the *Runner* to stop the execution.
  6. The *Runner* responds to the user's halt request. It stops the automation execution and exits gracefully. The *Runner* presents the user with the run results.
- **Note**: The run results (e.g. the data scraped from the websites) can be incomplete because of the early termination. Despite this, the early termination must not affect the data integrity.

Use Case 3 : Debugging an automation

- **Goal**: An advanced user (see the Developer user role) needs to gather information on their automation run performance.
- **Scenario**:
  1. User accesses the *Runner* application.
  2. Using the *Runner* interface, the user passes the automation to the *Runner*. The user also switches the debugging mode on.
  3. The *Runner* validates the passed automation.
  4. The *Runner* runs the automation, sharing the progress with the user. The *Runner* now also shares the internal debugging information with the user.
  5. After the automation is done, the *Runner* notifies the user, presenting the debugging and performance data. Eventually, it also presents the results of the automation run.

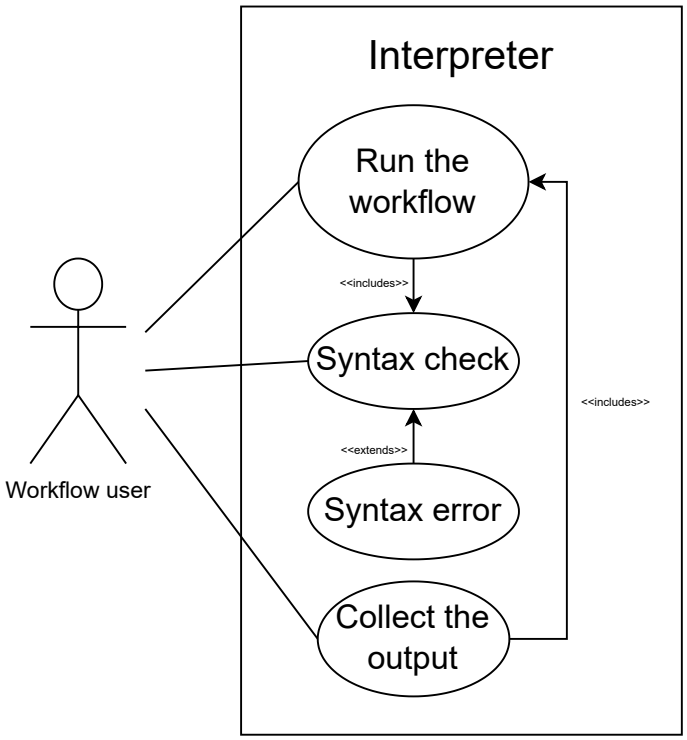

Figure 1.2: Interpreter - Use Case UML diagram

1.4 Existing solutions

As of now, there are already numerous solutions for automating web actions on the market. A majority of those uses existing web browsers and offer a programmable interface for simulating user input.

1.4.1 Evaluation criteria

To provide a comprehensive comparison, the existing solutions will be evaluated based on several common criteria.

1. **Ease of use**: User experience (UX) quality, evaluated based on different user knowledge levels.
2. **Universality**: How applicable is this solution for different web automation use cases, i.e. UI testing, web scraping, web crawling and other.
3. **Automation resilience**: Is the automation capable of dealing with unexpected situations? Does the solution allow for conditional decisions?
4. **Open format**: Whether or not does the solution publish open documentation of its internal format.

An universal, open and automation tool should provide all of those qualities.

1.4.2 Programming-based solutions

In general, the existing automation solutions can be divided into two groups, based on the UX quality and granularity of supported actions. The first group consists of programmable solutions targetted at experienced users.

Cypress is a end-to-end (E2E) Javascript testing framework containing various assertions for quality assurance (QA) testing of webpages. It supports multiple web browsers and offers its own UI and toolkit for test programming and running. Due to its strong orientation towards testing, it does not provide much methods for data extraction and crawling.

- Ease of use - Creating Cypress automations requires JavaScript programming knowledge. Cypress also contains a graphical recorder *Cypress Studio* as an experimental feature.
- Universality - As a testing framework, Cypress offers methods for web-related assertions and DOM queries. Because of its library nature, writing crawlers and scrapers is also possible, despite having very little support from the library's side.
- Automation resilience - Since Cypress has been designed as a testing framework, i.e. to be used on one's own infrastructure, there is no inbuilt support for handling unexpected states. The official documentation speaks about modifying the (tested) server itself and explains the usual ways of handling exceptions in JavaScript. [CyprCond]
- Open format - The automations written in Cypress are JS programs.

Selenium WebDriver is a fairly popular tool among web UI testers, as it offers a wide variety of selector engines and comprehensive method naming. Distributed as a multilanguage library, Selenium implements a high-level interface for controlling web browsers from code.

- Ease of use - Creating Selenium WebDriver automations requires knowledge of at least one of the following programming languages: *Ruby, Java, Python, JavaScript, C#*.

  Selenium also offers an integrated development environment (IDE) with low-code/record and playback tools. This IDE is implemented as a Firefox and Chrome extension.

- Universality - Selenium WebDriver is quite low-level, enough to handle crawling, scraping and testing use cases alike. On the other hand, there are no native methods for testing, letting the user work with external testing frameworks.

- Automation resilience - In the Selenium WebDriver library, there is no inbuilt support for handling unexpected states. The Selenium IDE allows for drag-and-drop branching, which corresponds to writing conditions manually. [SelenCF]

- Open format - The automations written using Selenium are executable programs. Selenium IDE allows the users to export the created workflows as code in any of the supported languages, using various popular testing frameworks (e.g. Mocha for JavaScript, JUnit for Java etc.)

Puppeteer is a low-level library used for web browser automation. Unlike Cypress and Selenium, Puppeteer oficially supports Chromium based browsers as its only backend browsers as of now (May 12, 2022).

The communication with the browser is implemented via WebSockets and the Chrome DevTools Protocol (CDP), a Chromium-specific set of commands. This allows Puppeteer to exceed Selenium both in stability and performance, sporting up to 17% speedup in benchmarks [Chck21].

- Ease of use - Using Puppeteer for creating web automations requires knowledge of JavaScript.

- Universality - Just like Selenium WebDriver, Puppeteer provides a low-level browser control, which makes it an universal tool for testing, scraping and crawling web.

  On the other hand, users requiring high-level abstractions for testing or data extraction must resort to using third-party libraries or write this functionality themselves.

- Automation resilience - As stated before, Puppeteer is a low-level library for web browser automation. All conditional branching must be handled by the programmer themselves.

- Open format - The automations written using Puppeteer are executable JavaScript programs.

Playwright is another low-level library multilanguage library offering programmable ways of controlling a web browser. For browser communication, Playwright uses similar technology as Puppeteer unlike Puppeteer, Playwright supports multiple commercial browsers (Chromium, Firefox, Webkit as of May 12, 2022) and has official bindings for multiple languages (Type/JavaScript, Java, Python, .NET).

Due to differences between browsers and partial incompatibility of remote debugging protocols, Playwright is distributed with patched versions of Firefox and Webkit [PPatch21]. Stock versions of Chromium-based browsers (Google Chrome, Microsoft Edge) are supported. [PReadme22]

- Ease of use - Using Puppeteer for creating web automations requires knowledge of the selected programming language.
- Universality - Following Puppeteer's legacy, Playwright also provides a low-level browser control. This makes it a universal tool for various web automation related tasks.

  Just like Puppeteer, Playwright also lacks content-oriented high-level abstractions for scraping or crawling the web. Again, those must be implemented by the user or imported from third-party libraries.

  Unlike Puppeteer, TS/JS version of Playwright comes with Playwright Test, a simple test-runner using a Jest-inspired syntax.
- Automation resilience - The Playwright library does not provide decision-making algorithms capable of handling unexpected states.
- Open format - The automations written using Playwright are executable programs.

1.4.3 Codeless solutions

All the aforementioned examples require programming, which can mean a significant barrier to entry for beginners. Besides these examples, there are also other solutions, allowing the users to create, manage and execute automated workflows using higher-level UI actions.

Dexi.io is one of such services. Accessible as a web application, it provides the user with a web-based browser recorder, removing both the need for programming and package installation. Dexi.io GUI editor allows for creating branches based on user-specified conditions.

The recorder app suffers from problems stemming from its web nature, namely CORS-related issues, being targetted by anti-scraping measures and worse responsiveness.

- Ease of use - As mentioned above, Dexi.io provides a graphical WYSIWYG recorder, shielding the user from coding of any kind. While this recorder has several flaws - namely unintuitive GUI and problems related to anti-scraping measures employed by the websites, it still can be well useful in different use cases.

- Universality - The recorder offers many user-defined actions, including test assertions and smart data extraction with sibling detection, making Dexi.io a universal tool for testing and scraping. The recorder also allows for limited web-crawling functionality.
- Automation resilience - The recorder allows the users to manually specify custom branching conditions on certain places in the workflow.
- Open format - While the recordings are exportable in a JSON[1]-based format, the definition of this format is closed, effectively causing a vendor lock-in. Besides the recorder, Dexi.io provides a platform for scheduling and running the automations.

Browse.ai serves a similar purpose as Dexi.io. Utilizing a Chrome-only browser extension for workflow recording, Browser.ai offers arguably better UX than Dexi.io with more accurate web page representation.

The execution of Browser.ai recordings is available only through the associated web service without any export option, causing even stronger lock-in than Dexi.io.

- Ease of use - Browse.ai takes pride in making web automation as intuitive as possible. This, combined with the browser extension nature of the service, makes it the one with the best UX out of the mentioned options.

  However, as of May 12, 2022, author of this work struggled with severe performance issues when using this extension, possibly hinting at optimization problems.
- Universality - Being oriented mostly towards data extraction, the recorder provides advanced scraping techniques. Web crawling and UI testing have limited support.
- Automation resilience - There is no way of specifying conditional branches.
- Open format - There is no way of exporting the recordings. The Browse.ai website provides an environment for running the recordings, causing a vendor lock-in.

  Furthermore, the difference between the recording environment, i.e. client's browser, and the execution environment, the cloud service, causes errors. Those stem from differences between both environments and anti-bot measures employed by third-party services.

Chrome Recorder is a preview feature of the Google Chrome web browser (as of May 12, 2022). This can be seen as Google's reaction to the new emerging technologies and the first attempt to implement a native recording functionality into the browser.

- Ease of use - Embedded into the browser, the Chrome recorder provides the best performance and responsiveness of the mentioned examples.

[1] https://www.json.org/json-en.html

- Universality - Since this browser feature is targetted mainly at the QA testing community, the recorder offers detailed performance measurement features. For the same reason, data extraction methods are missing, rendering the Chrome Recorder unusable for web scraping use cases.
- Automation resilience - There is no way of specifying conditional branches.
- Open format - The created recording is exportable as a JavaScript code utilizing the Puppeteer library.

Name	Ease of use, UX	Universality	Recording resilience	Open format
Cypress, Selenium	✗ requires programming	✓	–	– source code
Puppeteer, Playwright	✗ requires programming	✓	–	– source code
Dexi.io	✓ web-based GUI recorder	–	–	✗ JSON-based closed format
Browse.ai	✓ GUI recorder (extension)	–	✗	✗ No export available
Chrome Recorder	✓ GUI recorder (browser feature)	✗	✗	– source code
This work	✓ GUI editor	✓	✓	✓ open format

2. Design

The following sections describe decisions made during the project design phase. Starting by breaking the project into individual parts, the following chapter initially describes all the parts separately, followed by definitions of their contact points.

Design decisions made here should reflect the requirements mentioned in the previous chapter. These decisions also directly influence the implementation of the project described further.

