BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

A TOOL FOR CREATING LOG MESSAGE PATTERNS

NÁSTROJ PRO TVORBU VZORŮ LOGOVACÍCH ZPRÁV

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **IGOR HANUS**
AUTOR PRÁCE

SUPERVISOR **Ing. ALEŠ SMRČKA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

145014

Institut:	Department of Intelligent Systems (UITS)
Student:	**Hanus Igor**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**A Tool for Creating Log Message Patterns**
Category:	Software analysis and testing
Academic year:	2022/23

Assignment:

1. Study the method of verifying programs at runtime. Learn about the Plogchecker tool developed at FIT BUT.
2. Design an interactive tool for the manual creation of log message patterns. Focus on the straightforward generalisation of messages for subsequent monitoring using the Plogchecker tool.
3. Implement the tool as a portable application. Select the implementation language and framework, taking into account the portability and maintainability criteria.
4. Evaluate the usability of the tool. Prepare a video demonstrating the functionality of the tool.

Literature:

- ČALÁDI, Filip. *Ověřování parametrických vlastností nad záznamy běhů programů*. Brno, 2022. Master Thesis (Czech). FIT BUT.
- E. BARTOCCI, Y. FALCONE. *Lectures on Runtime Verification*. Springer, 2018. doi: 10.1007/978-3-319-75632-5

Requirements for the semestral defence:
The first two points.

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

Supervisor:	**Smrčka Aleš, Ing., Ph.D.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	10.5.2023
Approval date:	2.5.2023

Abstract

The thesis aims to create a portable web application for processing logs using combinations of Grok patterns and regular expressions to create a pattern for individual log messages with the possibility of exporting them into YAML format that can be processed by the tool Plogchecker. The application was implemented using the React JavaScript library using the TypeScript language. Processing of individual regular expressions is achieved using the Oniguruma library, which is integrated into the application using WebAssembly. The reason for using the Oniguruma library was the incompatibility between regular expression compilers specified by the ECMAScript standard and compilers used for Grok patterns. Automated testing and user testing were conducted, and identified flaws were addressed.

Abstrakt

Cieľom práce je vyvtorenie prenositeľnej webovej aplikácie na spracovanie logov pomocou kombinácie Grok vzorova a regulárnych výrazov, za účelom vytvorenia vzoru pre jednotlivé záznamy logu s možnosťou exportovania do YAML formátu, ktorý je spracovateľný nástrojom Plogchecker. Aplikácia bola implementovaná pomocou JavaScript knižnice React použitím jazyka TypeScript. Spracovanie jednotlivých regulárnych výrazov je riešené pomocou knižnice Oniguruma, ktorá je integrovaná do aplikácie pomocou WebAssembly. Dôvodom použitia Oniguruma knižnice bola nekompatibilita medzi prekladačmi regulárnych výrazov definovanými štandardom ECMAScript a prekladačmi, ktoré sú využívané pre preklad Grok výrazov. Realizacia aplikácie bola podrobená automatizovaným testom a užívateľským testom, s opravou zistených nálezov.

Keywords

log, log processing, web application, Grok, regular expressions, Regex, Regexp, React, TypeScript, Oniguruma, WebAssembly

Kľúčové slová

log, spracovanie logov, webová aplikácia, Grok, regulárne výrazy, Regex, Regexp, React, TypeScript, Oniguruma, WebAssembly

Reference

HANUS, Igor. *A Tool for Creating Log Message Patterns.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Aleš Smrčka, Ph.D.

Rozšírený abstrakt

Zvyšujúca sa komplexnosť moderných počítačových systémov viedla k nárastu objemu logovacích dát generovaných týmito systémami. Účinná analýza logov je kľúčová pre detekciu a diagnostiku problémov a pre zlepšenie výkonu a bezpečnosti systému. Manuálne spracovanie veľkého objemu logov je však časovo náročnou úlohou.

Jedným z nástrojov, zaoberajúcim sa spracovaním logov v reálnom čase a detekciou špecifických udalostí v testovanom systéme, je nástroj Plogchecker. Tieto udalosti sú špecifikované kombináciou regulárnych výrazov a Grok vzorov, ktoré sú následne aplikované na jednotlivé záznamy v logu. V prípade, že záznam odpovedá vlastnostiam použitého vzoru, Plogchecker zaznamená podrobnosti o tejto udalosti. Aby bolo možné nástroj Plogchecker nasadiť, je potrebné overiť, či vytvorené vzory korešpondujú s očakaváným záznamami v logu.

Cieľom tejto práce je teda vytvorenie prenostieľnej webovej aplikácie na filtrovanie logov pomocou kombinácie Grok vzorov a regulárnych výrazov v reálnom čase. Aplikácia je vytvorená v JavaScript knižnici React za pomoci jazyka TypeScript. Vývoj aplikácie bol rozdelený do troch častí, ktorými sú návrh, implementácia a následné testovanie aplikácie.

Prvotný návrh aplikácie bol vytvorený pomocou online nástroja Figma. Pred implementačnou časťou bolo potrebné vybrať vhodný textový editor pre zobrazovanie vloženého logu a pre filter, slúžiaci na vytváranie jednotlivých vzorov. Knižnica použitá pre textový editor je Codemirror 6. Zvolená bola z dôvodu dlhodobej podpory a poskytovania možností vytvárania rozšírení, slúžiacich na farebné označovanie textu vo filtri a taktiež označovanie záznamov v logu, ktoré odpovedajú použitému vzoru. Ďalším zásadným bodom v návrhu bolo nájdenie spôsobu vyhodnocovania regulárnych výrazov, ktoré nie sú podporované JavaScript štandardom ECMAScript, nakoľko Grok vzory používajú primárne regulárne výrazy podporované PCRE knižnicou. Z tohto dôvodu bola použitá knižnica Oniguruma, podporujúca funkcionalitu väčšiny známych prekladačov regulárnych výrazov.

V implementačnej časti bolo potrebné integrovať Oniguruma knižnicu, ktorá je originálne vytvorená v jazyku C++, do TypeScriptu. Vďaka jazyku WebAssembly je možné použiť skompilovaný kód Onigurumy v rámci webovej aplikácie a následne s ním komunikovať pomocou knižnice Onigasm, ktorá ponúka rozhranie nad skompilovaným kódom Onigurumy.

Aplikácia bola testovaná pomocou end-to-end testov a podrobená užívateľskému testovaniu. Jednotlivé testy a ich kroky boli popísané v nástroji Cucumber a následne implementované pomocou knižnice Selenium, aby bolo možné simulovať reálne používanie aplikácie. Užívateľské testovanie sa následovne zameriavalo na celkovú použiteľnosť a pochopenie aplikácie. V rámci spätnej väzby z užívateľského testovania, boli opravené chyby v aplikácii a taktiež implementované niektoré požiadavky uživateľov, ktorí sa na testovaní podieľali.

A Tool for Creating Log Message Patterns

Declaration

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

.....
Igor Hanus
May 6, 2023

Acknowledgements

First I would like to thank my supervisor Ing. Aleš Smrčka, Ph.D. for his patience and guidance. Also I want to thank to my parents for continuous support and help throughout my studies.

Contents

Chapter 1

Introduction

The increasing complexity of modern computer systems has led to a corresponding increase in the volume and variety of log data generated by these systems. Effective log analysis is crucial for detecting and diagnosing issues and improving system performance and security. However, dealing with large volumes of logs can be a challenging and time-consuming task. In response to this challenge, many log analysis tools have been developed.

This bachelor's thesis presents a web application, named LogReaper, allowing users to filter logs in real-time using Grok patterns, regular expressions, and their combinations. Every created string of regular expressions can be saved and later edited or used to filter the inserted log. To ensure that the web application is accessible to users unfamiliar with Grok patterns, they are able to create them simply by highlighting a part of a log's message and choosing one of the Grok patterns offered in a popup window.

The application's primary purpose is to give users a better insight into whether the created log message patterns match the expected log messages before being used to define individual events in Plogchecker. As these events are defined in the YAML format, users can export selected log message patterns and export them in YAML format.

The next Chapter 2 explains how runtime verifications use regular expressions to detect particular behavior of the system under test. Then it explains how Grok patterns are constructed and used inside the application. It also compares the application to other available similar tools online. Chapter 3 shows how the application was developed from the low-fidelity prototype to the final design of individual components. Chapter 4 describes the technologies used for the application. Chapter 5 explains the implementation of individual components and core functionalities for Grok pattern and regular expression processing. Finally, the Chapter 6 explains the testing process of the application. The tests include end-to-end tests using web automation to simulate the user's interaction with the application through a browser and user testing.

Chapter 2

State of the Art

The state of the art chapter of this bachelor's thesis focuses on using regular expressions and Grok patterns to specify system behavior in runtime verification. We provide an overview of the current study on regular expressions, highlighting their limitations when processed by different regular expression engines. Then it demonstrates the possibility of removing these limitations using the Oniguruma regex library. The next focus is on Grok patterns and the benefit of using them to process system logs instead of regular expressions. Lastly, this chapter compares LogReaper with existing tools that work with regular expressions and Grok patterns to process logs.

2.1 Runtime verification

Runtime verification is a dynamic software analysis approach that analyzes the system under test (SUT) as it executes, observing the results of the execution and using those results to find unwanted behaviors [8].

Runtime verification relies on specific properties that the execution of the program should not violate. Some properties, like a lack of data races in a concurrent program, are universal and can be checked automatically. Other properties, like the specification for a proprietary library, are custom to a specific application or purpose. Runtime verification can check universal properties automatically, requiring no development input, and check any custom properties expressed formally by developers.

When considering how to check whether the runtime behavior of a system conforms to some specification, there are three necessary steps to be taken:

- Specifying (Un)Desired System Behaviour.
- Generating a Monitor from a Specification.
- Connecting a Monitor to a System.

This project focuses on the first step as it is used to efficiently create and manage regular expressions to define specific events inside a property to specify the behavior of a particular system through its logs. Plogchecker then uses these properties, as described in the Section 2.2.

2.2 Plogchecker

Parametric Log Checker (Plogchecker) is a tool for monitoring various plaintext logs and creates a report in case of a violation or satisfaction of defined properties.

These properties are specified in a single YAML file. Plogchecker then parses this file and creates runtime monitors based on the parsed properties [17].

The grammar of these properties is defined using the Bakus-Naur form (BNF). The grammar specifies three main parts, properties, events, and constraints. An example of a property is shown in Figure 2.1. A property file must contain at least one of the two sections. These sections are `properties` and `bad_properties`. The `properties` section specifies a sequence of events that users would want to see in the execution trace of the monitored system. On the other hand, the `bad_properties` section determines the unwanted behavior of the monitored system. The `events` section defines the events used to match the system's events of the SUT. The event format is defined as follows:

```
(id:event_value)
```

Where `id` is a specific identifier of the event and `event_value` defines a value that is used to match events of the SUT. These values are specified using regular expressions, parametrized regular expressions, or Grok patterns.

Finally, the constraints section defines parametric constraints over individual event parameters. One can specify the value of the operand as a parameter or constant. The parameter value is a reference to some event parameter. On the other hand, the constant parameter can be specified either as a number or string value.

```
properties:
  p1: "Open Use* Close"
bad_properties:
  p2:  "Open Close Use"

events:
  Open: "open %{WORD:p1}"
  Use: "use %{r(\b\w+\b):p1}"
  Close: "close \b\w+\b"

constraints:
  -- "Open.p1 = Use.p1"
  -- "Open.p1 = Close.p1"
  -- "Use.p1 = Close.p1"
```

Figure 2.1: Example of a Plogchecker property file.

2.3 Regular expressions

A regular expression (also known as regex or regexp) is a sequence of characters that creates a pattern describing a certain amount of text [13]. For example, `„test"` is a valid regular expression. It is the most basic form of regular expression pattern, matching the string literal `test`.

The pattern `(%\{[A-Z0-9_]+(?::[A-Za-z0-9_]+)?\})` is a more complex one. This pattern matches text corresponding to a `Grok pattern`'s syntax. The pattern defines that the piece of text starts with the characters `„%{"`. Then it is followed by a random combination of alphanumeric characters, which define the `Syntax` of the Grok pattern, explained in Section 2.4, optionally following by the character `':'` and the `ID` (see Section 2.4) of the Grok pattern, which again, can consists of a combination of alphanumeric characters or a character `'_'`. And at the end, it matches the closing bracket `„}"`.

A `match` is the piece of text or sequence of bytes or characters that the pattern was found to correspond to by a regex engine. These regex engines are commonly used in user input validation on webpages or „search and replace“ functions used by many text editors.

Regular expression engine

A regular expression engine (regex `flavor`) receives the regular expression pattern and the text we want to process, as seen on Figure 2.2. In programming, regular expressions are represented as a string. This regular expression is typically compiled into a form that can be executed efficiently on a computer, and the engine performs the actual matching [12]. The output from a regular expression engine can have many forms, including an array containing matched substrings, the position of matched substrings relative to the whole text, or a `boolean` value indicating if any substring matching the regular expression was found.

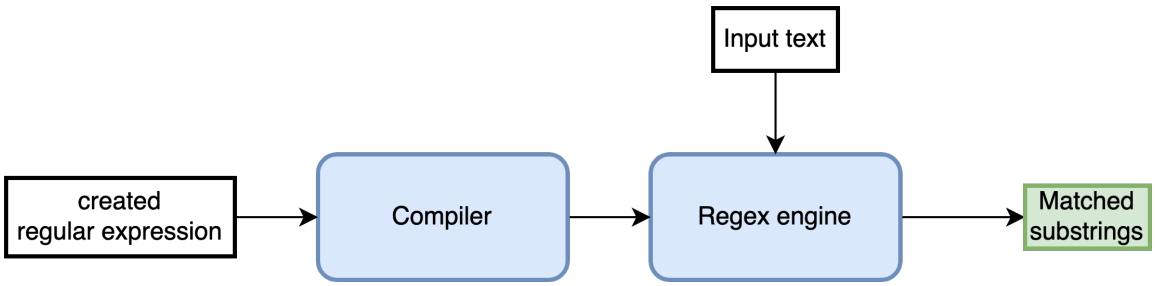

Figure 2.2: Processing of a regular expression.