2.1 Parts of the project

As stated in the introduction of this thesis, the goal of this thesis is to develop a clear, concise format for storing web automations as well as tools for simplifying the work with the format.

Given this assignment, it is only natural to first design the automation format, as the design of the tools for editing and debugging the automation files largely depends on the format design itself. For the tools part, we can reuse the rather informal partition of the tools into *Editor* and *Runner*, following the idea from the section 1.2 Requirements, as this still describes the two principal use cases of the toolkit. The notional interface and the middle ground between those two parts (*Editor* and *Runner*) is then the workflow definition format, as both tools are designed to work with it, albeit in different ways.

The main parts of the project from now on are therefore the *Format*, *Editor* and *Runner*. The design of those three parts is discussed in the following sections separately.

2.2 Workflow definition format

As both the *Runner* and the *Editor* work directly with the files containing the workflow definitions, the first part of the project to be designed is the workflow definition format itself.

The workflow definition files should contain all the information needed to describe an arbitrary web-related workflow. The files in this format should also be parsable, human- and machine-readable and provide a simple yet powerful way of programming the web automations.

2.2.1 Programming logic

As the workflow definitions are computer programs of sorts, the first design decision needs to be what programming concepts will the file format implement. To retain the steep learning curve and user-friendliness, this programming "language" also should not be too complicated.

The trend in the current automation tools, such as IFTTT, Zapier or Huginn shows a rise in the popularity of *declarative programming*. Such languages and tools work with definitions of the desired results rather than describing the complete control flow. [Sebesta2015]

Inspired by logic programming languages such as *Prolog* - and its popular implementation SWI Prolog - the workflow definition should contain a set of *conditions* describing a possible state of the environment, connected to their respective *reactions*, describing a sequence of actions to be carried out in case the condition applies.

2.2.2 Conditions

As stated before, the workflow definition format should allow the user to specify web environment-related conditions for running the automation steps.

Such conditions can be e.g. the browser visiting a certain `url`, the current page containing certain `selectors` or the current browser session having `cookies` set to specific values.

Moreover, the format should allow the user to combine the base conditions using *boolean operators* to create more comprehensible and compact syntax.

Following through with the *Prolog* comparison, the workflow definition could look something like this:

```
% X is denoting the current state of the browser
% Y is to be unified with the next state

nextState(X, Y) :- url(X, "https://jindrich.bar"),
                   % action to be
                   % executed on
                   % https://jindrich.bar

nextState(X, Y) :- selector(X, "button"),
                   % action to be
                   % executed if the current
                   % page contains a button

nextState(X, Y) :- cookies(X, "key", "value"),
                   % action to be
                   % executed if the current
                   % browser session has the
                   % ‘key‘ cookie for the
                   % page set to ‘value‘

nextState(X, Y) :- url(X, "https://example.org"),
                   selector(X, "input"),
                   % action to be
                   % executed in case of both
                   % conditions matching
                   % (boolean AND example)
```

The conditions might also provide support for advanced functions such as wildcards or regular expressions. Those would be particularly useful e.g. for URLs for targetting a specific domain, TLDs etc.

2.2.3 Reactions

The workflow definition format should also allow the user to specify the actions to be carried out when the respective condition matches.

Those can be e.g. `click`, `goto`, `scrapeData` and similar. The actions should be chainable, allowing the user to specify a set of actions to be executed sequentially, without additional condition matching between those.

Completing the *Prolog-inspired* example from the previous section, the complete workflow definition would look like this:

```
% X is denoting the current state of the browser
% Y is to be unified with the next state

nextState(X, Y) :- url(X, "https://jindrich.bar"),
                   goto(X, Y, "https://example.org"), !.

nextState(X, Y) :- selector(X, "button"),
                   click(X, Y, "button"), !.

nextState(X, Y) :- cookies(X, "key", "value"),
                   click(X, Y, "logout"), !.

nextState(X, Y) :- url(X, "https://example.org"),
                   selector(X, "input"),
                   fill(X, "input", "hello"), !.
```

The mock implementation of the workflow definition file in *SWI-Prolog* is available as a snippet[1] in the *Prolog* online execution environment *Swish*.

Please note that in this case, the *Prolog* interpreter is actually taking the role of the workflow runner.

> **Note:** The examples above also show that the new state of the browser depends only on the preceding one.
> Such quality, also called *memorylessness*, or *Markov property*, simplifies both the runner design and the programming concept itself. It might also allow for some optimizations utilizing parallel execution.

As mentioned in the Introduction, the workflow definition format should be application oblivious, allowing other developers to use it in their own automation tools. For this reason, the action names are not part of the format definition.

[1] Available at `https://swish.swi-prolog.org/p/dwaim.pl`

2.2.4 Serialization

Finally, the workflow definition needs to be physically stored in a file. As it would be rather counterproductive to develop a custom file format for storing the conditions and reactions, the workflow definitions might be stored using a host meta-format.

Based on the hierarchical nature of both *condition-action* pairs and possibly recursive nature of the *conditions* themselves, it would be only logical to store the definitions using a hierarchical data format like *JSON*, *XML*[2] or *YAML*[3].

Comparing these formats, *JSON* comes out as the most popular [GTrends22] and most space-saving [Medium21]. While the advanced features of *XML* are invaluable when working with complex structured data, it is perhaps too complicated for storing well-defined workflow definitions.

With YAML taking first place, JSON is also a runner-up in human readability. While improving the file legibility, the indentation oriented nature of YAML makes it very prone to input errors - this problem is absent in JSON because of its bracket-oriented grammar.

For the reasons mentioned, the workflow definition format will be built upon JSON - a host format providing a simple, human-readable serialization for a structured schema of the definitions.

The JSON serialization of the workflow definition file might then look as follows:

```
[
    {
        "conditions": {
            "url": "https://example.org"
        },
        "actions": [
            {
                "action": "goto",
                "args": ["https://jindrich.bar"]
            }
        ]
    },
    {
        "conditions": {
            "selector": "input"
        },
        "actions": [
            {
                "action": "fill",
                "args": ["input", "hello"]
            }
        ]
    }
]
```

[2]https://www.w3.org/TR/2006/REC-xml11-20060816/
[3]https://yaml.org/

While being more verbose, the JSON serialization is arguably more readable to a layman than the *Prolog*-syntax pseudo implementation.

The root of the workflow definition is a *JSON* array containing multiple objects, specifying the rules. Those have two keys, `conditions` and `actions`, defining the required conditions for the web browser environment and actions to be carried out in case the conditions apply, respectively.

The `conditions` object describes a valid state of the web browser using a set of predefined keys (`url`, `selector`, `cookies` and other).

The `action` object is an array of actions to be carried out. Every action is described using its name (`click`, `goto`, `fill`...) and an additional, action-specific set of arguments.

The action arguments are stored in an array. This also applies to singleton arguments, mainly to maintain consistency and improve the machine readability of the format.

Boolean operators

As mentioned in subsection 2.2.2 Conditions, the format should offer a way of combining the defined conditions using basic boolean operators. Following another popular format based on JSON (or rather JavaScript objects), the workflow definition format might take inspiration from MongoDB query syntax [Mongo20].

In this query language, boolean operations are expressed using an object with a specific key, containing an array of operand objects.

```
{ $and: [{expression},{expression},...] }
{ $or:  [{expression},{expression},...] }
          { $not: {expression} }
```

This approach has a direct mapping to JSON syntax, which makes it very suitable for our use case.

```
...
{
    "conditions": {
        "$and": [
            { "url": "https://example.org" },
            { "selector": "input" },
            { "selector": ".green" }
        ]
    },
    ...
},
...
```

Because of the associativity of those operations, the representation of the boolean operators AND and OR can have arbitrary arity. Writing an empty `$and` or `$or` clause corresponds to an empty rule. Specifying only one subrule (using `$and`/`$or` as unary operators) defies the purpose of using those operators, as the truth value is the same as of the inner condition alone.

With this being said, the format does not specify a required operator arity.

Regular expressions

Following the same logic as with the boolean operators, the support for regular expressions requested in the subsection 2.2.2 Conditions can take inspiration from MongoDB query operators [Mongo20]. In the MongoDB syntax, regular expressions are stored as objects with predefined keys - `"$regex"` and `"$options"`:

```
{ <field>: { $regex: /pattern/, $options: '<options>' } }
{ <field>: { $regex: 'pattern', $options: '<options>' } }
{ <field>: { $regex: /pattern/<options> } }
```

Regarding the types used, only the second mentioned alternative is directly translatable to JSON, as JSON cannot contain literal regular expressions. Therefore, storing the expressions as strings is the most fluent option.

```
...
    {
        "conditions": {
            "url": { "$regex": "http://.*" }
        }
    },
...
```

2.2.5 Validation

The GitHub repository of the project contains JSON Schema[4] for validating the workflow definition, as it is implemented by the *Runner* and *Editor* modules. This slightly differs from the format definition described in the previous sections, though only by naming - the `condition` part is called `where`, the action part is called `what`. The workflow definition can be also optionally prepended by `meta` object, containing the workflow's name and description. The format documentation[5] published there also describes the format of the workflow definition file in a similar manner as the chapter before.

The rest of the documentation published in the GitHub repository describes basics of workflow creation and execution using the workflow definition format. Please note that the remaining part of the GitHub-hosted documentation is not part of this thesis and should serve as a user guide only.

[4] https://github.com/barjin/wbr/blob/main/json-schema.json
[5] https://github.com/barjin/wbr/blob/main/docs/wbr-interpret/waw_definition.md

2.3 Runner

With the workflow definition format designed, we can now design the *Runner* of this format. As mentioned in the section 1.2 Requirements, the *Runner* should be a piece of software able to read, validate and execute the workflows defined in the aforementioned format.

2.3.1 Components

To facilitate later design decisions and understanding of the software, we divide the *Runner* project into several independent parts. These parts are:

Browser: The web browser to be automated. To simplify the usage and installation of the solution, the *Runner* should be able to work with stock (i.e. unpatched) versions of browsers, allowing the users to use their standard web browsers.

As seen in the section 1.4 Existing Solutions, most commercial web browsers already provide programmable interfaces (via CDP, RDP...). The *Runner* - browser communication can be further simplified by using low-level third-party libraries, also mentioned in the section 1.4 Existing Solutions.

Workflow validator: A piece of software able to statically validate a given workflow definition file. While it might provide various ways of validating the workflow definitions, the minimum is syntax validation, i.e. reading a file written in the format described above and telling whether it follows the definition of the format. Optionally, the syntax validator might also provide descriptive error messages to communicate the problem to the user.

Given the programmable nature of the format, static "code" analysis might also take place here. While some workflow definition files might be syntactically correct, it is possible that they might contain logical errors. The validator could then spot unreachable branches, suggesting reordering of the rules in the definition or suggesting updating the conditions.

Workflow interpreter: A piece of software comparing the current browser state with the conditions from the workflow definition, selecting the correct rule to be applied. Furthermore, the *Workflow interpreter* should also send the correct actions to the browser and ensure their execution went well. In accordance with the requirement *1.2.2.1.7*, the interpreter should inform the user in case of any exceptions.

2.3.2 Programming language, libraries

As mentioned above, the communication with the internal browser can be facilitated using a third-party library. This approach - compared to communicating with the web browser directly - leads to quicker development iteration and less cluttered code base, ultimately leading to a better tested software.

Looking at the competition analysis, the low-level automation libraries could be useful for this use case. Both *Puppeteer* and *Playwright* offer a lightweight

programmable interface by simply wrapping and unifying the debugging functionality of the web browsers (CDP and alternatives).