When creating a regular expression, it is good to know what operations the regex flavor supports, as programming languages typically adopt one of these flavors. Programmers should be aware of these differences when migrating from one flavor to another and adequately test the migrated regular expression. Another option is to use a regular expression library, like `Oniguruma`.

Oniguruma and Onigasm

`Onigasm` (OnigurumaASM)[1] is a `WebAssembly` port of the Oniguruma library.

`WebAssembly` is a binary format designed to run in web browsers and other environments and allows code to be written in languages such as C, C++, and Rust.

A port refers to the process of compiling software or programming code into a format that can be executed within a `WebAssembly` environment.

Using a `WebAssembly` binary file of the Oniguruma library can let the developer parse every regular expression through the program compiled inside the binary file, bypassing the limitations of the *ECMAScript* standard. The program's methods can be called using the `Onigasm` library interface.

Using Oniguruma in JavaScript

The main issue related to the regular expression flavors is that they all support different features [10]. As the application is implemented using JavaScript, it uses regular expression flavor specified by the ECMAScript standard [14]. In contrast, Grok uses Oniguruma regular expression library, which supports features of regular expression flavors that traditionally exist in different languages. These flavors include Perl Compatible Regular Expressions (PCRE), Perl, or Python. The difference between these flavors can be seen on Table 2.1.

This difference leads to situations when ECMAScript does not support regular expressions compatible with Oniguruma, but every regular expression supported by *ECMAScript* engine is also compatible with Oniguruma. Thus a transpiler between different regular expression engines or an external web assembly file of Oniguruma library is required.

Feature	Perl	PCRE	Python	ECMAScript
(?i) (case insensitive)	YES	YES	YES	/i only
(?>regex) (atomic group)	YES	YES	NO	NO
\z (end of string)	YES	YES	\Z	NO
(?m) (^and $ match at line breaks)	YES	YES	YES	NO

Table 2.1: Example of feature support amongst different regex flavors.

[1] https://github.com/zikaari/onigasm

2.4 Logstash Grok

Logstash Grok is a regular expression dialect that supports reusable aliased expressions [1]. Grok uses Oniguruma regular expression library, which means that any regular expression is valid in Grok. It is handy when dealing with log data because logs often contain a large amount of unstructured data, making it challenging to extract meaningful information.

Overall, Logstash Grok is a powerful tool that can help users extract valuable insights from their log data, which can be used to improve system performance or troubleshoot specific issues.

Grok patterns

Grok comes with a considerable amount of predefined Logstash Grok patterns. Grok uses these patterns to extract meaningful information from log data, such as timestamps, error codes, or URLs. These patterns simplify regular expressions by adding an alias to them. That means instead of creating an extended regular expression, for example, a regular expression defining an IP address, users can use a Grok pattern `%{IP}`. This Grok pattern is later translated into a regular expression that matches IP addresses in a given text. Grok also supports the nesting of Grok patterns. In other words, the Grok pattern `%{IP}` holds the regular expression consisting of the other two Grok patterns, which are `%{IPV4}` and `%{IPV6}` divided using logical OR, in regex, defined as |, creating a regular expression with a value of `%{IPV4}|%{IPV6}`.

Grok supports multiple forms of Grok patterns, which are:

- %{SYNTAX}
- %{SYNTAX:ID}
- %{SYNTAX:ID:TYPE}

SYNTAX defines the pattern's name that ties to an actual regular expression that will be used to match the text, for example the Grok pattern defined as `%{IP}` has a name `IP`, and this name ties to a regular expression `%{IPV4}|%IPV6`.

ID is an identifier given to the pattern. This identifier is used when extracting matching data from a given text and creating a JSON file, where the properties' names are set to the values of Grok pattern IDs. If a Grok pattern of value `%{IP:srcip}` is created and matched to a value of "127.0.0.1"inside of a given text, then the JSON value would look like `{"scrip": "127.0.0.1"}`.

TYPE defines the data type into which the matched data will be cast. Using the Grok pattern `%{BASE10NUM:number:int}`, where every number matched by the regular expression that is tied to the `BASE10NUM` name inside a given text will be cast into the data type `int`.

2.5 Similar applications

There are many tools for building strings of regular expressions, but not so many when it comes to support of building using Grok patterns. Many projects using Grok patterns are usually just libraries in different programming languages. Some add extended functionality to an existing application, such as support of Grok patterns in text search. However, some applications share similar properties with this project, especially matching Grok patterns to inserted log messages.

Grok Constructor

Grok Constructor[2] is a set of tools that lets users work with Grok patterns. One of these tools is called *Incremental Construction*. The incremental construction of Grok patterns lets users create a custom Grok pattern step by step, matching all messages inside a given log simultaneously. While testing the application, this tool seemed faulty as every input resulted in an error message stating that a bug had happened.

Another tool is called *Matcher* (see Figure 2.3). The tool's functionality is closest to this project's functionality and is also shown on the image displaying Grok Constructor. In this tool, users can insert a log they want to analyze, and in the other input, they can insert individual grok patterns that will get matched to individual log messages. The tool only lets users create one string of regular expressions at a time, even though the input size would indicate that multiple strings can be created simultaneously. The tool also lets users choose Grok sets they would like to use and checks for incorrect regular expressions. The overall experience of using this tool could be more optimal. When users would like to process an extensive log, and the string of regular expressions matches every line, it will create an HTML table that creates a row for every line, not even numbering them. Also, the description of individual inputs is somewhat complicated, and it needs to be more comprehensible what their purpose is.

[2] https://grokconstructor.appspot.com/

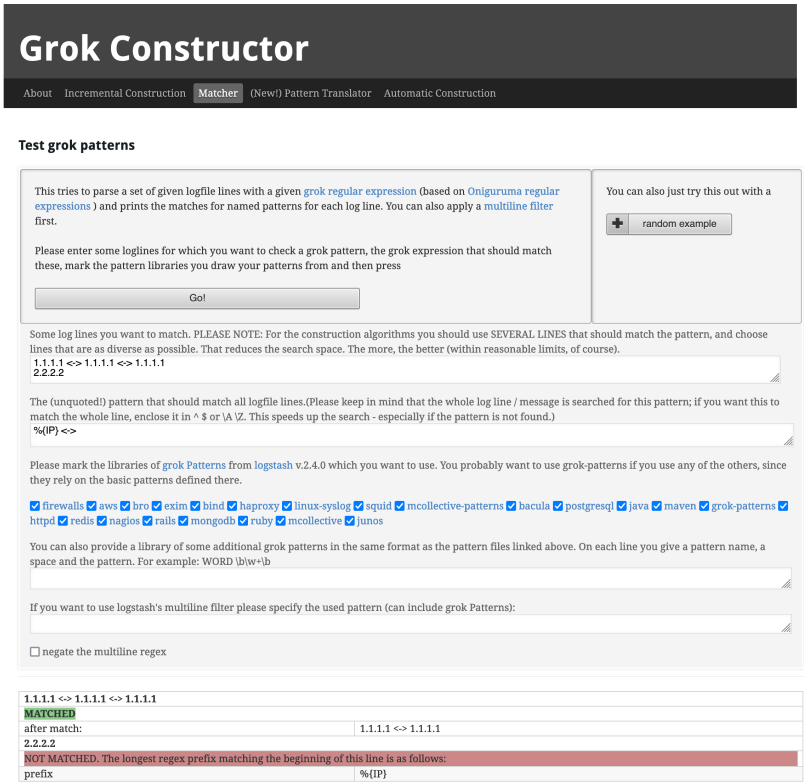

Figure 2.3: *Grok Constructor's Matcher* tool.

The following tool, *Pattern translator*, translates the log4j log format into a string of regular expressions.

Furthermore, the final tool *Automatic constructor* (see Figure 2.4) aims to give users recommended Grok patterns based on the inserted text. After executing the tool, users are given multiple options they can choose from, again creating a very long HTML table, making users search through tenths or even hundreds of recommended strings of regular expressions. Every Grok pattern included in these strings is displayed as a dropdown element, adding even more combinations to the existing recommendations. If users want fewer recommended strings to choose from, they would have to unclick every checkbox selecting a Grok pattern set, as they are all selected by default. While deselecting these patterns, users can accidentally misclick on the set's name, redirecting them to another webpage displaying the Grok patterns included in the clicked set.

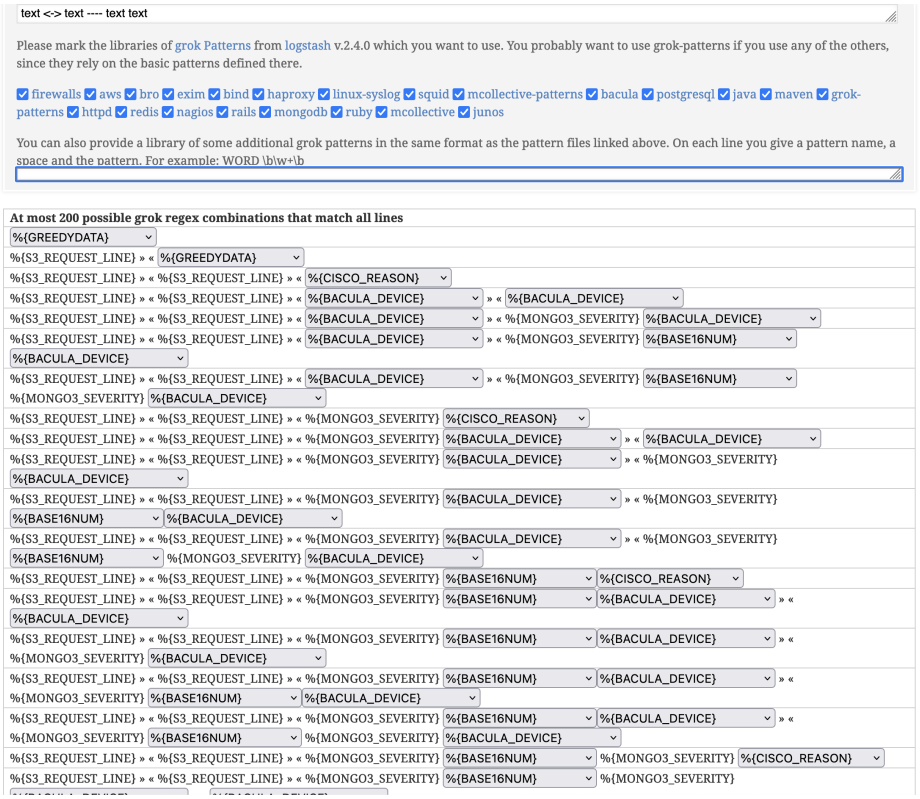

Figure 2.4: *Automatic constructor* tool of *Grok Constructor* application.

Grok Debugger

Grok debugger[3] (see Figure 2.5) lets users create a string of regular expressions by combining regular expressions and Grok patterns. It uses `CodeMirror 5` for its text windows. Users can add custom Grok pattern sets and create new Grok patterns or download Grok pattern sets from provided URL. The input for creating a string of regular expressions uses simple grammar to highlight detected Grok patterns.

The text window with an inserted log is updated in real-time as the input for creating a string of regular expressions is updated. The created string is matched with every message in the inserted log. Matching substrings are highlighted with yellow if it matches an inserted Grok pattern or green if it matches an inserted regular expression. In order to receive a valid output, every Grok pattern has to have its ID. Based on this ID, an attribute in the generated JSON is created with the value of matched substring.

Another helpful feature is the possibility of sharing the workspace using a URL. The only problem is that the length of the URL is browser limited and may only sometimes be possible to use.

The downsides of *Grok debugger* are the inability to work with multiple strings of regular expressions. Users have to create a single string and copy it elsewhere in case they want to reuse it. The highlighting method may not be ideal for extensive logs with many identical messages, as the output JSON will be full of duplicated data for every identical message.

[3] https://grokdebugger.com/

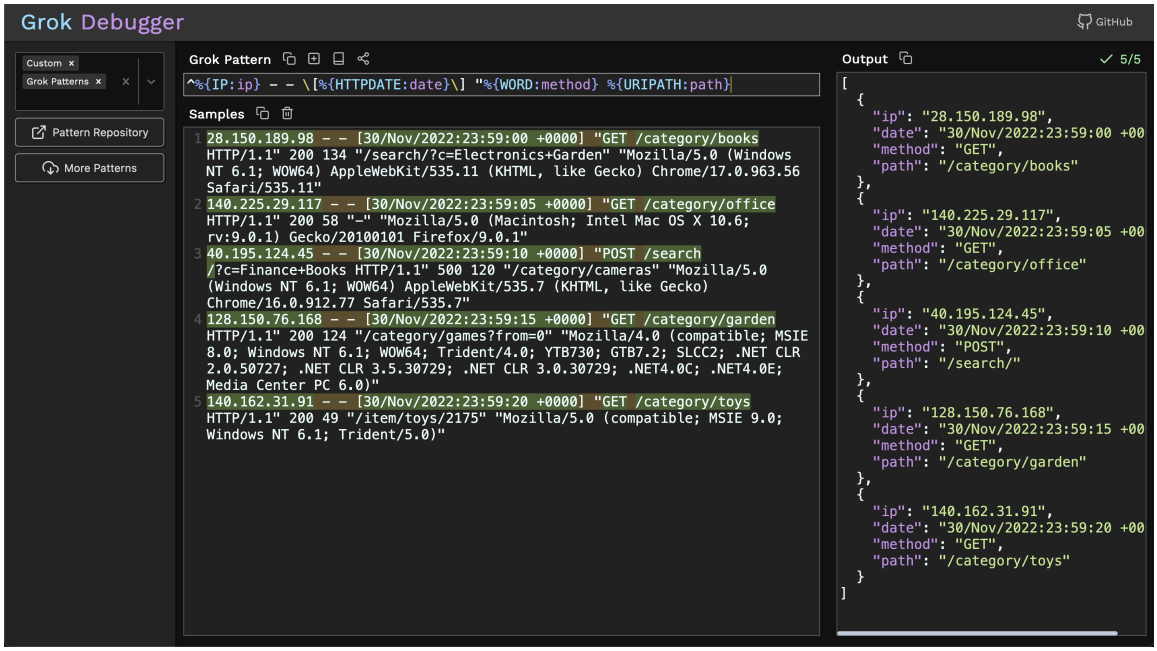

Figure 2.5: Main window of *Grok debugger*.

Comparison between applications

The Table 2.2 shows a comparison between the functionalities of LogReaper and other two available applications that work with Grok patterns and regular expressions.