As mentioned before, *Puppeteer's* provides official support only for Chromium-based browsers, while *Playwright* provides support for Chromium-, Firefox- and Webkit-based browsers alike. For these mentioned reasons, the *Workflow interpreter* will be using *Playwright* as its backend library.

While *Playwright* has bindings for different languages (*JavaScript*, *Python*, *.NET* and *Java* as of May 12, 2022), the primary development is made in TypeScript (superset of JavaScript).

Given these facts, it would be beneficial to develop the *Runner* also in Typescript. This makes sense both because of the library support and the closeness of the language to the web environment - Typescript can be statically transpilled into Javascript, a popular client-side web programming language.

The utilization of Typescript also ensures type safety and better IDE support compared to regular JavaScript code. This simplifies the development of the tool as well as the third-party adoption of the tools.

2.4 Editor

As mentioned before, the *Editor* should be a piece of software facilitating the process of creating and editing a web automation file.

While we designed the workflow definition format to be as readable and concise as possible, the strict JSON syntax makes the format hard to write manually. Especially less technical users - such as the model user role *User* - could experience difficulties when trying to edit larger automation files. As stated in the Requirements, the *Editor* should also provide performance-oriented features, targetting more experienced users - such as the model *Developer*.

2.4.1 Technologies

In accordance with the nonfunctional requirements for the *Editor*, the *Editor* app should be user friendly and easy to use. Due to its relatively lightweight nature, it is possible to implement the *Editor* as a web application.

This approach would be beneficial for several reasons - removing the need for installation, providing a cross-platform solution and speeding up the development and debugging process, to name a few. Implementing the *Editor* as a client-side web application in JavaScript seems like a sensible option also because it might later share a part of the codebase with the *Runner* application.

Frontend Framework

Client-side web applications are now seldom developed using plain JavaScript - most developers utilize frontend frameworks and libraries for easier manipulation with the Document Object Model (DOM) tree and state management [StateOfFtd20].

According to a popular 2021 JavaScript developer survey [StateOfJS21], the most popular JavaScript (JS) frontend frameworks among developers are *React*, *Angular* and *Vue.js*. While the popularity of the tools changes over time with

new emerging technologies coming every year, the aforementioned tools have the largest number of third-party libraries because of their large following. and modules.

When comparing the frameworks against each other, *React* comes off as the framework with the steepest learning curve while being only slightly less popular than *Vue.js*, based on the GitHub stars of the project [AnReVue22]. While all the frameworks are utilizing the model of reusable components, *Angular* and *Vue.js* are taking the concept a little further with their internal HTML templating systems.

React is the only framework of those three utilizing JSX, i.e. combining the HTML syntax with the JS syntax. While this might pose certain difficulties for developers learning this framework, it allows them to interleave the HTML and JS code in a way that is more readable and easier to maintain in the end.

Since the *Editor*, given the requirements, should not require any advanced features of *Angular* and *Vue*, we can utilize the *React* framework for the *Editor*.

2.4.2 UI design

This subsection describes the specific UI design and presents mockups for the *Editor* application. The designs presented here should adhere to the nonfunctional requirements of user-friendliness and ease of use.

Please note that the following images do not represent the finished software, only the UI mock-ups. The mock-up designs are also available in *Figma*[6] for further inspection and future reference.

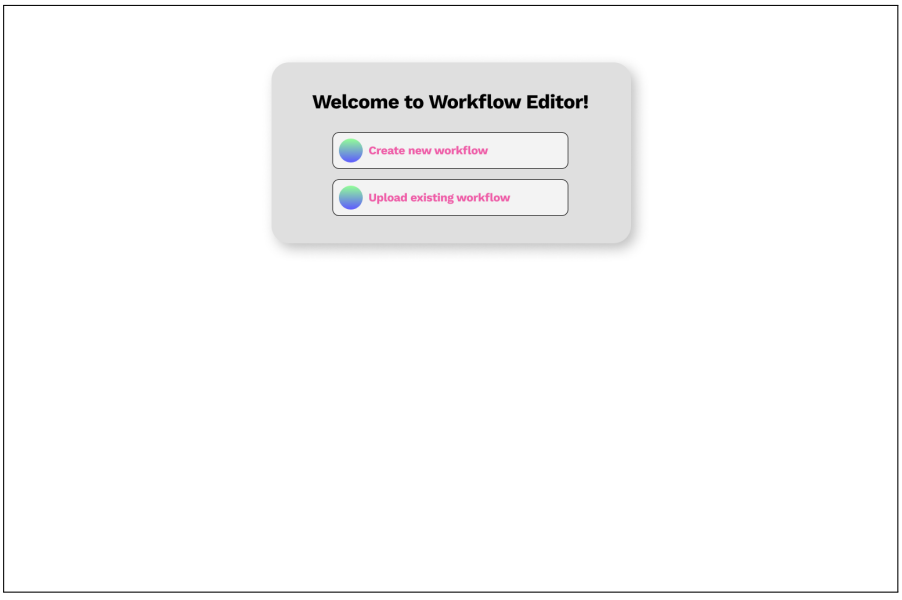

Figure 2.1: The initial screen

As described in the use case analysis, the initial screen of the application should allow the user to create a new blank automation file, or to load an existing automation file. Choosing to load an existing automation file opens a file selection dialog, allowing the user to select an file from their local file system.

[6]https://www.figma.com/file/gzisxDDNZX8vbvMdOjfMjT/wbr-editor

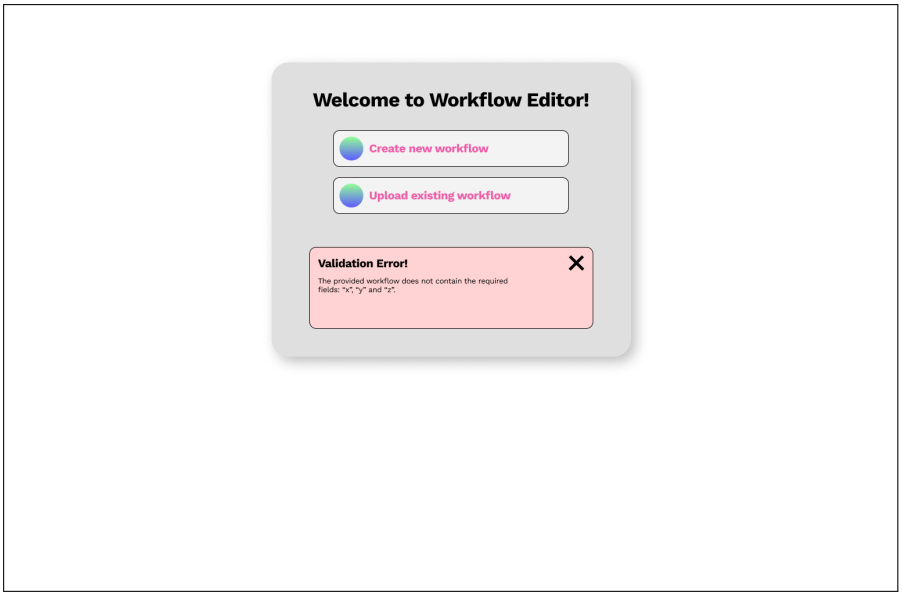

Figure 2.2: Validation error message

While creating a new blank file immediately opens the *Editor* application, in accordance with the functional requirement 1.2.1.1.3, an uploaded file needs to be validated first. If the provided file is not a valid workflow definition file, the user should be notified (Figure 2.2).

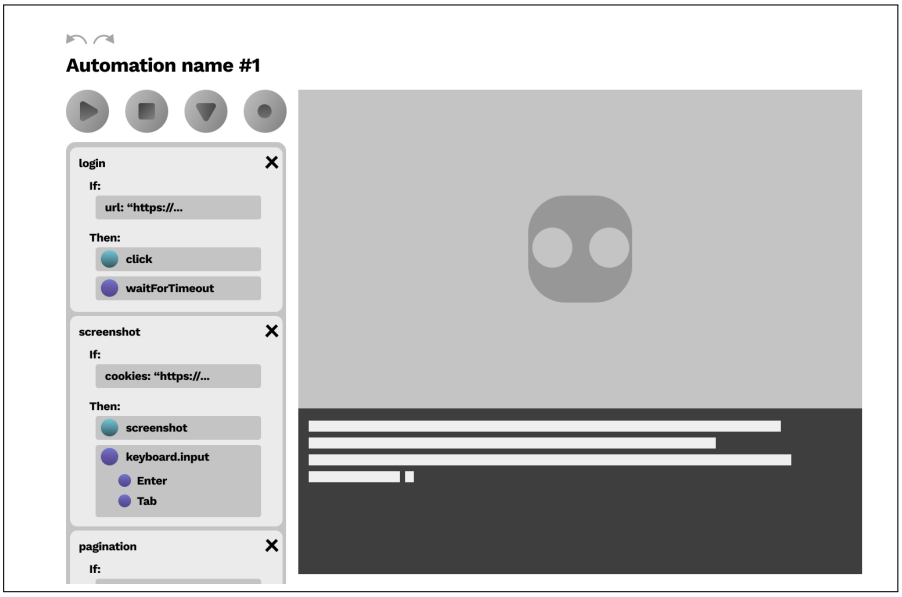

Figure 2.3: Workflow editor interface

After successfully opening a workflow definition file, the user is presented with the *Workflow editor* interface. Here, the user can edit the workflow definition file using a GUI editor - in the Figure 2.3, the workflow is presented in the left column.

The hierarchical structure of the workflow definition file allows us to present the workflow in a tree-like, collapsible structure, further enhancing the UX of the application.

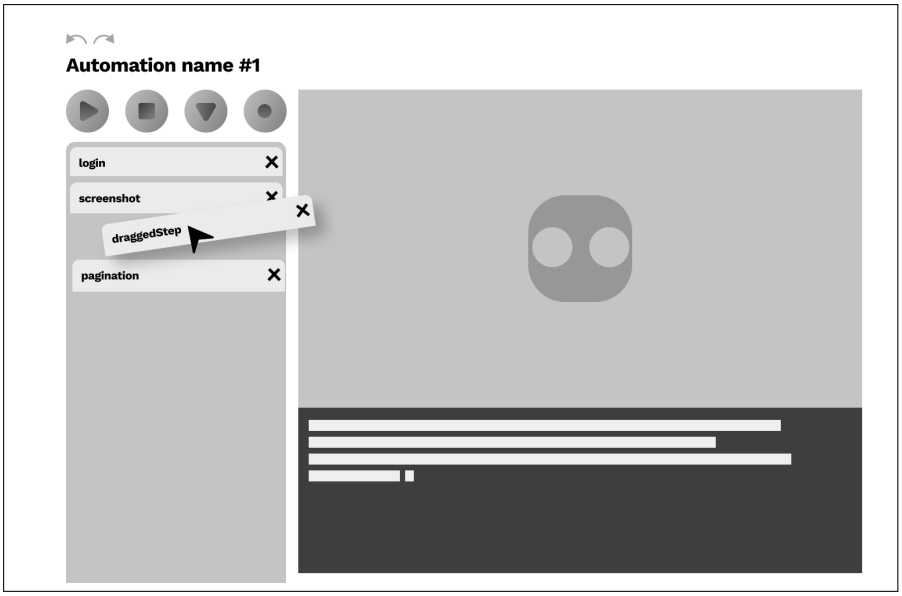

Figure 2.4: Drag & Drop controls

After finishing the editing of the workflow definition file, the user can test the workflow by clicking the *Run* button. This follows the functional requirement 1.2.1.1.6 on the *Runner* interface.