Feature	LogReaper	Grok debugger	Grok Constructor
Grok syntax highlight	yes	yes	no
Filtering text	yes	no	no
Hihglighting matched text	only hitboxes	yes	no
Selecting grok pattern sets	yes	yes	yes
Custom patterns	can be added trough code	yes	no
Working with multiple strings of regexes	yes	no	no
Error notifications	yes	no	yes
Responsible	yes	yes	no
Text editor	CodeMirror 6	CodeMirror 5	HTML Textarea
Grok recommendations	yes	yes	yes
Line numbers	yes	yes	no
Generating message pattern based on input	no	no	yes
Export of created message patterns	yes	no	no
Real-time log filtering	yes	yes	no

Table 2.2: Feature comparison between applications.

Chapter 3

Application design

The application design process is a critical component of software development that heavily influences the success of an application. The design impacts usability, reliability, and overall performance, underscoring the need for a meticulous approach to designing software applications. It is crucial to have a comprehensive plan to ensure the final product aligns with the target audience's expectations.

When designing an application, creating a set of functional requirements is essential in order to be able to design individual components based on their functionality. When requirements are set, a mockup is created. A mockup gives the developer a better idea of what the final design may look like. Several changes have been applied when implementing the final design, while the application's functionality remains unchanged.

3.1 Functional requirements

Every design must satisfy the target users' specific functional needs. It is crucial to create a set of requirements that closely define the interaction between the end user and the application itself. These requirements are represented as a set of features that developers must implement to enable users to achieve their goals [2]. The features for the application are defined as in Table 3.1.

Feature	Description
Inserting log	The user is able to insert a log that they want to examine.
Log message pattern creation	The user can combine regular expressions, parameterized regular expressions and Grok patterns to create a single log message pattern.
Text filtering	The text is filtered real-time, when a valid expression is passed in the input filter or a log message pattern is selected.
Grok pattern recommendations	When text inside of the loaded log is highlighted, the user is given multiple options for Grok patterns they can use. Upon selecting a pattern, it is then moved into the input filter component. User receives recommendation from the selected Grok pattern sets. Another possibility is to use the selected text as a regular expression.
Managing log message patterns	The user can change the name of created patterns, use them to filter text and export them.
Log message (row) highlight	When a single log message pattern or their combination is selected, every matched message inside of the inserted log will get a hit-box. Hit-box is a red line at the start of every matched message. This line indicates, that multiple log message patterns matched the exact same log message.
Pattern export	Once the user is satisfied with their created log message patterns, they are able to export them as individual events used for Plogchecker in YAML format.
Grok set selections	The app lets users choose Grok sets which they want to use. These Grok sets contain a collection of Grok patterns with a specific use. These sets are used to create recommendations for highlighted text, as they provide individual Grok patterns.
Filtered log messages display	The total amount of messages inside of the inserted log is displayed when the app is loaded. Once the user starts filtering the log, they will be informed how many messages have been matched out of all messages.
Application's fitler and edit mode	The application works with two modes. These modes are editing mode and filtering mode. When the user starts applying filters to the inserted log, the app goes into filtering mode, letting the user working with the copy of the original log. If the user wants to edit the document, they can simply remove all active log message patterns and delete the contents of the input filter.

Table 3.1: Application's features.

3.2 Application feature overview

This section provides a detailed description of the individual features that are defined in Table 3.1. By assigning these features to their specific roles, we can better understand their function in the application's components.

Editing modes

The application is divided into two editing modes. These modes define whether the user can edit the loaded log or can only filter it. These modes prevent the user from inserting text into the already actively filtered document, as the filtered text is just a filtered copy of the original document. These modes also maintain the original document's consistency.

The user can freely edit the inserted log only when no filters are applied. After the user starts filtering the inserted log, the application switches to filtering mode, preventing the user from further editing the inserted log. All filters need to be disabled in order to return to the edit mode.

Text filtering

The filter comprises two sub-filters, as shown on Figure 3.1, both functioning independently from each other. When the user passes a log through the filter, the filter will combine these two sub-filter expressions creating one single regular expression using a logical OR. This created filter expression is then applied to each message of the inserted log and removes the log messages that do not match.

The filter consists of two sub-filters:

- User input filter
- Message pattern filter

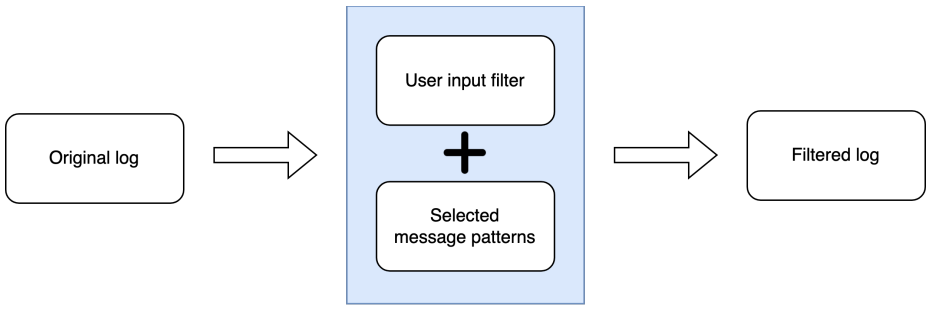

Figure 3.1: Filter combination.

User input filter

The user input filter is an input to which the user can type the text as a combination of regular expressions and Grok patterns, creating a string of regular expressions. After a slight delay, the filter will update the log inside the main text window using this created string of regular expressions. When the user is satisfied with the created string of regular

expressions, they can move it to the message pattern toolbar window and use it inside the log message filter.

Log message pattern filter

The log message pattern filter is the second of the sub-filters. This filter is a combination of user-selected log message patterns. By selecting these message patterns, an event that will match every whole message of the log to the selected message patterns is fired. A log message pattern is created by exporting the string of regular expressions from the `input filter`.

Managing log message patterns

The user can set a name for each log message pattern. Another possibility is to edit the chosen log message pattern. Clicking the edit arrow will export the single message pattern as a string of regular expressions into the input filter. The user can also create a message pattern filter by selecting individual message patterns. This will add hit-boxes to individual matched log messages. The hit-boxes get darker as more log message patterns match the exact log message. Once the user is satisfied with the filtered texts, they can then export the selected log message patterns into a YAML format, which can then be used to define individual events in Plogchecker property file.

Each message pattern in the message pattern toolbar displays the individual regular expressions, parametrized regular expressions, and Grok patterns used to create the string of regular expressions inside the input filter.

Grok recommendations

A popup will appear when highlighting a specific part of the text on a single line of the inserted log, suggesting Grok patterns that match the highlighted text. From this suggestion, the user can choose the Grok pattern he was looking for, and by clicking it, this pattern will be inserted at the end of the input filter text. If the user is unsatisfied with the available Grok patterns or wants it to be used as a regular expression, they can choose the option 'Use as regex' and copy the highlighted text at the end of the input filter text.

List of available Grok sets

In order to work with only specific collections of Grok patterns, the user can select specific Grok sets they want to use. Every Grok pattern set has a specific use case. For example, if the user wants to match error or warning messages of java, they should use the 'Java' Grok pattern set. Selecting only needed Grok pattern sets fastens the creation of strings of regular expressions. Grok patterns from different sets can match the exact text the user highlights, causing them to search through more patterns offered to them through the popup that may be unrelated to his work.

3.3 Initial design in Figma

Figma[1] is a web-based design and prototyping tool. It provides a platform for designers to create, edit, and share their work with other team members. Figma supports vector networks, constraints, and auto-layouts, making it a popular choice for creating responsive designs and user interfaces. Additionally, it has a wide array of plugins and integrations with other tools.

The goal was to create an initial prototype of the main application window in Figma. The application is intended to be used on larger-scaled screens (laptops and desktops). For that reason, the device chosen for the prototype was MacBook Air.

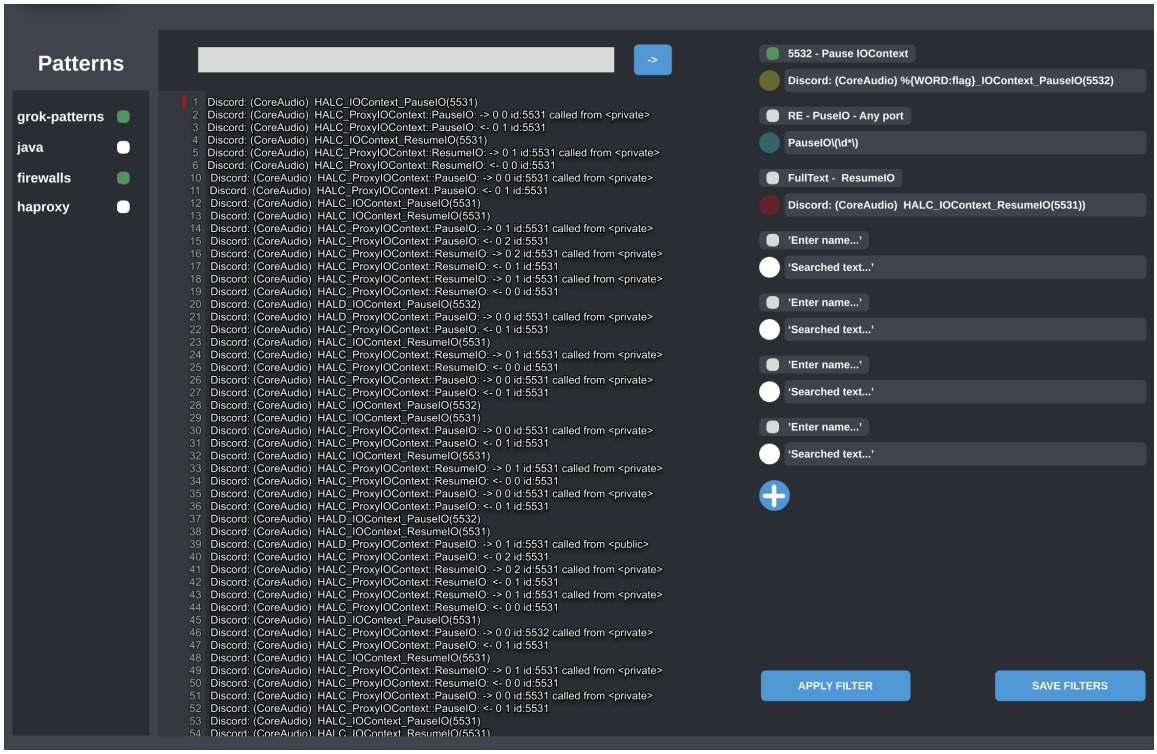

Figure 3.2: Initial design.

Figure 3.2 shows an initial design of the main application window. The middle part consists of the input filter, located at the top of the application window, and the main text window for the log the user wants to filter. The user can see the created message pattern on the right side of the application window. Individual message patterns have a checkbox. When selecting this checkbox, the user can use this message pattern in the filter by clicking the 'Apply filter' button. In the initial design, every message pattern has a color assigned to it. These colors are used to highlight individual log messages that match this message pattern. The user can also change the name of the message patterns, create new ones and export them if they want to load them to a new application instance.

[1] https://www.figma.com/

Color scheme

The inspiration for this palette (see Figure 3.3) comes from widely used text editors, such as Sublime Text [2], which was the primary source of inspiration. The main body of the application is displayed in shades of grey based on the popularity of the „Dark Mode.“ More bright colors are chosen to contrast the application's dark background. The palette was created using the online tool Coolors[3].

Figure 3.3: Color palette.

[2] https://www.sublimetext.com/
[3] https://coolors.co/

3.4 Final design

Specific changes have been made while implementing the initial design created in Figma (see Figure 3.2). The functionality remains the same, while the layout and colors of the components have changed. The purpose of these changes is to enhance the user's experience while using the application. This section shows the most notable changes in particular application components and the final design of the whole application window, shown on Figure 3.4.

The whole layout of individual components is divided into visible blocks and no longer appears to be merged inside of a one big window. The light blue color of individual buttons has been changed to a darker shade of blue to match the rest of the components, which are also dark colored. In the application's state bar, a notification component for error messages and a display of the current mode have been added. Another important component that has been added, is a popup that displays recommended Grok patterns based on the highlighted text.

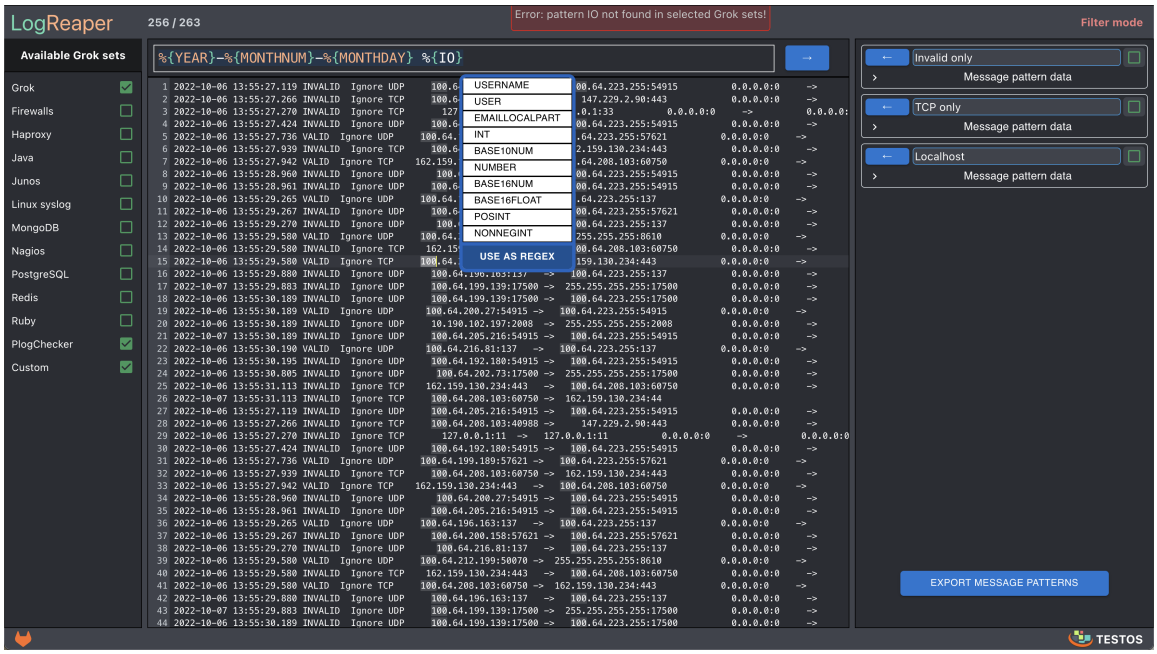

Figure 3.4: The final design of the application.