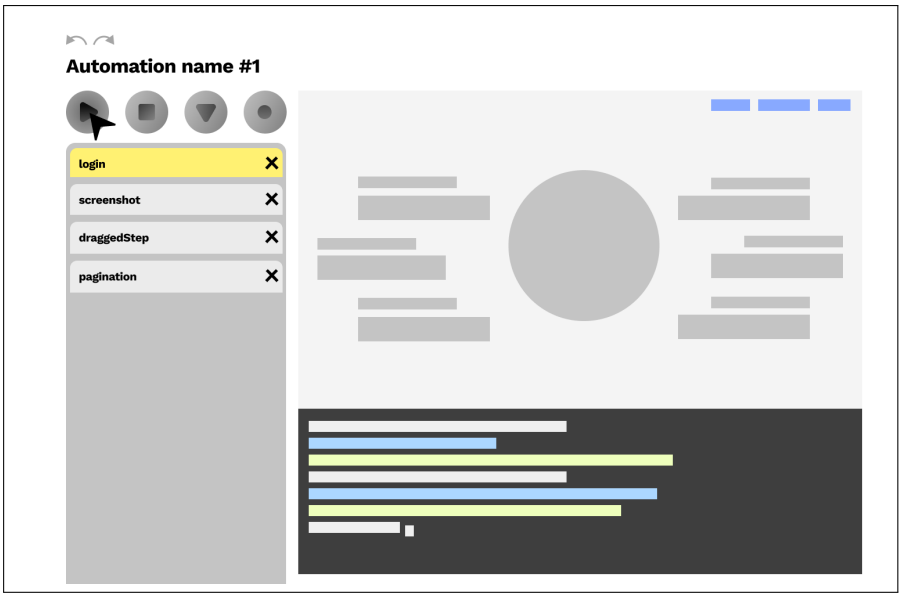

Figure 2.5: Workflow execution

In the Figure 2.5, the execution control buttons are located above the workflow editor column. The browser window showing the progress of the automation is in the right half of the screen. Below this window, there is a console for the user to view the output of the automation and debugging information.

During the automation execution, the *Editor* application can highlight the current step of the workflow, allowing the user to follow the execution progress.

3. Implementation

The toolkit source code is available in a public GitHub repository[1], along with the informal user documentation and the code and workflow file examples. The following chapter describes the decisions made during the implementation of the toolkit.

Just like in the previous chapters, the toolkit is described in a modular fashion, following the *Editor* - *Runner* dichotomy. This is also projected in the implementation, as both the *Editor* and the *Runner* are implemented as separate, standalone programs.

3.1 Runner

As stated in the section 1.2 Requirements, the *Runner* should be a piece of software enabling the user to run the automations created by the *Editor* application. During the implementation, only small changes were made to the initial design.

The *Runner* is implemented as a *Node.js* module and has been published as an `npm` package `@wbr-project/wbr-interpret@0.9.2`. The user-friendly *Runner* interface is a part of the *Editor* application, as both tools together create a simple and easy-to-use environment for developing the web automations.

3.1.1 Performance

According to the nonfunctional requirement *1.2.2.2.1*, the *Runner* should implement the automation execution in an optimized way. While this requirement is rather vague, there are actually several ways the *Runner* application tries to do so.

Parallelization

As mentioned in the subsection 2.2.3 Reactions, the workflow definition format is designed in such way that every step depends on the previous browser state only. This allows us to think of different browser tabs as of whole different environments [2] and let the *Runner* parallelize the automation between multiple tabs, possibly reducing the time required for the execution.

The `Interpreter` programmable interface allows the user to set maximum number of concurrent tabs. Using the proper method of enqueuing links in the workflow (action `enqueueLinks`) protects the internal *Runner* browser from opening too many tabs at once, which might hurt the performance. The enqueued links are then opened as individual tabs by the *Runner* with respect to the set concurrency.

In case a tab gets open e.g. as a popup window, it is not interacted with until the desired concurrency is reached. However, accumulating multiple such tabs can still lead to performace degradation, as they still have to exist in the browser memory.

[1] `https://github.com/barjin/wbr/`, tag `v1.0`, commit hash `bf45528225e3b9fc05963d75`

[2] Only regarding the workflow execution, they still can share e.g. cookies.

Browser communication

As mentioned in the section 2.3 Runner, the communication between the *Interpreter* part of the *Runner* and the internal web browser is facilitated using the *Playwright* library. While Playwright already provides an optimized way of communication with the browser using the CDP protocol and alternatives, the text-based interprocess communication still poses a certain performance bottleneck.

While designing the *Runner*, it was a priority to reduce the amount of calls to the *Playwright* library, as pretty much any *Playwright* call results in a CDP message being sent. During the condition matching phase of the workflow execution, the current browser state is fetched only once and the rule is then matched statically, instead of quering the browser repeatedly for the possible current URL, CSS selectors etc.

This is possible because of the simple design of the workflow definition format, allowing us to gather all the conditions statically. Knowing all the conditions, the full browser state can be then described by the truthiness/falsiness of those conditions, which is all that is needed for the decision making mechanism of the *Interpreter* to choose the next step to take.

3.1.2 Extra features

On top of the features described in the section 2.3 Runner, there are some additional features implemented into the *Runner* package. While those features are tested and are available in the `main` branch of the project, the other parts of the project - mainly the *Editor* - typically does not provide full support.

Workflow parametrization

The *Runner* module provides support for workflow parametrization. Any nonintegral part of the worklfow can be replaced with a special structure, for example like this:

```
{
    ...
        "url": { $param: "address" },
    ...
}
```

Before the workflow execution, the *Runner* receives a dictionary of the parameters' values, replacing every {`$param`} field with the declared value. When initialized with value {`"address" : "https://abc.xyz"`}, the example above turns into

```
{
    ...,
        "url": "https://abc.xyz",
    ...
}
```

In case the user does not provide values for all the parameters or provides values for parameters non-existent in the workflow, the *Runner* warns the user about this and does not continue with the workflow execution.

This feature can be utilized to create more universal workflow definitions, letting the end user to set certain parts of the workflow to match their use case. The parametrized fields can be e.g. login credentials, URL of the page to run the automation on or a custom message or data to paste to the website.

Automatic data extraction

While the *Runner* supports all the methods from the Playwright's `Page` class, it also implements methods for automating data extraction from the browser.

The `scrape` method allows the user to extract data from the current page by utilizing an algorithm looking for the "important" data in the page. The user can restrict the search to a specific element subtree by passing the selector of the root element as the only argument to this method.

The "importance" of the data in the page is determined using multiple heuristics, mostly by looking for similar-sized elements with similar content - these are believed to be the "scrapable" data - e.g. online store product cards, rows of a table, etc.

Guided data extraction

The `scrapeSchema` method acts as a guided counterpart of the `scrape` method. By specifying the names of columns and their respective selectors in the only argument of this method, the *Runner* extracts data from these selectors, and stores them in a dictionary, where the keys are the column names.

In case the selectors target multiple elements on the same page, the *Runner* will group the extracted data and output multiple dictionaries. If the numbers of the targetted elements do not match across the columns, the *Runner* tries to group the data by the DOM hierarchy in the web page, possibly leaving some output fields empty.

Here follows an example of the `scrapeSchema` method usage and the logic behind it.

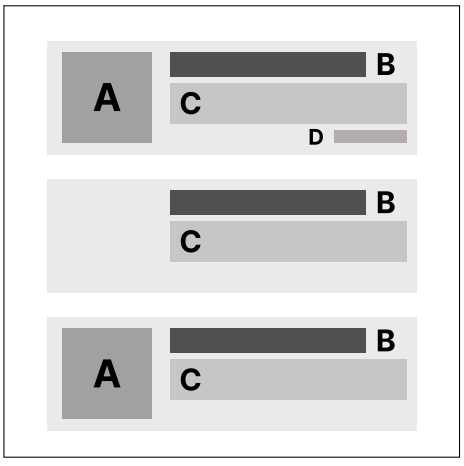

Figure 3.1: Example page to be scraped

The Figure 3.1 shows an example page containing a table of user profiles. A user card can contain a *photo of the user* (A), their *name* (B), their *profile description* (C) and their *phone number* (D). The letters represent the selectors for the *Runner* to extract the data from.

The user provides the `scrapeSchema` method with the following schema:

```
{
    "photo": "A",
    "name":  "B",
    "desc":  "C",
    "phone": "D"
}
```

The *Runner* extracts the data from the page. The data is stored in an array of dictionaries, where each dictionary corresponds to a user card.

```
[
    {
        "photo": "https://abc.xyz/img/user123.jpg",
        "name":  "John Doe",
        "desc":  "Lorem ipsum dolor sit...",
        "phone": "123-456-7890"
    },
    {
        "photo": undefined,
        "name":  "Mark Smith",
        "desc":  "Ipsum dolor amet sit...",
        "phone": undefined
    },
    {
        "photo": "https://abc.xyz/img/user234.jpg",
        "name":  "Jane Green",
        "desc":  "Sit dolor ipsum lorem...",
        "phone": undefined
    },
]
```

Note how the contents of the corresponding elements are paired, leaving the missing fields empty. This is done by traversing the DOM tree of the page, and grouping the elements by their common parents.

3.2 Editor

As described in the section 2.4 *Editor*, the workflow editor is implemented as a React Application. The following section describes decisions made during the implementation of the *Editor* application, certain problems and their solutions.

3.2.1 React

While it is possible to set up a *React* application as a plain Node.js application, it is not recommended. Handling the bundler configuration is cumbersome and

requires a respectable amount of knowledge about the React toolchain. The React's signature JSX syntax also requires configuring a transpiler (e.g. `babel`[3]), which adds another level of complexity.

For those reasons, the *Editor React* application has been initialized with `create-react-app`[4]. This is a command line interface (CLI) tool for simple initialization of *React* applications. It mainly provides the *Webpack* and *Babel* configuration files and bootstraps the project with a template website.

Configuring Webpack polyfills

For validating the uploaded worklflow files, the *Editor* application imports the *Preprocessor* class from the *Interpret* module. While both the *Editor* and *Interpret* module are written in TypeScript, there are slight differences between *Node.js* and browser code.

Most of those problems stem from the module resolution, as both *Node.js* and browsers use slightly different approaches to module loading. While this is covered by Webpack and Babel in most cases, importing the *Interpret* module initially caused errors.

This is because since the version `5.0.0`, *Webpack* no longer provides polyfills for the core *Node.js* modules, such as `path` used by the *Interpret* module. The `path` module is not used by the validation feature of the *Preprocessor*, so the polyfill could be simply disabled. However, doing so requires a manual update of the *Webpack* configuration.

Since the *Editor* has been created with `create-react-app`, the *Webpack* configuration `webpack.conf.js` is contained in the `node_modules/react-scripts` directory. While it might be possible to modify the `webpack.conf.js` file directly, it is not recommended - the `node_modules` directory is used to save the packages downloaded from the NPM registry. Updating the `react-scripts` package would then reset all changes made to the configuration file. Furthermore, because of it's dynamic nature, the `node_modules` directory is omitted from the `git` version control.

While the official `create-react-app` guide suggests to perform `eject`, i.e. to export the configuration files for manual maintenance, this is a borderline unsafe step. The `eject` script performs irreversible changes to the package structure, and forces the user to maintain the configuration and dependencies themselves from then on.

To retain the simplicity of automatic project management, the project now uses `react-app-rewired`[5]. Published as an `npm` package, `react-app-rewired` is an drop-in replacement of `create-react-app`, which lets the developer to sideload custom configuration files, while still maintaining the base configuration.

3.2.2 Improving the UX

Given the nonfunctional requirements for the *Editor*, the *Editor* application should adhere to the best UI/UX practices and provide a steep learning curve. This is manifested multiple times in the *Editor* application itself.

[3] `https://babeljs.io/`

[4] `https://create-react-app.dev/`

[5] `https://www.npmjs.com/package/react-app-rewired`

Drag & Drop

Since the main control elements in the *Editor* are modular blocks, allowing the user to use drag&drop controls e.g. for reordering the blocks seems like the superior solution in terms of UX.

The drag&drop feature is used for reordering the blocks in the *Editor* application. While it would be possible to drag the entire blocks around the application window, it is not recommended due to UX reasons. This is solved in the *Editor* application by collapsing all the blocks when the drag&drop action is initiated. This helps the user to focus on the actual meaning of the reordering action, and not on the content of the blocks.

The implementation of the drag&drop feature is based on the `react-dnd`[6] library. While the authors of this library offer an official example of reordering a list of block elements, the *Editor* implementation is not based on this example due to the mentioned features, which are not provided by the example solution.

Argument type suggestions

The *Editor* application generates a workflow interpretable by the *Runner* application, which is internally using the *Playwright* library. The function signatures of the workflow reaction steps depend on the *Playwright* library, as most of the supported steps are mirrored from the `Page` class methods. While the user might set the types of the arguments themselves, the *Editor* application aims to provide a simple way of creating the web automations.

For this reason, the *Editor* application provides a prefilled list of arguments with the correct types and matching input elements, which correspond to the *Playwright* `Page` class methods and present the correct usage of the arguments to the user. The range of optional arguments is hand-picked for every command to provide the best variability while still maintaining an exceptional level of user-friendliness.

Real-time validation

On top of the mentioned argument type suggestion feature, the *Editor* application also validates most of the user inputs based on their context in the workflow definition. Just like the previously mentioned features, the only purpose of this is enhancement of UX and a user guide; a workflow definition with inputs marked as invalid still can be executed.

[6] `https://react-dnd.github.io/react-dnd/`

4. Documentation

The following chapter contains the documentation of the project. It is divided into several sections based on the amount of experience of the reader and the desired actions.

4.1 User documentation

The following section contains user documentation for the *Editor* application, following the scenarios from the chapter 1 Analysis. This section describes the basic features of the *Editor* application and how to use it.

The documentation of the *Runner* module is to be found in the section 4.2 Developer documentation, as there is no user interface for the *Runner* module itself and the module is to be used only via its application programming interface (API) in code.

4.1.1 Editor application

The *Editor* application is available as a web application. To access the user interface, navigate to the URL of the running server instance using a modern web browser. Instructions on how to setup the *Editor* server instance are to be found in the section 4.3 Administrator documentation.

Getting started

After accessing the *Editor* application, the user is greeted by a welcome screen. From here, they can choose whether to create new blank automation or to load existing automation from their device. In case the uploaded file is not a valid workflow definition, the user is informed about the reason and asked to try again.

The user can also choose to open one of the example automations provided in the menu. While the automations themselves are part of the application, their reliability cannot be guaranteed. The functionality of those automations relies on the state of the targetted third-party webpages.

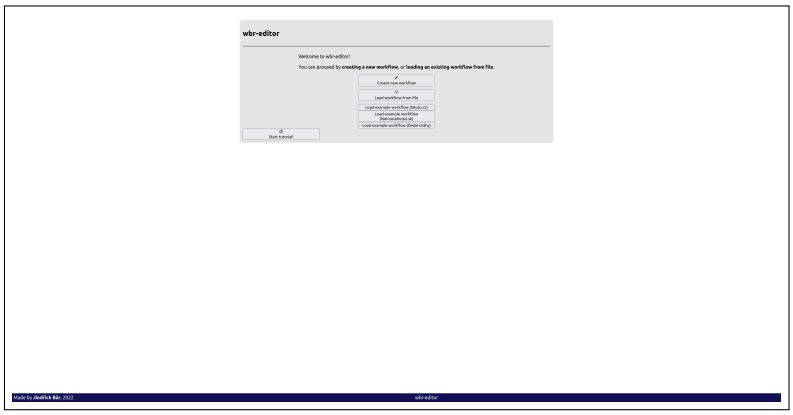

Figure 4.1: The initial screen

Adding rules

As described in the section 2.2 Workflow definition format, the workflow definition file is a JSON file, consisting of *Rules* - *Conditions* and their matching *Actions*.

The user can add a new blank pair to the workflow definition file by clicking the blue square `'+'` (*Plus*) button in the bottom of the workflow editor.

Figure 4.2: The workflow editor with a new blank workflow

After clicking the `'+'` button, the currently edited workflow is updated, appending the new blank pair to the end of the workflow definition. Clicking the `'+'` button repeatedly adds new blank pairs to the end of the workflow definition.

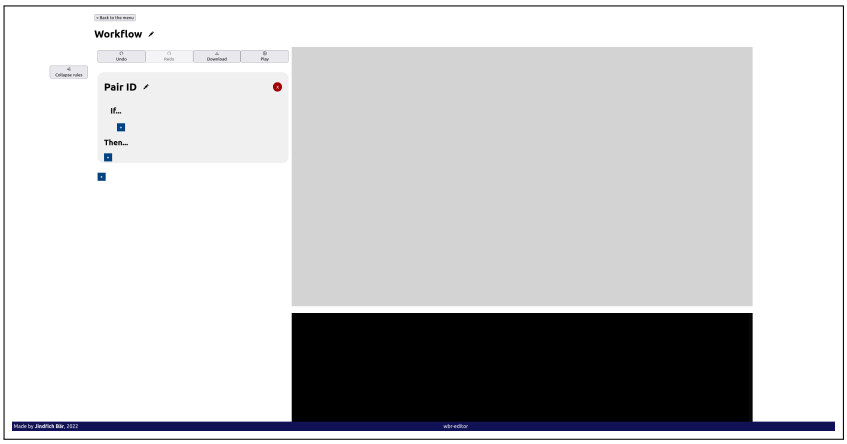

Figure 4.3: A new blank pair is added to the workflow

Unwanted pairs can be removed by clicking the red round `'x'` button in the top right corner of the pair editor. In such case, the workflow is updated and the pairs underneath the removed one are shifted up.

Editing the conditions

As stated in the section 2.2, Workflow definition format, the pairs are defined as *Conditions* and *Actions*. The automation interpreter is able to find the matching condition for the current state of the browser and execute the corresponding set of actions.

The pair's *Condition* is displayed under the section "If" in the workflow editor. The condition is defined as a possibly nested tree of simple condition expressions and logical operators. By default, the condition consists of the (invisible) top-level `$and` operator. This operator is used to combine the first-level conditions - all of the provided conditions must be true for the pair to be true. Not specifying any conditions will result in creating an empty clause - a condition that is always true. This is useful e.g. for defining a default set of actions.

The conditions can be specified by clicking the blue square `'+'` button in the bottom of the operator section.

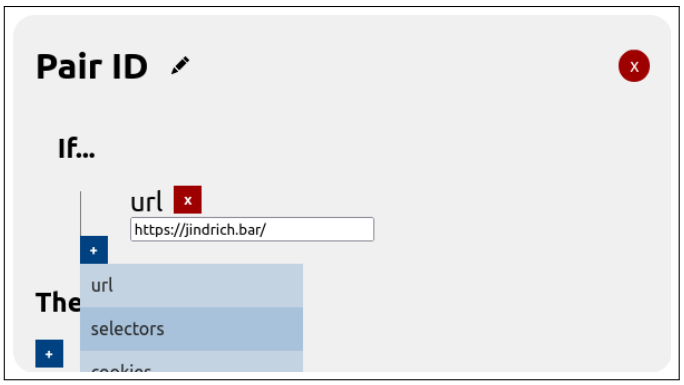

Figure 4.4: Specifying a condition within an `$and` operator

The Figure 4.4 shows the dropdown condition selector for the `$and` operator. The `'url'` condition above has been inserted and initialized before. All of the inserted conditions provide the correct type and related input element to ensure better user experience.

Adding a new condition from the presented dropdown menu will result in inserting a new "neighbor" condition of the `url` condition. This, together with the top-level `$and` operator, results in an combined condition, which is evaluated to true if all of the nested conditions is true. The dropdown menu also allows the user to insert a nested logic operator (`$and` or `$or`).

Reordering the rules

The order of the pairs in the workflow definition file is important. The workflow interpreter will evaluate the pairs in the order of the pairs in the workflow definition file and match the first pair that is true.

This means that the order of the pairs in the workflow definition corresponds to the priority of their conditions. The more specific the condition, the higher it should be placed in the workflow definition file, so it does not get overshadowed by other, possibly more general, pairs. The pairs can be reordered using a drag and drop interface.

Adding the actions

After specifying the conditions for the given pair, the user can specify the actions to be executed when the condition is true.

The pair's *Actions* are displayed under the section "Then" in the workflow editor. The set of actions is defined as a list of parametrized action expressions.

A new action can be selected from the dropdown menu by clicking the blue square `'+'` button in the bottom of the action section.

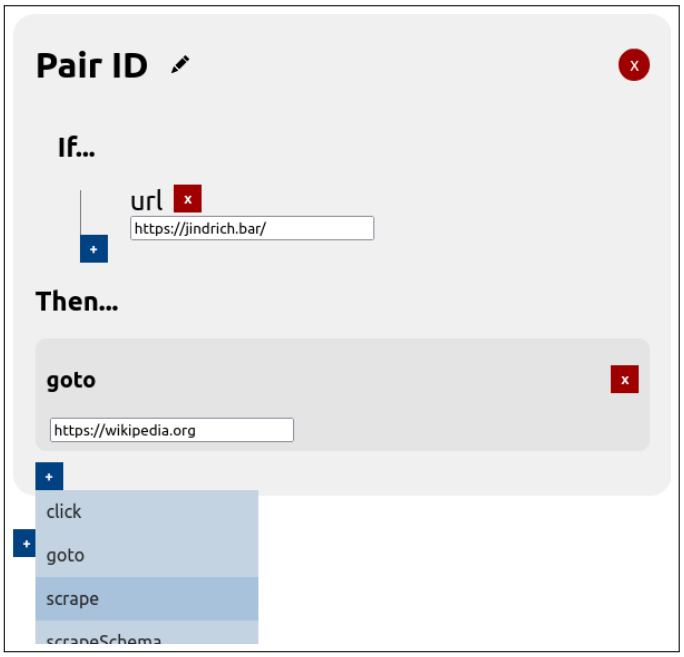

Figure 4.5: Selecting the actions for a pair

The Figure 4.5 shows the process of adding new actions to the action sequence corresponding to the pair.

Selecting an action from the dropdown menu results in inserting a new action expression into the action sequence. The action expression is defined by the action type and optional parameters for the action - e.g. the `'click'` action requires a `'selector'` parameter, the `'goto'` action requires a `'url'` parameter etc.

The *Editor* application suggests the required parameters for the individual actions to enhance the user experience and avoid errors.

While the `wbr-interpret` module provides support for all the *Playwright's Page* methods, the *Editor* application provides only a subset of the most common actions.

Testing the automation

After creating a new automation, the user can test run the automation by clicking the `'Run'` button in the top left part of the screen.

This executes the automation, displaying the execution progress in the remote browser window on the right side of the screen. Additional information about the execution is displayed in the console below the browser window. To further facili-

tate the automation debugging, the *Editor* also highlights the currently matched pair in the workflow definition section.