Input filter

The user input filter has gone through three iterations of design changes as it can be seen on Figure 3.5. In the first iteration, the input filter consisted of a simple input without any syntax highlighting.

In the second iteration, a pattern builder was added. A block was added to the pattern builder when the user created a single Grok pattern or a regular expression. This block was colored based on the type of the created expression. The `RE` type indicated that a valid regular expression was created, and the `GROK` type indicated that a valid Grok pattern was created. The user could also add a new block through the pattern builder by clicking the `+` button. The same evaluating process was applied when confirming the creation of the new pattern. The main downside of this approach was that the user had to create every pattern individually by pressing the Enter key every time the user created a single pattern inside the input filter, significantly slowing down the creation of a log message pattern.

The third iteration combines the two initial iterations combining the efficiency using a simple input window with a highlight of curly brackets to distinct a Grok pattern from the remaining parts of the string of regular expressions. The functionality missing from the second iteration is the color distinction of valid Grok patterns. An error message is shown in the third iteration whenever an invalid Grok pattern is created.

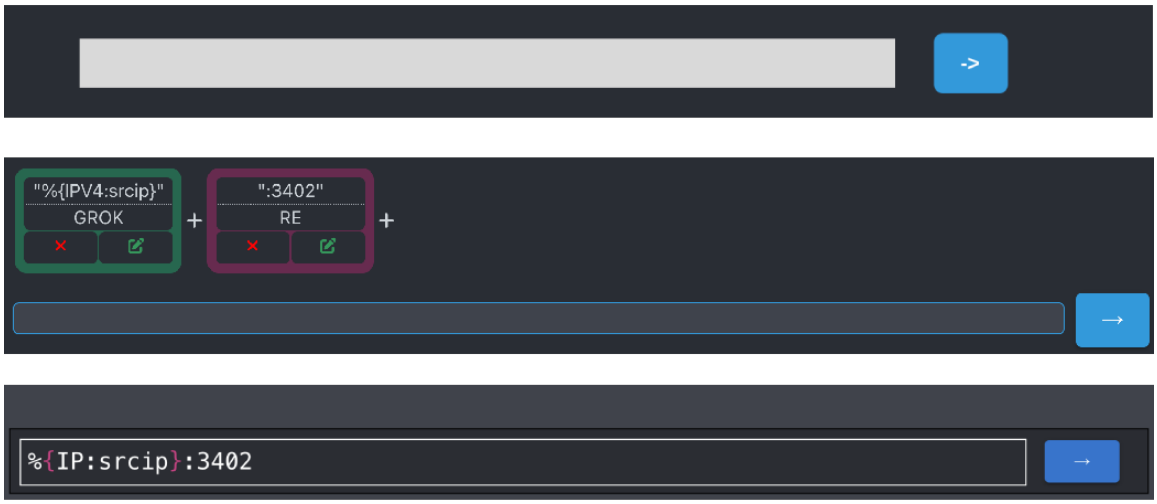

Figure 3.5: Comparison of input filter (ordered from the first iteration to the last one).

Grok pattern set selection

The Grok pattern selection (see Figure 3.6) loads Grok pattern sets, each containing a set of related Grok patterns from a `GrokPatterns` project directory. Each pattern has a checkbox allowing the user to select or deselect a specific Grok pattern set easily. Grok pattern sets can be preselected or added, if required, through the source code.

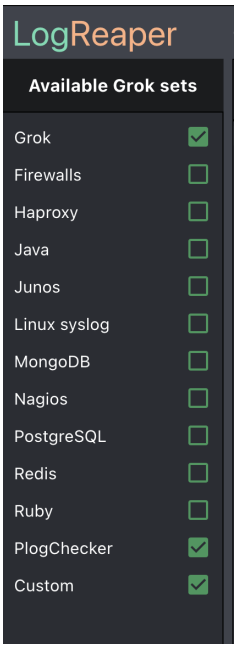

Figure 3.6: Grok pattern set selection component

State bar

Initially included in the initial design solely for aesthetic purposes, the state bar has evolved to encompass multiple functionalities, including line count, error notification, and application mode. The line count feature provides the user with a quick and easy way to keep track of the number of lines of the processed log. This feature is helpful for users who work with large logs, allowing for a better idea of how many lines have been filtered.

The error notification feature is also a valuable addition, as it alerts the user to any incorrect regular expressions or Grok patterns passed to the input filter. This lets the user quickly identify these issues and correct the inserted pattern.

Lastly, the application mode feature allows the user to keep track of the mode they are currently working with. The `Filter mode` lets the user know that they are working with a copy of the original inserted log and a filter has been applied. The second `Edit mode` means that the user can edit the inserted log. The state bar can be seen on Figure 3.7.

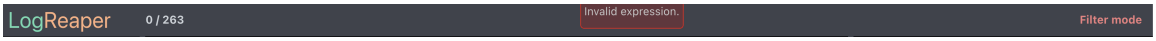

Figure 3.7: State bar component

Message pattern toolbar

The log message pattern toolbar has also undergone three design iterations as shown on Figure 3.8. In the first iteration, every log message pattern had its color assigned, and this color was used to highlight individual log messages that would match this pattern. If multiple log message patterns matched one exact log message, multiple hitboxes, each with its color, would be created, pushing the matched log message's element to move sideways as these hitboxes would stack next to each other. Each log message pattern had a checkbox, which, when selected, added this log message pattern to the `log message pattern filter`. The user could also edit the value of the log message pattern inside the toolbar itself.

In the second iteration, color assigning was removed entirely and replaced by a single red hitbox which gets darker as more log message patterns match the exact log message. The user can no longer edit the value of the log message pattern inside the toolbar as it missed all of the highlighting functions, which would let the user know that they created a valid pattern. Instead, the user can move this log message pattern to the input filter to test it on the inserted log while editing and then move it back once they are satisfied with the applied changes. The value held by the log message pattern is now read-only and is represented as a list of individual sub-patterns used to create the log message pattern, each with its type assigned to it.

The third iteration includes slight design changes. These changes include the ability to hide the sub-pattern list, change of the button style, `paremetrized` regular expression display support, and color highlight based on the sub-pattern type.

The two buttons have been replaced with a single export button as the initial functionalities have proven to have no use. Mainly because the text gets updated everytime a message pattern is selected.

Figure 3.8: Comparison of message pattern toolbar (ordered from the first iteration to the last one).

Grok recommendation popup

This popup is triggered when the user highlights a text inside a log, providing a list of recommended Grok patterns relevant to the highlighted text. This feature is particularly useful for users who may be new to Grok patterns or unfamiliar with the available Grok patterns. Once the user selects a Grok pattern that meets their requirements, the popup generates a valid one and inserts it into the input filter, preventing user errors such as incorrectly writing the pattern (see Figure 3.9).

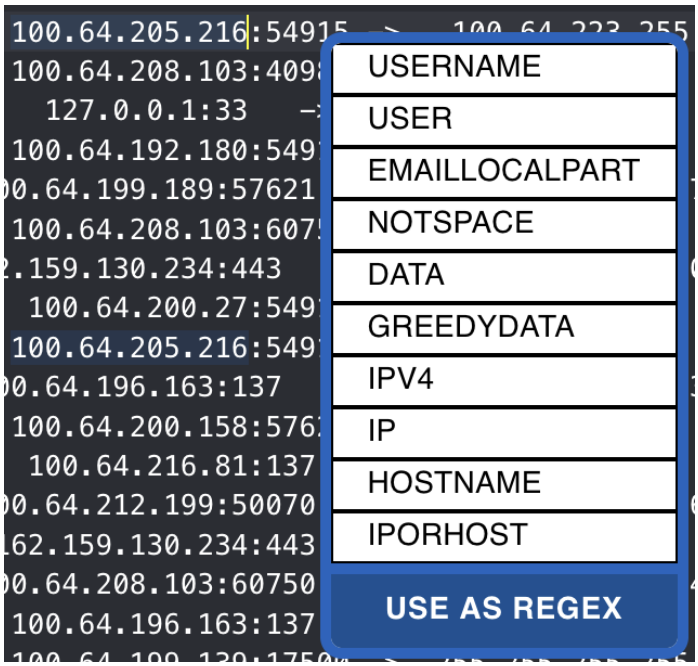

Figure 3.9: Grok recommendation popup.

3.5 Application usage

This section focuses on the functionality of individual parts of the application and the interaction with them from the user's perspective. The functionality of these parts was implemented based on the features defined in the Table 3.1.

Functionality of individual components

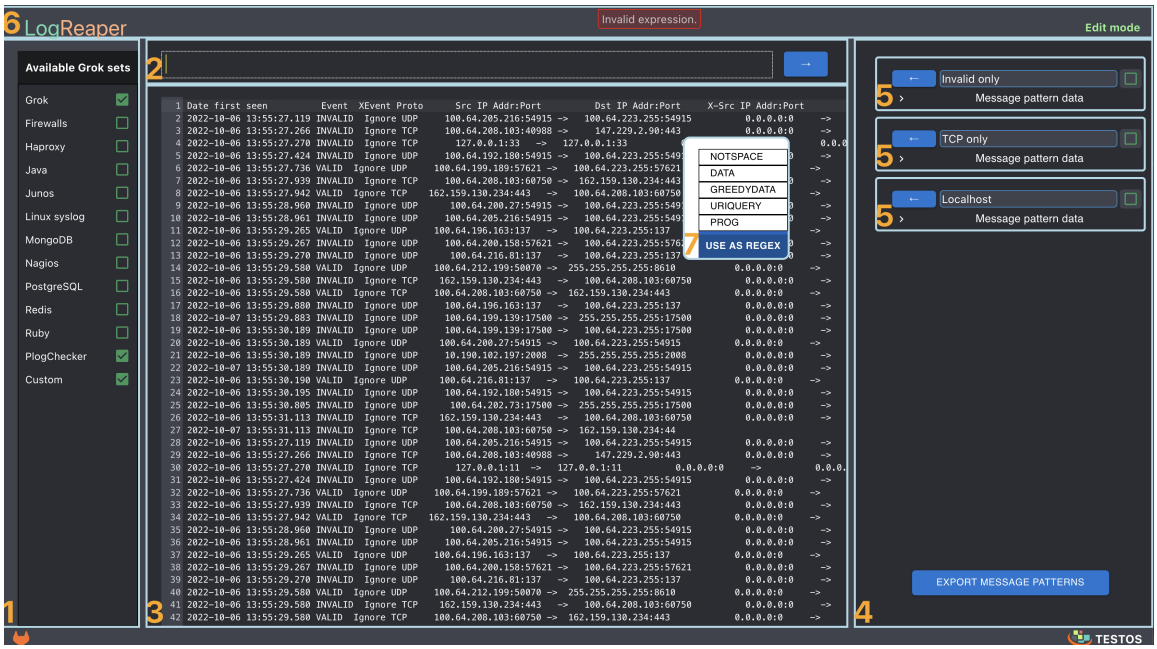

Figure 3.10: Individual components of the application.

Every highlighted component in the Figure 3.10 has an assigned number. This number matches the number of the description below:

Num	Component	Description
1	Grok set selector	Individual Grok pattern sets are used to select available patterns that the user can choose from.
2	Input filter	A component where the user can type a combination of Grok patterns and regular expressions, creating a string of expressions. This string is used to filter the main text window actively. The main goal is to create a string that fully defines the pasted log's specific lines (messages). Clicking the button will move the created string into a message pattern.
3	Main text window	The main text window, where the user can paste the log they want to filter and edit it.
4	Toolbar	Toolbar containing created message patterns that the user can interact with. The export button is used to export selected message patterns into individual events that can be later used in Plogchecker.
5	Log message pattern	A message pattern that consists of a button that moves this pattern into the filter, a pattern's name, and individual sub-patterns that create the pattern. The checkbox selects the pattern, triggering an event that will filter the inserted log, leaving only messages that match the pattern. Also, the user can export only patterns that are selected.
6	Application state bar	This component contains three functional elements which are `Applicaiton mode`, `Error message` and `Log message counter`. The `Error message` shows if the used regular expression or Grok pattern is invalid or not found in selected Grok sets. The `Log message counter` displays the total and current amount of log's messages. Finally, the `Application mode` is a mode that defines the current state of the application. Any edits made to the inserted log will not be applied during filter mode, as this mode is only used to filter it. When no message pattern is selected, and no text is inside the input filter, the application changes into an edit mode, during which the user can edit the inserted log freely.
7	Popup	A popup providing recommended Grok patterns matching the highlighted text inside the inserted log. These patterns are sourced from the selected Grok pattern sets.

Table 3.2: Description of individual application's components.

Usage of the application from the user perspective

The following is a detailed explanation of the usage of the application as depicted in Figure 3.11 and 3.12.

After the initial load, the application lets the user do several different things. The user can insert the log they would like to process into the main text window and choose the Grok pattern sets they want to use. Once the log is inserted, the user can create a string of regular expressions. They can either start typing in the input filter window or select a part of a single log message inside the inserted log to get recommended Grok patterns from active Grok pattern sets.

If a text that matches the syntax of a Grok pattern is detected, it will get evaluated and either accepted as a valid Grok pattern or revoked. The user will receive an error notification if the Grok pattern does not exist, and the input will be considered as a regular expression. In this case, the user can select more Grok pattern sets that may include the non-existent pattern. When the user successfully creates a Grok pattern or a regular expression, they can continue appending the text with more patterns or expressions, creating a string of regular expressions.

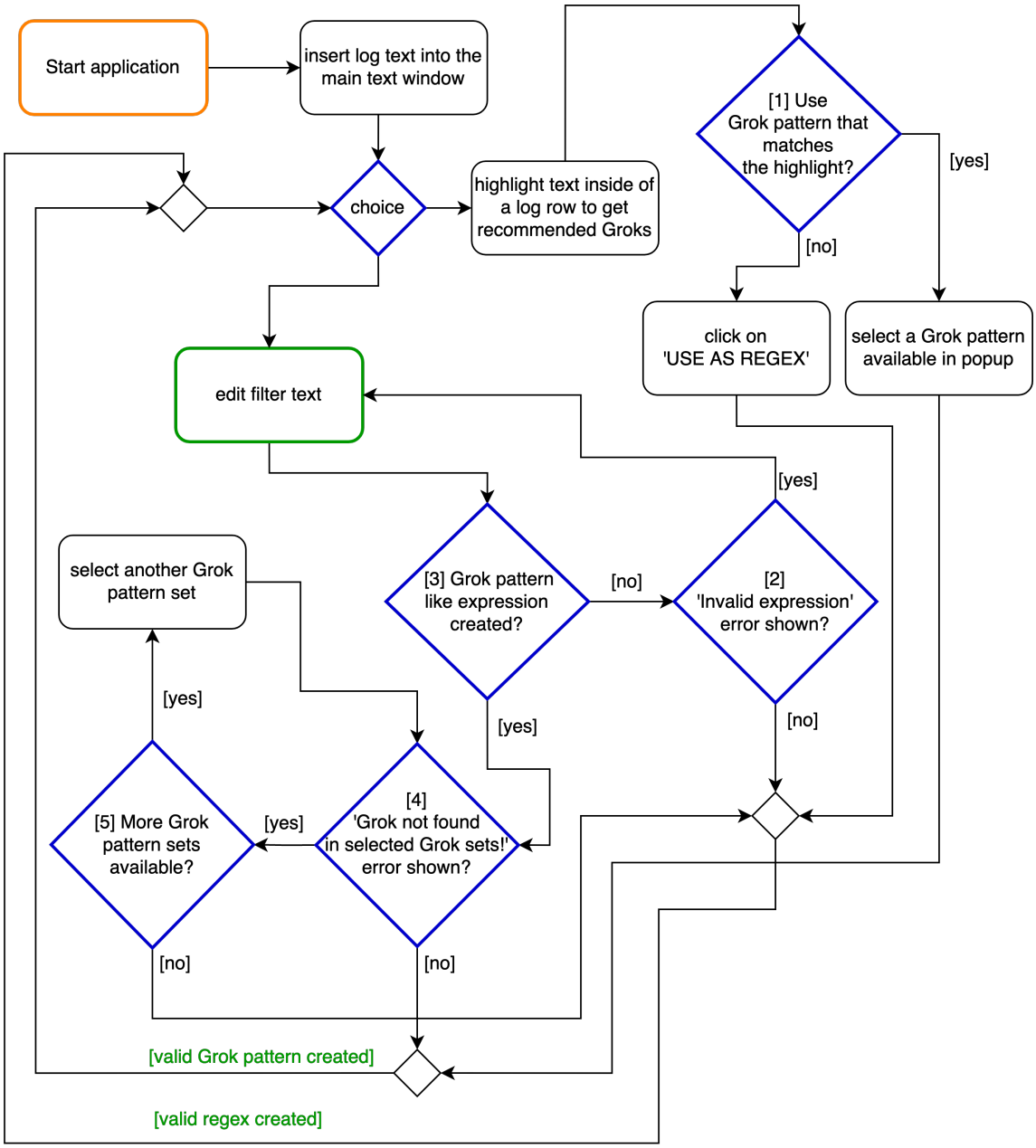

Figure 3.11: Activity diagram describing the creation of a valid regular expression.

Once satisfied with the created string, the user can export it as a log message pattern, which will be displayed in the log message pattern toolbar. There they can select their newly created, which will trigger an event that will match this messaging pattern with every message of the inserted log. Suppose the expectations have been met, and only the expected log messages are selected after the log message pattern is applied through the filter. In that case, the user can export this messaging pattern as a single event for Plogchecker. Otherwise, the user can move this log message pattern back to the input filter and edit it again.

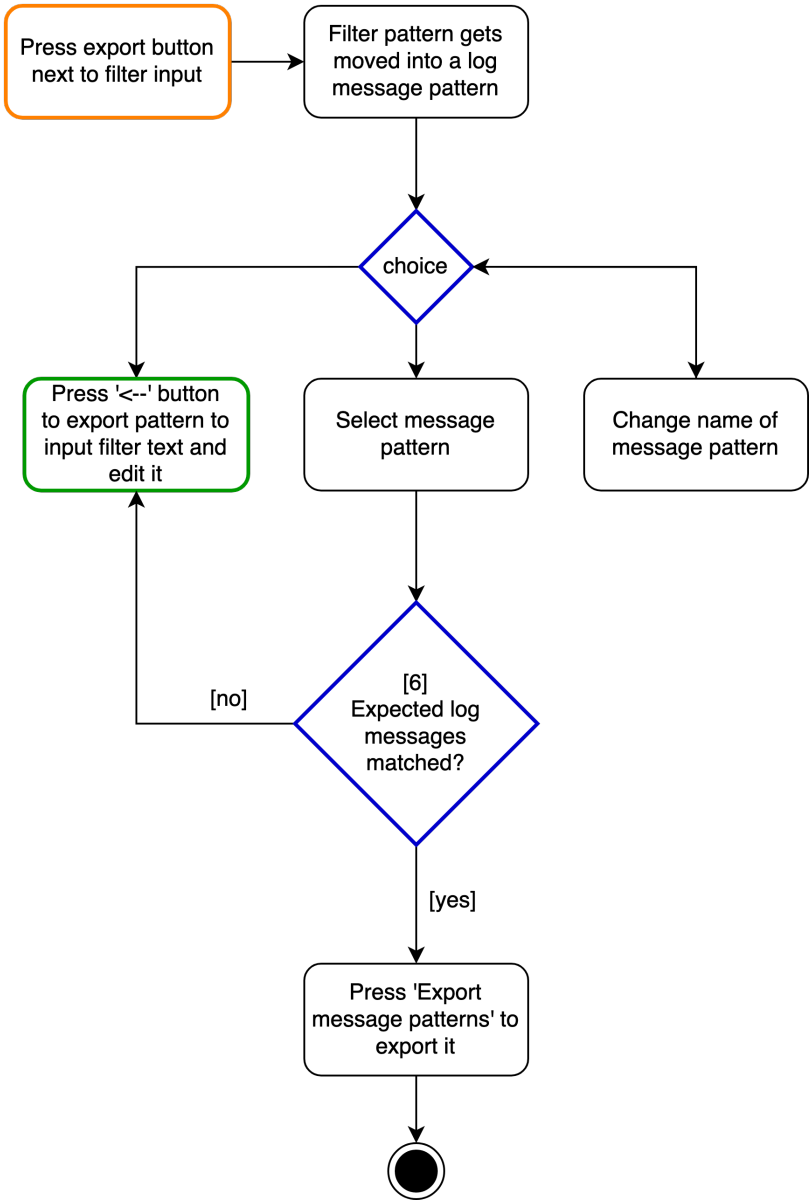

Figure 3.12: Activity diagram describing the export of created string of regular expressions.

Chapter 4

Used technologies

This chapter describes the technologies used throughout the development of this application. These technologies to create a portable web application were selected based on their popularity among developers.

4.1 Application

This part focuses on the technologies and libraries used to implement this application UI and its core functionalities, such as processing regular expressions incompatible with JavaScript and processing inserted logs.

React

React is a JavaScript library intended for building user interfaces. It allows developers to create UI components. They are reusable pieces of code that can be used to create a UI elements, such as a button or a table. The most commonly used components are so-called functional components. In order to make it possible for functional components to have the same capabilities as class components, version 16.8 introduced hooks. With hooks, functional components can manage the state and lifecycle of their methods. React uses the virtual DOM (Document Object Model), a lightweight in-memory representation of the actual DOM, which describes how a document's logical structure is defined and how the document can be accessed and manipulated. When a state of a component changes, the change first applies to the virtual DOM, where the current state is compared with the previous state, and based on this comparison, React will then determine which parts of the real DOM need to be updated [5].

TypeScript

TypeScript is an open-source programming language developed and maintained by Microsoft. It is a strict syntactical superset of JavaScript and adds optional static typing to the language, significantly improving the final code's readability. Type-checking execution happens during compile time, which allows developers to catch errors before the code execution. One of the main benefits of TypeScript is that it transpiles to JavaScript. That means that TypeScript is compatible with any web browser [7].

JSX

JSX is a syntax extension for JavaScript used in React to create individual elements. For the sake of simplicity, an element can be imagined as a single HTML element. JSX elements are transpired by a tool called a JSX transformer, such as Babel, into plain JavaScript. JSX transformers, such as Babel, do transpilation of JSX elements into plain JavaScript. Additionally, JSX allows developers to embed expressions and variables within individual elements. By default, React DOM escapes any values in JSX before rendering them. Thus it prevents injecting any malicious code that is not explicitly written in the application [11].

CodeMirror

CodeMirror is a versatile text editor implemented in JavaScript. It provides many features, including syntax highlighting, line numbers, and auto-completion. A rich programming API and a CSS theming system are available for customizing the editor [15]. This project uses a React component with the latest version, Codemirror 6.

Oniguruma

Oniguruma[1] is a regular expression library initially developed for the Ruby programming language. It provides a set of functions for pattern matching with regular expressions and supports various features, including character classes, alternations, quantifiers, or backreferences. Oniguruma is known for supporting Unicode character encoding, allowing it to handle various character encodings.

Material UI

Material UI is an open-source React component library that implements Material Design [4], developed by Google. Google uses Material Design [3] to ensure that no matter how users interact with the products they use, they will have a consistent user experience. Material Design sets the rules for typography, grid, images, and many other things. The library for React offers a wide variety of components for any chosen design. Material UI also comes with a built-in theming system that makes creating custom themes for components easy.

4.2 Tests

This part describes frameworks used to create individual end-to-end tests (see Section 6.1) for the application through web automation in order to simulate user interaction with the application.

Selenium WebDriver

Selenium WebDriver is a framework that lets users write automated tests that can be run on different browsers, such as Chrome, Firefox, or Safari. WebDriver runs the browser natively, just as a user would locally or on a remote machine using the Selenium server.

The primary purpose of Selenium WebDriver is to be able to interact with individual webpage components using actions like clicking, typing, and executing javascript scripts. It

[1] https://github.com/kkos/oniguruma

can also handle dynamic web pages as it can wait for elements to appear on the page and execute the next step once they are visible [6].

NUnit

NUnit is an open-source unit testing framework created for .NET languages. It provides a set of assertions, different parameters for individual tests and reports the results of the tests [16].

In order to create a test class in `C#`, the class needs to be marked with a `TestFixture` attribute. This class contains individual test methods that are marked with the Test attribute. In order to generate a combination of inputs for the test, a `Combinatorial` attribute is used. This attribute means that the test method expects `ValueSources` as its arguments. These `ValueSources` are generally any collection that can iterate through its elements. NUnit will generate tests based on their combinations when multiple collections are passed.

Other crucial NUnit attributes are:

- `OneTimeSetUp` – a method that is run only once for every test class before all tests.
- `OneTimeTearDown` – a method that is run only once for every test class after all tests.
- `SetUp` – a method that is run before each test.
- `TearDown` – a method that is run after each test.

Chapter 5

Implementation details

This chapter describes the structure of the application, communication between individual components, and the details of their implementation.

5.1 Application Folder Structure

The project is organized by dividing source files based on their types or common attributes. The LogReaper's folder structure is shown in Figure 5.1.

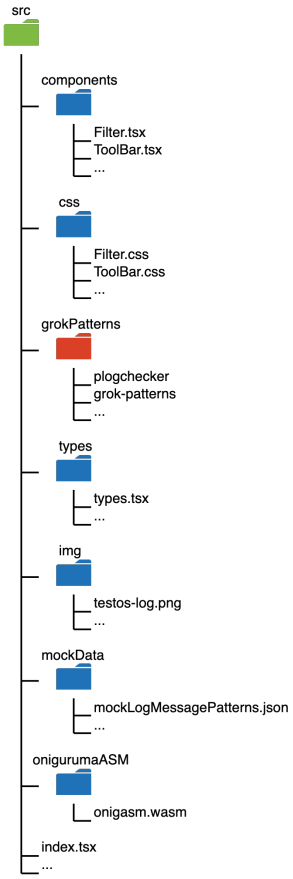

Figure 5.1: Application folder structure.

The project solution is split into sub-folders as follows:

- `components` folder contains the source code of individual application's components.
- `css` folder holds CSS styles defining the design aspect of the application's components.
- `grokPatterns` folder stores Grok pattern sets in plain text. Here, the user can add custom Grok pattern sets. The application loads all the Grok pattern sets in this folder and displays them in the Grok pattern set selection 3.4.
- `types` folder holds the custom types mentioned in Section 5.3 used with `Typescript`.
- `img` folder contains graphics used for the application's and Testos logos.
- `mockData` folder holds `JSON` files containing preset values for the main text and logs message patterns. This data can be loaded into the application for testing.
- `onigurumaASM` folder contains the `onigasm.wasm` file. This is a `WebAssembly` file containing the compiled Oniguruma regular expression library that is responsible for allowing the application to process regular expressions that are not natively supported by the *ECMAScript* flavor. The application communicates with this `WebAssembly` through the `Onigasm` library (see Section 5.4).

5.2 Application Component Structure

Splitting the functionality into individual components keeps the application organized. The structure consisting of individual components is shown in the Figure 5.2. These components update their inner states based on the changes made by other components. This communication between them is show in the Figure 5.3.