To ensure consistency, any update to the workflow definition made during the workflow execution will stop the automation execution. The execution can be also stopped manually by clicking the `'Stop'` button in right part of the control panel.

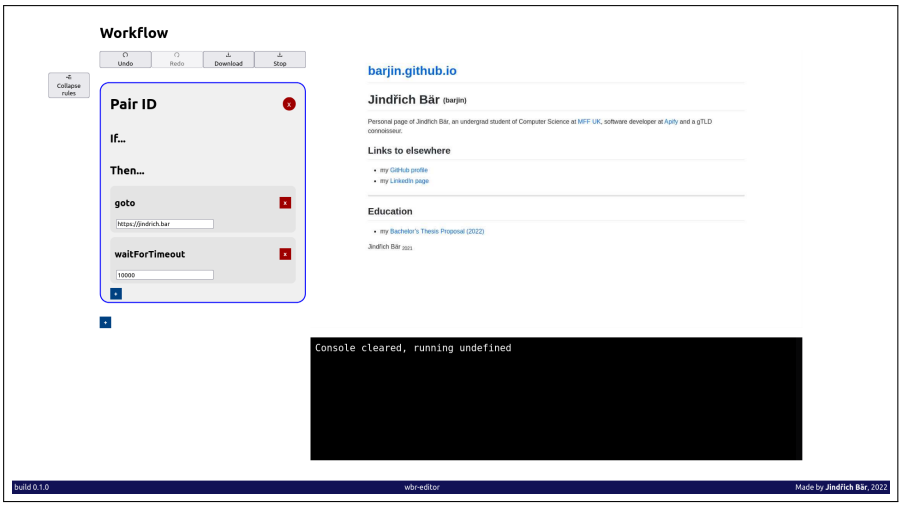

Figure 4.6: Execution of a simple workflow

4.2 Developer documentation

This section contains the developer documentation for the `wbr-interpret` module and the *Editor* application. The documentation presented here describes the main design principles of both pieces of software and how they are implemented.

4.2.1 wbr-interpret

To run the web automations made in the *Editor* application programatically, it is possible to install the `wbr-interpret` module as an `npm` package.[1]

This is simply done by running the command

```
npm i -s @wbr-project/wbr-interpret@0.9.2
```

in one's *Node* project folder. This installs the module into the current project's dependency tree.

The module package contains extensive TS typings, further simplifying the development. The type definition in this package also contains typings for the workflow definition files, allowing the user to update the workflow files manually, utilizing the TypeScript IDE suggestions and validation.

Module usage

Using the `wbr-interpret` module from one's code is quite simple. The main *Interpreter* class provides only two public methods - `run()` and `stop()`. The intended usage is then demonstrated in the following example.[2]

```
1  const Interpreter = require('wbr-interpret');
2  const { chromium } = require('playwright');
3
4  const workflow = {...}
5
6  (async () => {
7      const browser = await chromium.launch();
8      const page = await browser.newPage();
9      const interpreter = new Interpreter(workflow);
10
11     await interpreter.run(page, { paramName: '
           paramValue' });
12     await page.close();
13     await browser.close();
14 })();
```

As shown in the example, the *Interpreter* class accepts the workflow definition in the constructor. The `run()` method of the *Interpreter* takes a Playwright `page` object as an argument and a `parameters` object as an optional second argument.

[1] Version `0.9.2` is the latest version of the `wbr-interpret` module at the time of submission of this work.

[2] This example uses CommonJS module syntax.

The package structure

The majority of the logic in the `wbr-interpret` module is implemented in two main files: `interpret.ts` and `preprocessor.ts`. Both of those files also import utility functions from the files located in the `utils/` directory.

`interpret.ts` contains the main logic for the workflow execution. While the `wbr-interpret` module mostly imitates the behaviour of the *Playwright*'s `Page` class during the workflow execution, it also provides custom functions or overrides for the existing `Page` class functions, updating their behaviour to suit the interface of `wbr-interpret` better. The `interpret.ts` file contains implementation for those methods as well.

`preprocessor.ts` contains the validation logic for the workflow definition files, as well as methods for workflow initialization. The methods from this file can be used for runtime validation of the workflow definition files and other preprocessing analysis tasks.

More detailed descriptions of the code in the `wbr-interpret` module can be found in the Attachment A.2.

4.2.2 wbr-editor

The React application `wbr-editor` is a web application that allows the user to create and edit web automation workflows. The following section describes the main design principles of the `wbr-editor` application.

To run the development server, the user requires to run the following sequence of commands, while in the `wbr-editor` project folder:

```
npm i -s
npm start
```

This runs the `react-scripts` development server on a free port (typically `3000`). The user documentation for the web application is available in the section 4.1 User documentation.

Please note that the `react-scripts` development server does not contain the backend logic for the automation execution and is meant to be run only during the development of the web application.

If the user wants to run the web application in production mode, it needs to be built first. This is done simply by using the following command:

```
npm i -s
npm run build
```

Package structure

The `wbr-editor` package is structured as a regular *React* application. All the source code files are located in the `src/` directory. The static files (i.e. images) are located in the `public/` directory.

Closer description of the file contents can be found in the Attachment A.3.

4.2.3 wbr-cloud

The `wbr-cloud` package is factually a part of the `wbr-editor` application, serving as its backend. Due to design decisions made prior to the implementation of the `wbr-editor` package, these packages are not directly related and are developed as two individual packages.

Technically speaking, the `wbr-cloud` package is an HTTP server used for serving the *Editor* application and providing a REST API for instantiating and managing the web automation runs.

Detailed information about the specific REST API endpoints can be found in the Attachment A.4.

4.3 Administrator documentation

The following section contains the administrator documentation of the project. It contains installation instructions and a short troubleshooting guide for setting up a server for the *Editor* and *Runner* applications. It also contains a list of system requirements on the server where the applications are to be deployed.

Aside from this, this section also contains installation instructions for the *Runner* `npm` package, enabling the user to create software able to execute and validate the workflow definition files.

4.3.1 Editor application

Installation instructions

The *Editor* and *Runner* application are both packaged in the Docker image `barjin/wbr`. This Docker image[3] represents a complete server setup, including all the dependencies and the *Editor* and *Runner* applications. To run a Docker image, the workstation must have *Docker* installed. Aside from meeting the *Docker* system requirements, the *Editor* Docker image presses no other requirements on the workstation.

The web server providing the user interface is running on port `8080` of the Docker container, which is also the only port utilized by the *Editor* and *Runner* applications. On a system with `docker` installed, running the container requires only one command:

```
docker run -p HOSTPORT:8080 -d barjin/wbr
```

where `HOSTPORT` is the port number used by the *Editor* and *Runner* applications on the host machine. After running this command, the user interface should be now available at `http://localhost:HOSTPORT/`.

The `-d` option instructs Docker to run the container in so-called 'detached mode', which means that the container is not terminated when the command finishes. [DockerRef] In case the user wants to stop the container, they can use the `docker stop` command with the container ID.

In case the user never ran the `docker run` nor the `docker pull` command before, the *Docker daemon* first downloads the `barjin/wbr` image from the Docker

[3] Tag `barjin/wbr:final`, `sha256:b9e237a3ccf619f4a9b36e6584191bf`

Hub. Note that this can cause a delay of several seconds and consume a certain amount of bandwidth (ca. 200 MiB).

The Docker image can also be built from the attached Dockerfile by invoking the `docker build` command in the root folder of the project repository.

Please note that the *Editor* application allows the users to run arbitrary code on the server. This is a security risk, and the user is advised to only share their instance of *Editor* server with trusted users.

Build instructions

Besides the *Docker* image, all the software source files are also available in the project's GitHub repository.[4]

The `package.json` files for the packages `wbr-interpret`, `wbr-editor` and `wbr-cloud` contain the build instructions for the respective applications, as well as the required dependencies.

> **Note:** `npm` is required for building the *Editor* and *Runner* applications. Before building the applications, run
>
> ```
> npm run confBuildDeps
> ```
>
> command in the root folder of the project repository. This installs the correct versions of build dependencies in the correct order. Running the `build` command without the dependency installation - or installing the build dependecies in a different way - can result in build errors and/or unexpected behavior.

The build process is managed by the *Turborepo*[5] build system. *Turborepo* allows for faster build times and less wordy build configurations by utilizing a `make`-like approach to the build process. It caches the built files and allows for incremental builds and faster rebuilds. It also constructs a dependency graph of the source packages by reading their `package.json` files, which is used to determine the order in which the packages are built.

The `turbo` build is invoked by the `npm run build` command in the root folder of the repository. Invoking the `npm start` command in the root folder of the repository after the build starts the *Editor* application server.

While the build process has been made to be as streamlined as possible, the author of this work gives no guarantees about the build process. Use the official *Docker* image when possible.

[4] `https://github.com/barjin/wbr/`, tag `v1.0`, commit hash `bf45528225e3b9fc05963d75`
[5] `https://turborepo.org/`

5. Testing

During the development phase, all parts of the project have been tested using various testing methods. The following chapter describes the testing methodology and the nature of the individual tests. Aside from testing the software using automated test suites, user testing was carried out on the *Editor* application.

5.1 Code quality

While the code quality does not directly influence the correctness of the results, it is an important aspect of the project, as it influences readability and maintainability of the code. Maintaining a consistent code style also speeds up the development process and reduces the risk of introducing bugs.

The `wbr-interpret` module and the *Editor* application are written in *TypeScript* and *TSX*, respectively. The codebase of the entire project follows the practices described in the *Airbnb JavaScript Style Guide*[1], made and maintained by *Airbnb, Inc. Airbnb Inc.* also provides a `.eslintrc` file for configuring *ESLint* in accordance with the practices described in the *Style Guide*. All the code in the project is then automatically checked for conforming to the style guide using `ESLint` already during the development phase, which simplifies formatting and allows us to see potential errors right away.

The project's GitHub repository also contains an automated *GitHub Actions* workflow, running the *ESLint* check with every push to the repository. This helps with catching the bad code patterns early on and prevents the need for further manual testing.

5.2 Automated tests

To ensure the elementary correctness of the project parts, the project has contained automated test suites since the early stage of development. This helps both with code maintenance and new feature implementation, as the automated test suites ensure we do not introduce any breaking changes.

All of the tests mentioned in the following sections are also ran in the corresponding *GitHub Actions* workflow. This ensures that the code available in the main branch of the public repository is always tested against the latest changes.

All the following tests were carried out using the `jest`[2] testing framework, if not stated otherwise.

5.2.1 Unit tests

The entirety of the `wbr-interpret` package is covered with unit tests testing the individual components of the package. Special attention is paid to the *Interpreter* and *Preprocessor* classes, as those are the core of the project and contain the largest amount of complex code.

[1] https://airbnb.io/javascript/
[2] https://jestjs.io/

The success of the unit tests is directly related to the code design of the module, since almost all methods in both classes were first written as pure functions, and only then they were converted to classes for encapsulation and to provide a more convenient interface.

While it is possible to unit test components in a React application, most of the components in the *Editor* application are trivial. For this reason, there are no unit tests for the *Editor* application and all the testing is carried out by E2E tests.

5.2.2 E2E tests

Both the *Editor* application and `wbr-interpret` package have been tested with end-to-end tests as well. These tests are typically longer and follow a complete user path, testing the entire application from the user's perspective.

The `wbr-interpret` package is end-to-end tested using custom scripts (not `jest`) for two different scenarios:

- Loading, validating and executing a simple workflow.
- Loading, validating, initializing and executing an advanced, parametrized workflow.