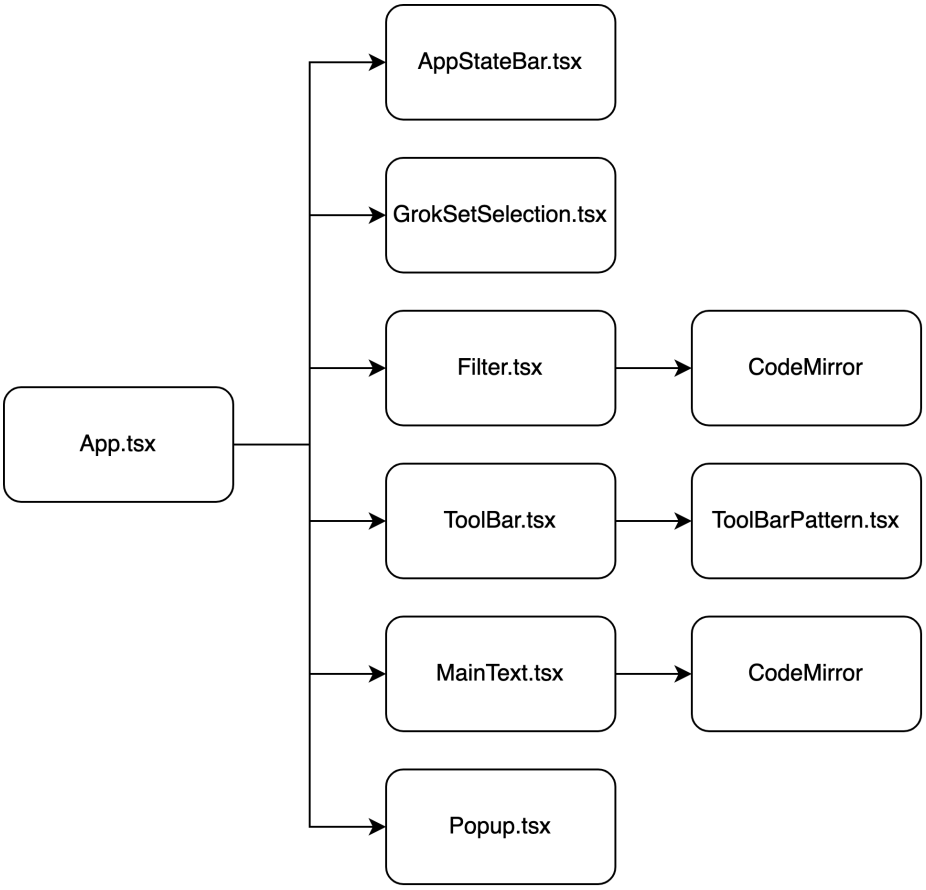

Figure 5.2: Application component structure.

- **AppStateBar** – this component displays the overall state of the application. It consists of a log message counter, which indicates the total and current messages in the inserted log, application mode display, and notification error handle.
- **GrokSetSelection** – this component displays individual Grok sets loaded from the GrokPatterns directory. These Grok sets are loaded into objects containing arrays of individual Grok patterns. When selecting a specific set, individual Grok patterns are moved into active Grok patterns, which are used to validate Grok patterns inside the `Filter` component and display recommended Grok patterns inside the `Popup` component.
- **Filter** – handles everything related to filtering the main log. Every time the user updates text inside the input filter, this text gets processed and scanned for valid Grok patterns or parametrized regular expressions. If valid Grok patterns or parametrized regular expressions are detected, the `Filter` component will highlight them. The `Filter` component also reacts to selecting a specific log message pattern inside of the `Toolbar` component. The strings of regular expressions from these components are then combined, creating one regular expression, which is then applied to the `MainText` component.
- **ToolBar** – consists of individual `ToolBarPatten` components, representing individual log message patterns. This component listens to the `Filter` component and creates

a new `ToolBarPattern` component when the user exports a created string of regular expressions inside of the `Filter` component. Once the log message pattern is created, the `Toolbar` component lets the user edit its name and select it so that the `Filter` component can use it to filter the inserted log. Once the user is satisfied with the created message pattern, they can export it as a YAML file.

- **MainText** – this component handles every change directed to the processed log. It listens for requests from the `Filter` component and updates the log, adds hitboxes to individual log messages when a matching log message pattern is selected inside of the `Toolbar` component, and switches between editing and filtering mode so that the user is not able to edit the text while in filtering mode as they work with a copy of the original log.
- **Popup** – this component is displayed once the `MainText` component sends a request when the user highlights text inside a log message. This component then checks for active Grok patterns and matches them to the highlighted text that was provided by the `MainText` component. Matched Grok patterns are then displayed and can be selected. Upon selecting a pattern, it will get moved into the `Filter` component.

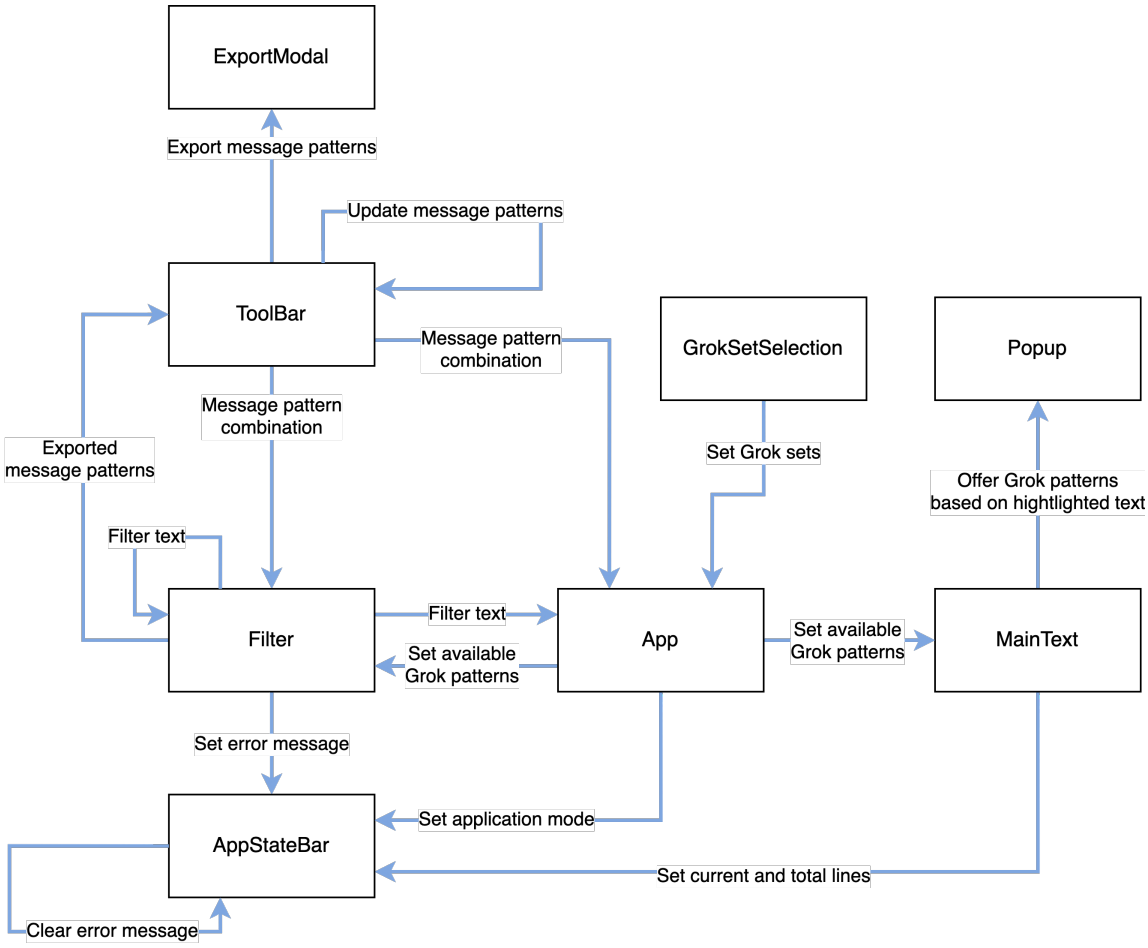

Figure 5.3: Communication among individual components.

5.3 Custom types

Creating custom types means defining custom data structures that can be used throughout the codebase. Doing so can make the code more readable, maintainable, and reusable. There are currently three custom types present in the code.

```
type GrokPattern = {
  name: string;
  value: string;
};
```

The first custom type is `GrokPattern`. This type is used to process Grok patterns passed into the `Filter` component. The attribute `name` matches Grok patterns inserted into the `Filter` component. The `value` of the matched pattern is then resolved (in case it contains nested Grok patterns) and used as a regular expression to filter the inserted log.

```
type GrokPatternSet = {
  type: string;
  name: string;
  isActive: boolean;
};
```

The second custom type is `GrokPatternSet`. This type is used to define the individual Grok pattern sets that are present in the application. `Type` defines the name of a Grok pattern set inside the `GrokPatterns` directory and is used to load the plain-text Grok pattern set into the application. `Name` serves as a display name, which is shown in the `GrokSetSelection` component, and `isActive` is used as a flag to keep track of selected Grok pattern sets.

```
type MessagePatternData = {
  id: string;
  name: string;
  value: string;
  checked: boolean;
  patterns: { type: number; value: string }[];
};
```

Finally, the last type `MessagePatternData` defines how individual log message patterns are stored. `id` is a unique identification of a log message pattern, `name` is the name that the user gives to the pattern, and `value` is a regular expression that this log message pattern stores. `checked` represents a specific log message pattern selection.

Every log message pattern is created using multiple sub-patterns. These patterns are displayed in the `ToolBar` component. Every sub-pattern has its `type`, which defines if it is a regular expression, parametrized regular expression, or a Grok pattern.

5.4 Oniguruma Web Asembly

For the application to support regular expression engines outside of the *ECMAScript* standard, an `Oniguruma` WebAssembly is used. This assembly is included in the `Onigasm` library.

```
(async () => {
  await loadWASM(require("./onigurumaASM/onigasm.wasm"));
  root.render(
      <App />
  );
})();
```

To use `Oniguruma` WebAssembly, we first need to load it into the application. The loading occurs before the application is rendered to prevent errors when calling functions using this WebAssembly after the initial application render.

Once the WebAssembly is loaded, it is possible to use `Oniguruma` regular expression functions inside of a `JavaScript` application. This is possible because the WebAssembly file contains a binary code of `Oniguruma`, which is programmed in `C`, meaning it does not have a built-in regular expression engine, unlike `JavaScript`.

```
var onigRe = new OnigRegExp("pattern");
```

Using `Oniguruma` regular expression engine is now relatively straightforward. First, we need to create a new instance of `OnigRegExp`, just like we would with `RegExp`.

```
var str = "random string";
var res = onigRe.searchSync(str) != null;
```

After creating the `OnigRegExp` object, we can search through a string using the `searchSync` function.

5.5 React component functionalities

This section covers a detailed implementation of the core functionalities of React components. These functionalities include the usage of Oniguruma to process regular expressions that do not meet the *ECMAScript* standard, processing text using `CodeMirror's` transactions, sourcing Grok sets and exporting created log message patterns in YAML format.

Grok sets

Individual Grok sets are stored locally inside the `GrokPatterns` directory. Every Grok set is represented as a plain-text file containing Grok patterns. Every Grok pattern consists of two parts. One of them is the `Syntax` of the Grok pattern, and the part is the value that this Grok pattern stores. A single whitespace character splits these two parts. For example, a Grok pattern `NUMBER` is stored as follows:

```
NUMBER (?:%{BASE10NUM})
BASE10NUM (?<![0-9.+-])(?>[+-]?(?:(?:[0-9]+(?:\.[0-9]+)?)|(?:\.[0-9]+)))
```

In this case, `NUMBER` is the `Syntax` of the Grok pattern, and the value that this Grok pattern holds is `(?:%BASE10NUM)`. The example shows that the value contains another Grok pattern, which is `BASE10NUM`. As mentioned in Section 2.4, a single Grok pattern can contain multiple Grok patterns inside its value. The same principle stores all of the Grok patterns inside the `GrokPatterns` directory.

Loading Grok sets

A `UseEffect` hook is called when a Grok set is selected. Inside this hook, every selected Grok set is fetched from the directory `GrokPatterns`. After bringing the selected Grok sets, every Grok pattern included in these sets is parsed and loaded inside of the `grokPattern` property.

```
const fetchData = async () => {
  var grokArray = [];
  for (var i = 0; i < grokPatternSets.length; i++) {
    if (grokPatternSets[i].activated) {
      // fetching Grok sets from GrokPatterns directory
      var loadedFile = require("../GrokPatterns/" + grokPatternSets[i].type);
      const data = await axios.get(loadedFile);

      // load individual Grok patterns into an array
      const dataArray = data.data.split(/\r?\n/);

      // iterating trough individual Grok patterns
      for (var j = 0; j < dataArray.length; j++) {
        // ignoring comments
        if (dataArray[j].length > 0 && dataArray[j][0] !== '#') {
          // spliting Grok pattern
          var alias = dataArray[j].substring(0, dataArray[j].indexOf(' '));
          var value = dataArray[j].substring(dataArray[j].indexOf(' ') + 1);

          grokArray.push({name: alias, value: value});
        }
      }
    }
  }

  // saving the Grok patterns
  setGrokPattern(grokArray);
}
```

Adding new Grok sets

In case the user wants to add a new Grok set, they first need to copy it inside the `GrokPatterns` directory. After that, they must add it to the `GrokPatternSets` property. This property consists of three attributes. Attribute `type` is the name of the Grok pattern set file. Attribute `name` is the display name, which will be shown inside of the `GrokSetSelection` component, and the last attribute is `activated`, which, if set to `true` will be pre-selected upon the application load.

```
const [grokPatternSets] = useState([
  {type: "grok-patterns," name: "Grok," activated: true}
]);
```

This example saves a Grok pattern set file as `grok-patterns`. The name shown inside the Application will be `Grok`, and it will be pre-selected upon the Application load.

Resolving Grok patterns

As mentioned earlier, a single Grok pattern can contain more Grok patterns inside its value. As the log that we want to process is filtered using regular expressions, the created Grok pattern needs to be transformed into one. A Grok pattern is transformed into a regular expression using recursive Grok pattern resolution. The resolution of a Grok pattern can be explained using several steps.

The first step is locating the Grok pattern inside another Grok pattern's value using a regular expression that matches the form in which a Grok pattern is written.

```
NUMBER (?:%{BASE10NUM:num})
BASE10NUM (?<![0-9.+-])(?>[+-]?(?:(?:[0-9]+(?:\.[0-9]+)?)|(?:\.[0-9]+)))
```

The value of the Grok pattern `NUMBER` contains a Grok pattern `BASE10NUM`, which is written in the form of `%{BASE10NUM}`.

```
var finalExpression = "";
var patternValue = "(?:%{BASE10NUM:num})";
var matchedPatterns = ["%{BASE10NUM:num}"];
```

When the regular expression used to match Grok patterns is applied, an array is created containing a value, that is `%{BASE10NUM}` as it is the only Grok pattern present.

```
var pattern = matchedPatterns[0].slice(2, -1);
var patternSplit = [];
var indexOfSplit  = pattern.lastIndexOf(':');

    patternSplit.push(pattern.slice(0, indexOfSplit))
patternSplit.push(pattern.slice(indexOfSplit + 1))
```

In the second step, the pattern `%{BASE10NUM}` is stripped of the brackets and split into two parts, which are the `Syntax` of the Grok pattern and `ID` of the Grok pattern and saved inside of an array.

```
var regex = new RegExp(matchedPatterns[0] + "(.*)");
var splitPattern = pattern.split(regex);
pattern = splitPattern[1];
finalExpression = splitPattern[0];
```

The third step creates a regular expression named `regex`. This regular expression matches a string starting with `(?:%{BASE10NUM:num})` to the end of the line. This regular expression is then used to split the string `patternValue`, creating an array `SplitPattern`, which holds a value as follows:

```
splitPattern = ["?:", "%{BASE10NUM:num}"]
```

The left side of the array is then appended to the string `finalExpression` string. This string now contains the value of:

```
finalExpression = "?:"
```

The last step is resolving the right side of the array `splitPattern`. That is done by calling the same process that we used in this example, but the new value of the string `patternValue` would be „`%{BASE10NUM:num}`", and as the value of the Grok pattern `BASE10NUM` does not contain any Grok patterns, we can add its value to the string `finalExpression`, which will now hold the value of:

```
finalExpression = \
    "?:" + "(?<![0-9.+-])(?>[+-]?(?:(?:[0-9]+(?:\.[0-9]+)?)|(?:\.[0-9]+)))"
```

Now the Grok pattern `NUMBER` is successfully converted into a regular expression, and thus, it can be used to filter the processed log. In case an invalid Grok pattern is found at any moment during the resolution, the user is notified with an error (see Figure 3.11, node [3]).

Grok recomendation popup

When the user highlights a text inside a log message, the `MainText` component will try to match individual Grok patterns from the selected Grok sets using the Grok pattern resolution mentioned in the Section 5.5. After the Grok pattern matching is finished, the `MainText` component lets the `Popup` component know which Grok patterns should be displayed. Once the popup is displayed, the user can choose a fitting Grok pattern, which will then get appended to the input filter's text (see Figure 3.11, node [1]).

```
document.onselectionchange = function(event:any) {
    // timer to prevent the popup appearing after selection change
    // and wait until the user is finished selecting the text
    clearTimeout(popupTimer);
    popupTimer = setTimeout(() => checkForSelection(event), popupInterval);
};
```

First, an event listening for selections inside of the Application is created. Once the user highlights texts inside the `MainText` component, the function `checkForSelection` is called. This function checks if the highlighted text is only inside one log message. In other words, it does not contain a newline character.

```
const getRecommendedGrok = (textToMatch:string) => {
matchedPatterns = [];

    for (var i = 0; i < grokPatterns.length; i++) {
      // compare pattern to the entire selected string
      var reVal = "^(" + resolveSubPatterns(grokPattern[i].value) + ")$";
      var onigRe = new OnigRegExp(reVal);

      if (onigRe.searchSync(textToMatch).length > 0)
        matchedPatterns.push(grokPattern[i]);
    }

    // setting the property storing matched Grok patterns
    setSelectionMatch(matchedPatterns);
}
```

After the selection check, a function `getRecommendedGrok` is called. This function then uses `Oniguruma` regular expression library to match the highlighted text to Grok patterns of each activated Grok set. Before the string is passed to the regular expression library, it first needs to be resolved in case the Grok pattern that is being matched contains Grok patterns in its value.

The final array containing matched Grok patterns is then passed to the `setSelectionMatch` property. Updating the property value will trigger a `UseEffect` hook inside the `Popup` component. The position of the popup is then calculated before rendering to prevent it from leaking outside of the application window bounds.

CodeMirror Editor

Both `Filter` and `MainText` components use `CodeMirror` library. It allows the application to have a more advanced text editor with line numbering, text highlighting, and the ability to make the editor read-only.

`CodeMirror` uses transactions to modify the editor's current state. This transaction creates a new instance of the editor's state as it is a persistent (immutable) data structure, meaning that if the object is directly edited without using transactions, it will become inconsistent. Inconsistency may cause the editor's state not to be updated inside the editor's `view`.

The `editor's state` represents the document that the editor holds and all of the logic around it, such as changing the value of the document, slicing the document, setting it to the read-only mode, or storing the current selection.

The editor's `view` is the user interface of the editor. It handles events, dispatches state transactions for editing actions, and holds the editable DOM (Document Object Model), representing the editor's element structure, style, and content.

Filtering log contents

When the application is in the `Edit mode`, the user can edit the log text directly. Once the application switches to the `Filter mode`, the editor must be updated internally, and this is when `transactions` take place.

```
      view.dispatch(
        view.state.update(
          {changes: {from: 0,to: view.state.doc.length , insert: filteredDoc}}
      ),
    )
```

When the log gets filtered, a new value must be set to the `MainText` component. First, we create a transaction on the `editor's state`. Here we define its specifications. In this case, we want to change the text inside the document. Here we set the interval we want to change to the whole document and insert the new filtered value.

As we want these changes to take effect immediately, we dispatch this transaction through the editor's `view`.

Filter text highlight

When the user creates a valid Grok pattern or parametrized regular expression, it will be assigned a CSS class highlighting it.

The initial idea was to create an LR grammar defining the form of Grok patterns and parametrized regular expressions using `Lezer` parser[1]. On top of the grammar, a syntax highlight would've been created. This highlighter would have extended the `CodeMirror` editor. While making the grammar seem relatively simple, a few complications occurred. One of these complications was the inability to find a proper manual for parsing the created LR grammar into a form that the syntax highlighter would understand. Another complication was the incompatibility of the Lezer library with React. Based on these complications, a different highlighting method has been chosen.

This method utilizes `CodeMirror's decorations`. Decorations define how certain parts of the document should be styled. For the `Filter` component, a `mark` decoration has been used.

```
// creating a~state effect
// creating effect extension
const highlightExtension = StateField.define({
  create() { return Decoration.none },
  update(value, transaction) {
    value = value.map(transaction.changes)

    for (let effect of transaction.effects) {
      if (effect.value != null)
        if (effect.is(highlightEffect))
          value = value.update({add: effect.value, sort: true})
    }

    return value
  },
  provide: f => EditorView.decorations.from(f)
});
```

Before being able to integrate the decoration inside of the `CodeMirror` editor, an extension needs to be made. First, a `highlightEffect` is defined. Then this effect is used inside of the `highlightExtension` extension. An empty decoration is created upon the editor's initialization. Then on every `editor's state` update, the `update()` function is called. This function creates a transaction which purpose is to change the document. If a `highlightEffect` effect is dispatched inside a transaction, this effect is added to decoration sets of the `CodeMirror` editor. This code was adapted from the official CodeMirror documentation[2].

[1] https://lezer.codemirror.net/

[2] https://codemirror.net/examples/decoration/

```
<CodeMirror
    extensions = {[highlight_extension]}
/>
```

Now we can add the created extension to the `CodeMirror` component.

```
var lineDecorations = [];
var lineDecoration = Decoration.mark({
  attributes: {style: "background-color: pink"}
 });

lineDecorations.push(lineDecoration.range(from: 0, to: 10);
)
```

Once the extension has been added to the `CodeMirror` component, we can use it inside the code. As mentioned earlier, the `Filter` component uses the `mark` decoration. This decoration takes CSS styles as its argument. In this example, we want to set the background color to pink. After that, we can apply the decoration to the editor's text, in this case, from the start of the string to its 10th character.

```
view.dispatch(
  {effects: (highlightEffect as any).of(lineDecorations)}
);
```

The last step is to dispatch this change through the editor's `view`. This process creates output, as shown in Figure 5.4.

Figure 5.4: CodeMirror highlight decoration.

Log message hitboxes

The process of creating an effect extension is the same as with the `Filter` component. The only difference is in the decoration used.

```
var lineDecorations = [];
var lineDecoration = Decoration.line({
  attributes: {style: "background-color: pink"}
 });

lineDecorations.push(lineDecoration.range(from: 0);
```

The decoration that is used to create individual hit-boxes when the text is filtered using log message patterns is the `line decoration`. This decoration applies to the whole line, so the end of the selection range is not set.

Log message pattern export

Once the user is satisfied with the matched log messages after filtering, (see Figure 3.12, node [6]), exporting the selected log message patterns is pretty straightforward. First, a new object `exportedPatterns` is created. This object holds individual selected log message patterns as follows:

```
exportedPatterns["patternName"] = "patternValue";
```

The pattern name acts as a `key` based on which the pattern will later get matched to a YAML attribute and the pattern value as the `value` this pattern holds.

```
var yamlString = yaml.stringify(
  { events: exportedPatterns },
  { defaultStringType: "PLAIN", nullStr: "~" }
);
```

The next step is to parse these log message patterns into a YAML format string. Every pattern represents an `event` as shown in the Figure 2.1. The default string type is set to `PLAIN`, meaning that the strings are not quoted or double quoted unless deemed necessary. In YAML, a string is generally represented without any quotes. Double quotes are used in case we want to parse escape codes. For example `"\n"` will be returned as a line feed character. When single quotes are used, any escape codes or special characters inside the string will not be parsed. This means that the string containing `'\n'`, will be returned as the string `\n`.

Chapter 6

Testing

This chapter describes the testing methods used for this project. The first part describes the usage of end-to-end testing using `Selenium` with `NUnit`. The next part explains how user testing was used to earn feedback on the application and feature suggestions that may be implemented.

6.1 End-to-end Testing

End-to-end testing (E2E testing) is a software testing approach that tests the entire application workflow from start to finish by focusing on the application's overall behavior rather than testing individual isolated components. In E2E, the application is tested from the end user's perspective by simulating real-world user scenarios, including the user interface, backend services, databases, and network communication [9].

The main reason E2E is being used for this application is to identify any unwanted behavior that may arise when different application components communicate while simulating individual user scenarios that are contained inside of individual features shown on Table 6.1.

Feature	Representing class
Application-mode-display.feature	ApplicationModeTest.cs
Grok-recommendations.feature	GrokRecommendationTest.cs
Grok-set-selection.feature	GrokSetSelectionTest.cs
Insert-log.feature	InsertingLogTest.cs
Log-filtering.feature	LogFilteringTest.cs
Log-message-counter.feature	LogMessageCounterTest.cs
Filter-highlight.feature	FilterHighlightTest.cs
Log-message-pattern-operations.feature	LogMessagePatternTest.cs
Message-highlighting.feature	MessagePatternExportTest.cs
Log-message-pattern-export.feature	MessagePatternExportTest.cs

Table 6.1: Inidividual features and their corresponding classes.

Every feature is represented by a class that implements individual pre-defined scenarios. The class `CommonTest` holds shared initialization of individual tests. That means creating a new instance of the application and the chosen `Selenium WebDriver`, currently supporting `Chrome` and `Safari` web drivers.

Web automation

Web automation is the ability to programmatically control a website through its web interface using scripts and tools. It saves time by automating processes, usually done manually. In this project, web automation is done using `Selenium WebDriver`.

This project uses web automation to simulate a user using the application through a web browser. This includes filling out text fields, selecting recommended Grok patterns, filtering the inserted log, selecting Grok sets, switching between application states, managing log message patterns, and exporting them.

Web automation was chosen over React tests because of the possibility of detecting errors from the user's perspective and not interacting with individual components through `JavaScript`. This way, we can see if the application behaves correctly as a whole instead of testing isolated components' behavior which may not fully capture how the component interacts with the rest of the application. We can check if the user will get a recommendation of Grok patterns once they highlight a specific part of the log or if the text window is not editable while in filtering mode.

Selenium WebDriver setup

Before starting a test, the application is initialized in the `CommonTest` class. When initializing the application, a new instance of `Selenium WebDriver` is created inside the `BrowserDriver` class.

```
public class BrowserDriver
{
    public IWebDriver? driver;
    public WebDriverWait? driverWait;
    private string url = "http://localhost:3000/";

    public void Start()
    {
        // creating a~web driver instance
        if (DriverType == BrowserType.Chrome)
            driver = new ChromeDriver();
        else
            driver = new SafariDriver();
        // loading the application using selected URL
        driver.Navigate().GoToUrl(url);
        // new instance of WebDriverWait with the timeout of 10 seconds
        driverWait = new WebDriverWait(driver, TimeSpan.FromSeconds(10));
    }
}
```

Based on the web driver that the test requires, either a `Chrome` or `Safari` web driver is initialized. Upon initializing, a new application's page is loaded. Then an instance of `WebDriverWait` class is created. The `WebDriverWait` class is used to wait for the webpage element to load, meaning that the tests do not need to use implicit wait before executing a particular operation on the webpage.

Selenium operations

The class `App` contains a collection of helper methods (see Table 6.2) to prevent code duplicity. Each method defines a specific interaction that can be done with the application. They contain an `Xpath` to locate an element inside the application and then interact with it using `Selenium WebDriver` operations. These methods serve as building blocks for creating scenarios of individual features.

Method	Functionality
SetFilterText(string text)	Sets the text of the input filter.
GetFilterText()	Returns the text of the input filter.
SetMainText(string text)	Sets the text of the main text window.
GetMainText()	Returns the text of the main text window.
HighlightMainText()	Highlights all text inside of the main text window.
ClickFilterExportButton()	Moves text from the input filter and converts it into a log message pattern.
GetMessagePatternCount()	Returns the number of a log message patterns.
GetMessagePatternText(int num)	Returns the name of a log message pattern based on the position from the top.
SetMessagePatternText(int num, string text)	Sets the name of a log message pattern based on the position from the top.
ClickMessagePatternCheckBox(int num)	Selects a log message pattern based on the position from the top.
ClickMessagePatternExport(int num)	Moves a log message pattern into the input filter.
ExpandMessagePatternSubPatterns(int num)	Unfolds the list of individual sub-patterns inside of a log message pattern.
GetMessagePatternSubPatternTypes(int num)	Returns the type a log message sub-pattern (Re, Grok, PRe) based on the position from the top.
GetExportText()	Returns the text inside of the export dialog.
GetEditorMode()	Returns the editor's mode value (Filter mode, Edit mode)
PressExportButton()	Exports log message patterns.
IsPopupVisible()	Returns true if the popup is visible, otherwise returns false.
PopupUseAsRegex()	Clicks on 'USE AS REGEX' inside of the popup.
PopupUseGrok(string name)	Clicks on a Grok pattern inside of the popup based on its name.
GetValidGrokCount()	Returns the number of valid Grok patterns inside of the input filter.
GetValidParamReCount()	Returns the number of valid Parametrized regular expressions inside of the input filter.
SelectGrokSet(string name)	Selects/deselects a Grok set based in its name.
GetMessageCounterValue()	Returns the amount of current and total log messages(lines).
Kill()	Kills the Selenium WebDriver (closes the application).