Both E2E tests are run against a local server, which is started before the tests are run. This way we can observe the actions carried out by the *Runner* even without obtaining any output data.

The *Editor* application has been tested using *Playwright* library to simulate the user's interaction with the application. There are two tested scenarios:

- Uploading various workflow files and seeing which one is valid.
- Creating new automation, editing it, running it, seeing the results and downloading the workflow definition file.

All the automations are also run against a local server, which allows us to control the content of the webpages and observe the actions caused by the automation execution.

5.3 User testing

Aside from the code testing, the *Editor* application has also been tested by 8 potential users of different technical backgrounds. This was done in two parts. First, the users were asked to use the *Editor* application to create a simple automation file. After getting accustomed to the software, the users were given a Software Usability Scale (SUS) survey and were asked to rate the application's usability.

5.3.1 Test scenario

The task for the users testing the *Editor* application was to create a simple web automation using the application and execute it to validate the functionality. Such a task requires utilizing a number of UI elements of the *Editor* application, while not being too demanding or requiring much technical knowledge.

The created automation should be able to perform the following steps:

1. Navigate to https://jindrich.bar/.
2. Click an arbitrary link on the page.
3. If the resulting page contains paragraphs with text, download these first, otherwise just terminate the execution.

The users were provided no additional support during the task. Out of the eight users, only three of them were able to successfully complete the task, while the others failed at various points. The successful users were all computer specialists with a strong technical background. While this might show a trend towards excessive technicality of the application, the sample size is not significant enough to make any conclusions.

The individual user complaints bring better insight into the why some users failed to complete the task. The less successful users complained mostly about the complexity of the workflow definition format and the confusing logic of the workflow interpretation.

There were no direct complaints about the application's user interface layout. After concluding the experiment, all of the users were instinctively able to find all the control elements when asked to perform a specific action. All of them were also able to answer questions about the system state.

5.3.2 SUS survey

All the testers have also shared their opinions on the usability of the application via the SUS survey. The detailed results of the SUS survey are included in the Attachment A.1 SUS Survey Results. The attachment also contains additional information about the survey itself.

The average SUS score over the ratings of the eight users is **52.5**. This sets the *Editor* application somewhere between the *10th* and *20th* percentile of typical SUS evaluated systems. While this might come off as negative result, it is still a valuable indicator of the quality of the *Editor* application.

The main reasons for the low ratings the users stated were *the workflow definition format being confusing* and the general unfamiliarity with debugging tools. While it would be easy to brush these points off as too subjective, the goal of the thesis was to create a tool that would be easy to use and intuitive for the user - which makes all the user complaints very relevant.

6. Conclusion

The goal of this work, as stated in the Introduction was to *create a human-readable, declarative format for storing and creating web automations, with an interpreter of this format and a visual editor, allowing less technical users to create and maintain automations in this format.*

By implementing the *Editor* application and the *Runner* module, this goal has been achieved in full scale. Both parts of the project are now fully functional - in accordance with the requirements from the section 1.2 Requiements - and can be used to create and edit web automations for data extraction and process automation. Both parts of the project have been developed in a modular fashion, which makes the further development process easier and faster.

Future work on the project includes further simplification of the *Editor* application, as the current implementation does not quite fulfill the user expectations. The simple format design could allow for automatic generation of the workflow definition files, allowing the users to create the resilient web automations by e.g. recording their actions in a browser. The *Runner* module could also be updated with more advanced features, better support for automated data extraction or support for crawling the web.

Bibliography

Andrew Hayes. YAML vs JSON vs XML in Go. 2021. [Online; accessed 6 March 2022].

Applitools. 2022's Most Popular Programming Languages for UI Test Automation. https://www.slideshare.net/Applitools/2022s-most-popular-programming-languages-for-ui-test-automation, 2021. [Online; accessed 22 February 2022].

Aris Pattakos. Angular vs React vs Vue 2022. https://athemes.com/guides/angular-vs-react-vs-vue/, 2021. [Online; accessed 7 April 2022].

Cypress.io. Cypress - Conditional Testing. https://docs.cypress.io/guides/core-concepts/conditional-testing, 2022? [Online; accessed 27 February 2022].

Docker Inc. Docker run reference. https://docs.docker.com/engine/reference/run/, 2013-2023. [Online; accessed 9 April 2022].

Giovanni Rago. Cypress vs Selenium vs Playwright vs Puppeteer speed comparison. https://blog.checklyhq.com/cypress-vs-selenium-vs-playwright-vs-puppeteer-speed-comparison/#scenario-3-test-suite-against-a-production-web-app, 2021. [Online; accessed 26 February 2022].

Google. Google Trends - Data Formats. https://trends.google.com/trends/explore?date=all&q=json,yaml,xml,rdf, 2022. [Online; accessed 6 March 2022].

Market Research Future. Web Scraper Software Market Research Report. https://www.marketresearchfuture.com/reports/web-scraper-software-market-10347, 2020. [Online; accessed 22 February 2022].

Microsoft. Playwright - Browser Patches. https://github.com/microsoft/playwright/blob/main/browser_patches/README.md, 2021. [Online; accessed 27 February 2022].

Microsoft. Playwright - Browsers. https://playwright.dev/docs/browsers, 2022. [Online; accessed 27 February 2022].

MongoDB, Inc. MongoDB - Query and Projection Operators. https://docs.mongodb.com/manual/reference/operator/query/, 2020? [Online; accessed 13 March 2022].

Sacha Greif. State of JS 2021. https://2021.stateofjs.com/en-US/libraries/front-end-frameworks, 2021. [Online; accessed 7 April 2022].

Robert Sebesta. *Concepts of Programming Languages (11th Edition)*. Pearson, hardcover edition, 2 2015. ISBN 978-0133943023.

Selenium. Selenium IDE - Control Flow. https://www.selenium.dev/selenium-ide/docs/en/introduction/control-flow, 2022? [Online; accessed 27 February 2022].

The Software House. State of Frontend 2020. https://tsh.io/frontend-development-trends-2020/, 2020. [Online; accessed 7 April 2022].

Usability.gov. User Interface Design Basics. https://www.usability.gov/what-and-why/user-interface-design.html, 2014. [Online; accessed 8 May 2022].

List of Figures

List of Tables

List of Abbreviations

GUI graphical user interface

API application programming interface

CLI command line interface

RPA robotic process automation

QA quality assurance

E2E end-to-end

SW software

UX user experience

UI user interface

CDP Chrome DevTools Protocol

CSS Cascade Style Sheet

IDE integrated development environment

JS JavaScript

TS TypeScript

JSX JavaScript Extended Syntax

HTML Hypertext Markup Language

WYSIWYG What You See Is What You Get

DOM Document Object Model

CORS cross-origin resource sharing

NPM Node Package Manager

SUS Software Usability Scale

A. Attachments

A.1 SUS survey details

This Attachment contains the detailed results of the SUS survey for the *Editor* application. While the SUS is somewhat standardized set of questions for evaluating the quality of software, the questions can be updated to match the evaluated software's needs. For completeness, the questions are included in here:

1. I think that I would like to use this application frequently.
2. I found the application unnecessarily complex.
3. I thought the application was easy to use.
4. I think that I would need the support of a technical person to be able to use this application.
5. I found the various functions in this application were well integrated.
6. I thought there was too much inconsistency in this application.
7. I would imagine that people would learn to use this application very quickly.
8. I found the application very cumbersome to use.
9. I felt very confident using the application.
10. I needed to learn a lot of things before I could get going with this application.

The response scale for each question is then a 5 point Likert agreement scale:

Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree
1	2	3	4	5

The results of the evaluees' survey are presented in the following table:

	1	2	3	4	5	6	7	8	9	10	**SUS**
User 1	**3**	**2**	**4**	**2**	**3**	**2**	**3**	**2**	**4**	**1**	**70**
User 2	**2**	**2**	**2**	**3**	**4**	**3**	**2**	**2**	**3**	**2**	**52.5**
User 3	**4**	**2**	**4**	**2**	**5**	**2**	**4**	**1**	**5**	**2**	**82.5**
User 4	**3**	**3**	**2**	**4**	**4**	**3**	**3**	**3**	**3**	**4**	**45**
User 5	**4**	**2**	**3**	**2**	**4**	**1**	**2**	**2**	**4**	**1**	**72.5**
User 6	**1**	**3**	**2**	**4**	**3**	**2**	**2**	**3**	**1**	**3**	**35**
User 7	**2**	**3**	**2**	**4**	**2**	**3**	**2**	**3**	**2**	**4**	**32.5**
User 8	**1**	**2**	**2**	**5**	**3**	**3**	**2**	**4**	**1**	**3**	**30**
Average	**2.5**	**2.4**	**2.6**	**3.3**	**3.5**	**2.4**	**2.5**	**2.5**	**2.9**	**2.5**	**52.5**

A.2 `wbr-interpret` code documentation

This attachment contains in-depth documentation of the `wbr-interpret` module, with detailed descriptions of the classes and methods used in the `wbr-interpret` module.

The following content is structured by individual files.

A.2.1 `interpret.ts`

The *Interpreter* class defined in the `interpret.ts` file implements the main logic for the workflow execution.

Type Aliases and Interfaces

In the following subsection, we define the type aliases and interfaces used in the `interpret.ts` file.

`interface InterpreterOptions`

This interface describes the object of the optional parameters passed to the `Interpreter` class constructor. All of the following fields are optional, as is the whole object.

- `maxRepeats: number`
  - The maximum number of times a condition-action pair can be repeated without interruption before the execution of the workflow is stopped. This option defaults to `5`. Setting this to `0` disables this loop protection mechanism.
- `maxConcurrency: number`
  - The maximum number of concurrent browser tabs that can be handled by the *Interpreter*. This option defaults to `5`. Setting this option to `null` disables the concurrency limit. This can lead to overloading the browser and thus to a browser crash.
- `serializableCallback: (output: any) => (void | Promise<void>)`
  - The function called when a serializable output is received from the browser. An object is *Serializable* iff `JSON.stringify` can be called on it without throwing an exception. Defaults to `console.log`.
- `binaryCallback: (output: any, mimeType: string) => (void | Promise<void>)`
  - The function called when a non-serializable output is received from the browser. The data is accompanied by its *MIME* type passed in `mimeType` parameter. Default behavior logs a warning message in the console.

- `debug: boolean`
  - If set to `true`, the *Interpreter* will log additional debug messages to the console.
- `debugChannel: Record<string, unknown>`
  - An object containing the following fields (all optional):
    - `debugMessage: Function`
      - If set, this function is called with all the log messages emitted by the *Interpreter*. Can be used to run a remote console for monitoring the automation run.
    - `activeId: Function`
      - If set, this function is called with every new matched pair's index within the workflow definition.

Methods

In the following subsection, both public and private methods of the `Interpreter` class are described. The relations between those are also explained here, making it easier to understand the code.

```
constructor(workflow:WorkflowFile,
            options?:Partial<InterpreterOptions>)
```

The constructor for the Interpreter class. Accepts a `WorkflowFile` and a (subset of) `InterpreterOptions` object, throws if the `WorkflowFile` is not a valid workflow definition.

```
public async run(page: Page, params?: ParamType): Promise<void>
```

This method is the main entry point for the workflow execution. Using the *Preprocessor*'s methods, it initializes the workflow with new parameters and starts executing the workflow using the `Interpreter.runLoop()` private method.

This method also registers the `Interpreter.stopper` callback function for stopping the workflow execution.

```
public async stop(): Promise<void>
```

This method runs the necessary checks and stops the workflow execution. In case the checks fail - for example, the interpreter was not running any workflow - this method throws an exception.

```
async runLoop(p: Page, workflow: Workflow): Promise<void>
```

This private method represents the main loop of the workflow execution. Accepting the *Playwright* Page object and an initialized `Workflow` object as arguments, it keeps repeatedly looking for the applicable condition among the workflow's conditions and executes the corresponding actions.

It also keeps track of already executed actions and the execution history in general.

```
async getState(page:Page, workflow:Workflow):Promise<PageState>
```

This private method extracts the state representation from the current browser page context.

Receiving both the Page instance and the currently active workflow, the `getState` method extracts the smallest representation of the browser's state required for the current workflow execution. For example, when extracting the information on elements present in the page, the method first compiles all the selectors from the workflow, which are then used to query the page for the elements present.

```
applicable(where: Where, context: PageState,
                    usedActions : string[] = []): boolean
```

This private method compares the extracted page state with the conditions of the workflow.

Given a condition from the workflow being executed and the current page state, the `applicable()` method returns true if the condition is applicable to the current page state. Optionally it also accepts a list of names of actions that were already executed in the current workflow execution.

```
async carryOutSteps(page: Page, steps: What[]) : Promise<void>
```

Given a *Page* class instance and a list of actions from the current matched pair, this method carries out the actions on the given Page object.

The implementation of this method also contains the definitions of the custom actions and overrides for some specific *Playwright* methods.

A.2.2 `proprocessor.ts`

The *Preprocessor* class defined in the `preprocessor.ts` file implements the validation and static analysis of the input workflow definitions.

Methods

In the following subsection, all the methods of the `Preprocessor` class are described. The relations between those are also explained here, making it easier to understand the code. All of the following methods defined on the class are static, i.e. they can be called without creating an instance of the class.

`validateWorkflow(workflow: WorkflowFile): any`

This method validates the given workflow definition based on the syntax definition from the section 2.2 Format Design. If the object passed is a valid workflow definition, it returns null. In case of a syntax error in the workflow definition, it returns an error message describing the problem.

`getParams(workflow: WorkflowFile) : string[]`

For a parametrized workflow, this method returns the list of the parameter names used in the workflow. Internally, it performs a recursive search for the specific parameter structure.

`extractSelectors(workflow: Workflow) : SelectorArray`

Given a workflow definition, this method extracts the list of the string selectors used in the entire workflow. This is used for the state matching in the `Interpreter.runLoop()` method.

`initWorkflow(workflow: Workflow, params?: ParamType) : Workflow`

Initializes the workflow with the given parameters. Performs checks on the provided arguments and throws an exception if the parameters are not valid - the workflow parameters and the provided object do not match, for example. Also transforms the `{$regex: "regex"}` objects into regular JS regular expressions.

A.2.3 `types/`

The `types/` directory contains various TS typings used in the `wbr-interpret` package.

A.2.4 `utils/utils.ts`

A file containing various utility functions used in the `wbr-interpret` package.

Methods

`arrayToObject(array : any[]) : any`

Converts an array of scalars to an object with items of the array as keys. All the values are empty arrays.

A.2.5 utils/logger.ts

The `logger.ts` file exports the `logger` function, used for logging messages across the `wbr-interpret` package.

Type Aliases and Enums

`Level`

The `Level` enum defines the different log levels. The number values assigned to the different levels are used for assigning different colours to the messages and correspond to the ANSI Escape Codes.

Methods

`logger(message: string | Error, level: Level)`

The exported logger function for logging messages. Prepends every message with a timestamp and sets the message colour based on the value of the `level` parameter.

A.2.6 utils/concurrency.ts

Defines a generic `Concurrency` class for managing the concurrency of multiple long-running jobs running in parallel.

Methods

`constructor(maxConcurrency: number)`

A constructor for the `Concurrency` class. Accepts a `maxConcurrency` parameter, which defines the maximum number of jobs that can be running in parallel.

`addJob(job: () => Promise<any>) : void`

Passes a job (a time-demanding async function) to the concurrency manager. The time of the job's execution depends on the concurrency manager.

The passed function is guaranteed to be called sometime in the future.

`waitForCompletion() : Promise<void>`

Waits until there is no running or a waiting job. If the concurrency manager is idle at the time of calling this function, it waits until at least one job is completed.

Returns a `Promise`, which gets resolved after there is no running or an awaiting job.

`private runNextJob() : void`

A private method taking a waiting job from the queue and processing it. Once the job is completed, it calls the `runNextJob()` method until there are no more jobs in the queue.

A.2.7 `browserSide/scraper.js`

File containing browser-side code for the `wbr-interpret` specific actions (`scrape`, `scrapeSchema` and other.

Methods

`getBiggestElement(selector)`

Finds the largest element (based on its area) in the DOM tree that matches the given selector. Currently unused.

`getSelectorStructural(element)`

Returns the structural CSS selector of the given element. The structural selector describes a path to the element from the root element, based on the tag names. The returned selector is not unique, as this method is used to find repeating sibling elements on the page in the `scrapableHeuristics` method.

`scrapableHeuristics(...)`

An method implementing an heuristic for determining the most probably interesting elements on the page. The "value" of the element is determined by amount of similar looking elements on the same page.

`scrape(selector)`

Returns the text content of the elements targetted by `scrapableHeuristics`.

`scrapeSchema(selector)`

Given a set of element lists, this method matches the elements from the individual lists with each other based on their common ancestors. This is particularly useful for scraping incomplete tables of elements, where certain columns are not fully populated.

A.3 `wbr-editor` code documentation

This attachment contains in-depth documentation of the `wbr-editor` module, with detailed descriptions of the classes and methods used in the `wbr-editor` module.

The project is described in a directory-by-directory manner, as the directory structure clearly outlines the project's structure and groups components based on their locality inside of the application.

A.3.1 `src/App.tsx`

The root file of the `wbr-editor` application. This file defines the application entry point and combines the main application components. It also imports the CSS files used in the application.

A.3.2 `src/Application/`

The root directory of the `wbr-editor` application.

`Modal.tsx`

The file `Modal.tsx` describes the invitation modal element that is displayed when the user first accesses the application. The file upload and workflow validation logic is implemented here.

A.3.3 `src/Application/Reusables`

Folder containing small, generally reusable components. While the name might be a bit misleading, as all the *React* components are by design reusable, this folder contains the most general types of components usable throughout the whole project.

`Button.tsx`

The `Button` component is a simple *React* button with a label, icon and a click handler. It is used throughout the project to create button elements.

`Controls.tsx`

The `Controls.tsx` file contains control elements with less independence than the button mentioned above.

The file contains the `<Select>` component, representing the collapsible dropdown menu. It also contains the `<DeleteButton>` component, rendering as a small red button used for expressing the intent to delete.

A.3.4 src/Application/WorkflowEditor

The `WorkflowEditor` contains the elements that allow the user to create and edit the workflow definition files.

`WorkflowManager.tsx`

The `WorkflowManager.tsx` file contains the highest-level logic for the workflow definition files editing. The `<WorkflowManager>` element is the main component of the workflow editor part of the application. As the only *React* component in the application, `<WorkflowEditor>` holds the current state of the edited automation.

Using the generic `HistoryManager` class, it also maintains the history of the workflow definition files for the undo/redo operations.

`WorkflowEditor.tsx`

The `<WorkflowEditor>` component defined in the `WorkflowEditor.tsx` file handles the workflow definition files editing on a lower level than the above described `WorkfowManager` component.

This component also introduces the *React Contexts* for sharing the global state of the current actions made in the workflow editor (e.g. collapsing the pairs for better visibility).

A.3.5 src/Application/WorkflowEditor/Components

The `Components` directory contains the main building blocks of the workflow editor.

`Where.tsx`

The `Where.tsx` file contains the `<Where>` component, which is used to render the recursive condition structures inside the workflow definitions.

`What.tsx`

The `What.tsx` file contains the `<What>` component, which renders the list of actions inside the workflow definitions. The action list editor implements drag&drop behavior using the `react-dnd` library.

`Pair.tsx`

The `<Pair>` component defined in the `Pair.tsx` file combines the `<Where>` and `<What>` components to create an condition-action pair. This component is draggable (implemented using the `react-dnd` library) and can be collapsed and expanded using the outer contexts.

`DropZone.tsx`

The `DropZone.tsx` file defines the `<DropZone>` component, representing the spots where the `<Pair>` components can be dropped. This component changes appearance when being dragged over.

A.3.6 src/Application/WorkflowEditor/Editables

The files in this directory contain the definitions of editable input components used mostly by the `<What>` and `<Where>` components.

`EditableValue.tsx`

The `<EditableValue>` component defined in this file wrap a simple `<input>` HTML element in a *React* component, while enhancing it's functionality. It analyzes the `value` property of the `<EditableValue>` component and casts it to the most appropriate type.

`EditableHeading.tsx`

A component for displaying a heading with an editable value. The editing mode is enabled by double-clicking on the heading.

`EditableArray.tsx`

The `<EditableArray>` component defined in this file renders an array of values as a list of `<EditableValue>` components. If the `dynamic` option is set, the length of the array can be changed by the user.

`EditableObject.tsx`

The `<EditableObject>` component defined in the `EditableObject.tsx` file renders an flat JS object as a table of `<EditableValue>` components. If the `dynamic` option is set, the keys can be added by the user.

`RenderValue.tsx`

The `<RenderValue>` component combines the functionality of the previously mentioned `<EditableValue>`, `<EditableArray>` and `<EditableObject>` components. Given a value, this component renders it as an appropriate *React* component, based on its type.

A.3.7 src/Application/WorkflowEditor/Utils

This directory contains various helper functions and global context exports.

`GlobalStates.tsx`

This file defines the *React* contexts used in the application to share the global state of the workflow editor.

`UpdaterFactory.tsx`

This file implements several factory methods for creating updaters - functions used for updating the immutable *React* state. These are used mainly in the `<Where>` and `<What>` components to ensure uniform interface and consistency.

`utils.ts`

A file containing helper TS methods.

A.3.8 `src/Application/WorkflowPlayer`

This directory contains the components and classes responsible for the frontend for the remote workflow player.

`Screen.tsx`

This file contains the `<Screen>` component, which is the main component of the workflow player, representing the virtual "screen" for the remote browser view. Furthermore, it contains the `ScreenControls` class, which provides methods for updating the screen component.

`Console.tsx`

Contains the `<Console>` component, representing the console for logging the data from the remote browser view. The file also contains the `ConsoleControls` class, providing methods for updating the console component.

`Player.tsx`

Implements the `Player` component combining the `Screen` and `Console` components. Also exports the `runWorkflow()` function, which implements the remote browser communication, handling the data from the remote browser and updating the components.

A.4 `wbr-cloud` API documentation

This attachment documents the `wbr-cloud` server REST API.

`POST /api/performer`

The REST API endpoint for passing the automation to the *Editor* application backend server for execution.

The body of the request must contain a JSON object with the fields `workflow` containing the workflow definition object and `parameters`, containing the optional parameters for the current workflow.

Returns a JSON object with the fields `url` containing the unique identifier of the workflow run, boolean `status` signalizing whether the workflow started successfully and `message` containing the status message.

`POST /api/performer/:id`

The REST API endpoint for management of the automation running in the *Editor* application backend server.

The body of the request must contain a JSON object with the field `action` describing the desired action to be executed. Currently, the only valid action is `stop`, stopping the specified workflow run.

Other endpoints

Because of legacy reasons, the `wbr-cloud` server also provides other endpoints, following the REST API practices. These endpoints are however, not used by the `wbr-editor` frontend application and their functionality is not ensured to be stable, as they are no longer maintained.