Table 6.2: Supported operations in the test suite.

Test categories

Every test class has a category named after a feature's name in the form as follows:

```
[Category("FeatureName")]
```

These categories are then used when only selected test classes need to be run. They make it easier to test only certain parts of the applications that need to be verified repeatedly.

```
dotnet test Automation.LogReaper --filter='TestCategory=\
    "Application-mode-display"|TestCategory="Filter-highlight"'
```

In this example, only tests covering scenarios of the feature `Application-mode-display` and `Filter-highlight` will be run.

Test results

During the testing of the application, multiple errors were detected. Most of these errors required a simple fix.

The first one of these errors was incorrect log line counting. On the application's initial load, when no user input has been detected, the length of the `CodeMirror` document (processed log) is `0`, which means that the line display will show `0\0`. The issue occurs when the user tries to filter this empty document, creating a copy, but this copy will contain an empty string as the first line, making the line counter think that the current amount of lines is set to `1`, out of the total of `0` lines.

The second error let the user edit the text in the `Filter mode`. This issue was caused by a missing hook which reacts to the change in the application modes, making it still possible to edit the text even tho the application was not in the `Edit mode`.

The third problem was related to the `Popup` component. The issue was an incorrect display of this popup when highlighting the text at the top of the inserted log, causing the `Popup` to leak through the top of the application's viewport. A new validation was added to check if the popup can fit the viewport. Now when the user highlights a top part of the inserted log, the popup will appear below the highlighted text in case it is too large to fit above it, preventing the popup from leaking past the viewport.

Finally, the last error that was detected through the E2E tests was an incorrect validation of `Parametrized regular expressions`. The `Parametrized regular expression` has a set syntax as follows:

```
%{r(VALUE):ID}
```

As the regular expression used to detect the correct syntax of the inserted `Parametrized regular expression` in the input filter was faulty, expressions without an `ID` were also validated.

6.2 User Testing

The testing was split into two parts. The first part was a quick introduction to the application using a brief video showing the usage of the application, mainly because many users may not be familiar with Grok patterns.

The second part consisted of three tasks throughout the application. The first task focused on filtering the inserted log through the input filter and selecting Grok sets. The second task focused on switching between the application's modes and creating log message patterns. The last task's goal was to filter the inserted log by using the created log message patterns and then exporting these patterns.

This testing aimed to see if users could finish the tasks in a reasonable time, give feedback on the application's usability, report observed bugs and suggest new features.

Based on the results, the average time it took to finish the first and the second task was 1-2 minutes. The second task took longer, with a timespan of 4-7 minutes. This time difference occurred mainly because of needing more explanation regarding the application's modes and how to switch between them.

In the third task, users were asked if they could see a duplicate match of the exact log message when filtering using the log message patterns. Most of them could see that the line was marked with a darker shade of red. However, they would only have realized this mark symbolized a duplicate match if they had been told before.

As seen in Figure 6.1, most users found the application easy to learn, and it took them around 11 minutes on average to get familiar with it.

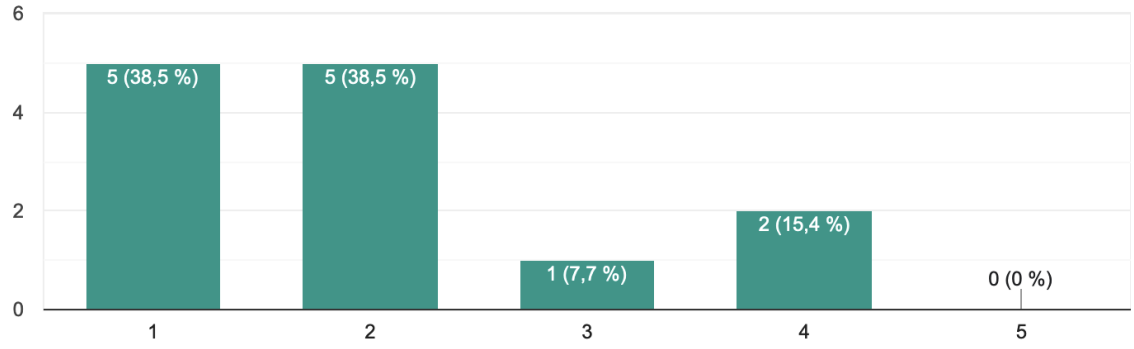

Figure 6.1: How easy was it to understand the application (1 – very easy, 5 – very difficult).

The following Figure 6.2, represents how user-friendly the application was. Most users found it relatively simple to use, with few exceptions. The issues that users experienced during testing could be reported as a feature request. These features are mentioned in the Table 6.3.

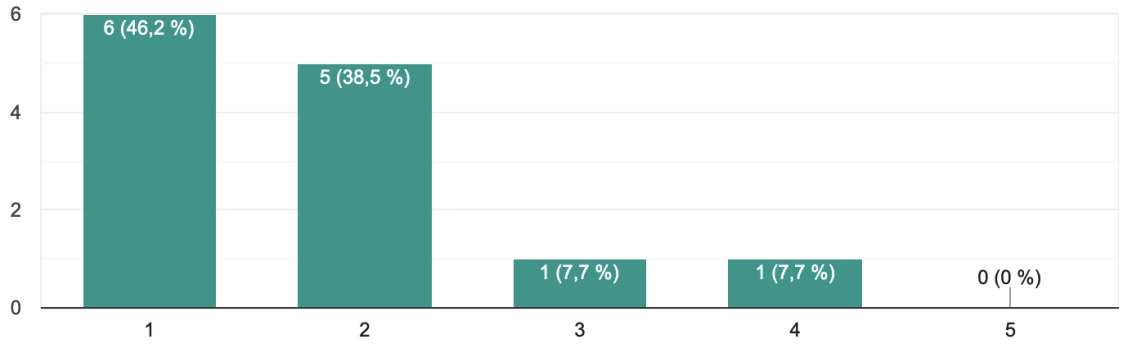

Figure 6.2: How user friendly is the application (1 – easy to use, 5 – rather confusing).

The results were deemed successful as the users involved in the testing had no previous experiences using Grok patterns (see Figure 6.3) and only got to know them from the video provided in the questionnaire.

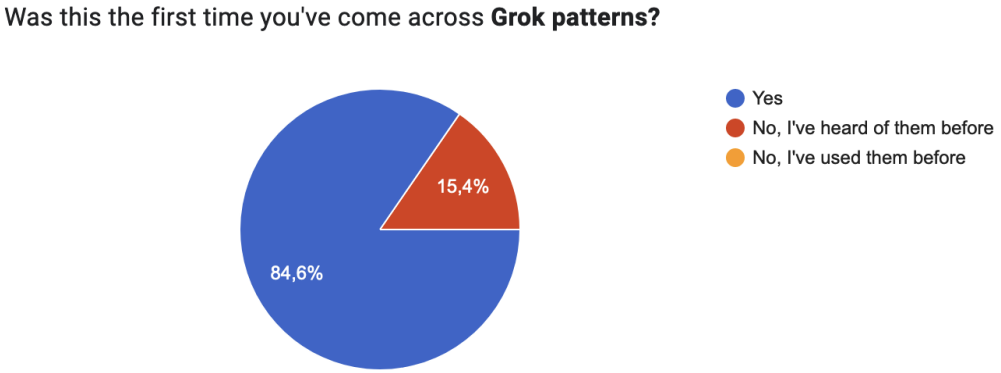

Figure 6.3: Familiarity of users using Grok patterns.

Requested features

During the testing, users could suggest features they would like to be added to the application in the future. Table 6.3 shows that some of these features have already been implemented based on user suggestions as they did not require in significant change in the application's code.

Requested feature	Implemented
A warning when moving a log message pattern to the input filter that is not empty, so it does not rewrite its value	Yes
A modal window with a short tutorial/explanation of the application	No
A possibility of combining the created log message patterns also using logical `AND`, as only logical `OR` is supported	No
Restricting the user from editing the exported YAML	Yes

Table 6.3: Suggested features for LogReaper.

The suggestion regarding a tutorial was not implemented as a short video was already provided. The second suggestion was not implemented mainly because it adds unnecessary functionality to the application, as Plogchecker matches every event with every log message.

Reported bugs

Few bugs were reported during the testing. Some of these bugs were purely cosmetic, other affected the application's functionality. One of the bugs was an incorrect display of the input filter and log message patterns, as they did not fully stretch to the entire length of the component's window on screens with larger resolutions.

The second bug was the inability to use the scroll bar inside of the popup, as the popup would immediately disappear upon clicking on it. The last bug occurred when the user edited the exported modal window. All of the reported bugs have been fixed.

Chapter 7

Conclusion

The bachelor's thesis aimed to design and implement a portable application for log processing using regular expressions and Grok patterns. After creating the initial design and researching existing similar applications, the application was implemented in React using Typescript. The design underwent multiple iterations to ensure that the final application fully met the set requirements and was simple to navigate and operate.

It was essential to research different regular expression engines and their limitations in processing regular expressions stored inside Grok patterns during the development. The *ECMAScript* standard defines the regular expression engine used in JavaScript. This *ECMAScript* regex engine can often not process regular expressions stored inside Grok patterns, as it does not support necessary modifiers, such as atomic groups or lookbehinds. The incompatibility issue led to the usage of Oniguruma, a regex library supporting the features of all well know regex engines. This library was mounted to the React application as a WebAssembly binary file and was used to process regular expressions that were also not compatible with the *ECMAScript* standard.

The following important part of the implementation was finding a text editor library that supports the set features. A CodeMirror editor was chosen as it is constantly updated, supports custom grammar for text highlights, and has many user-created extensions.

The application was tested using end-to-end tests through the Selenium framework to simulate user interaction. In the next testing phase, individual end users were given an instance of the application with specific tasks to complete. This testing provided feedback on the application's usability and suggestions for future features. Based on the feedback, the application's ease of use was deemed satisfactory.

The resulting application met all the predefined requirements, creating a portable web application. This application is used to filter logs in real-time using log message patterns created with regular expressions and Grok patterns. These log message patterns can be exported in YAML format and used in Plogchecker to define events to detect specific properties in the system under test.

Bibliography

[1] *Grok Processor Plugin* [online]. [cit. 2023-03-19]. Available at: https://www.elastic.co/guide/en/elasticsearch/reference/8.6/grok.html.

[2] *A Guide to Functional Requirements (with Examples)* [online]. [cit. 2023-03-25]. Available at: https://www.nuclino.com/articles/functional-requirements.

[3] *Material Design 2.0 - Introduction* [online]. [cit. 2023-03-19]. Available at: https://m2.material.io/design/introduction.

[4] *Material UI - Overview* [online]. [cit. 2023-03-19]. Available at: https://mui.com/material-ui/getting-started/overview/.

[5] *React – A JavaScript library for building user interfaces* [online]. [cit. 2023-03-19]. Available at: https://react.dev/.

[6] *Selenium WebDriver Documentation* [online]. [cit. 2023-03-19]. Available at: https://www.selenium.dev/documentation/webdriver/.

[7] *TypeScript for JavaScript Programmers* [online]. Microsoft Corporation, 2022 [cit. 2023-03-19]. Available at: https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html.

[8] Bartocci, E., Falcone, Y., Francalanza, A. and Reger, G. Introduction to Runtime Verification. In:. February 2018, p. 1–33 [cit. 2023-05-01]. DOI: 10.1007/978-3-319-75632-5_1. ISBN 978-3-319-75631-8. Available at: https://www.researchgate.net/publication/323082969_Introduction_to_Runtime_Verification.

[9] Bose, S. *What is end to end testing?* Feb 2023 [cit. 2023-03-21]. Available at: https://www.browserstack.com/guide/end-to-end-testing.

[10] CMCDragonkai. *Regular Expression Engine Comparison Chart.* 2018 [cit. 2023-03-19]. Available at: https://gist.github.com/CMCDragonkai/6c933f4a7d713ef712145c5eb94a1816.

[11] Danilec, A. *What Is JSX?* Jan 2021 [cit. 2023-04-19]. Available at: https://www.blog.duomly.com/what-is-jsx/.

[12] Devopedia. *Regex Engines* [online]. February 2022 [cit. 2023-03-19]. Available at: https://devopedia.org/regex-engines.

[13] Goyvaerts, J. *Regular expressions tutoriallearn how to use and get the most out of regular expressions*. Aug 2021 [cit. 2023-03-25]. Available at: https://www.regular-expressions.info/tutorial.html.

[14] Goyvaerts, J. *Using regular expressions with JavaScript*. Nov 2021 [cit. 2023-04-09]. Available at: https://www.regular-expressions.info/javascript.html.

[15] Haverbeke, M. *CodeMirror*. 2007–2023 [cit. 2023-03-19]. Available at: http://codemirror.net/.

[16] Poole, C., Prouse, R. et al. *NUnit Documentation*. 2021 [cit. 2023-03-19]. Available at: https://nunit.org/.

[17] Čaládi, F. *Ověřování parametrických vlastností nad záznamy běhů programů*. Brno, CZ, 2022. [cit. 2023-05-01]. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/23298/.

Appendix A

Test features and test coverage

```
Feature: Log message pattern operations

    Background:
    Given the User inserted a log

    Scenario: Creating a message pattern and naming it
    When the User creates a pattern in input filter
    And clicks filter export button
    And and sets a name for the created pattern
    Then then message a pattern is created

    Scenario: Selecting a message pattern
    When the User selects a message pattern by clicking a checkbox
    Then the log text will be filtered

    Scenario: Moving a message pattern into the Input Filter
    When the User moves a message parttern into the input filter
            by clicking the '<-' button
    Then message pattern value will be moved into the input filter
    And the message pattern will be removed from the Toolbar
```

Figure A.1: A feature file example used for the log message pattern operation test

Feature	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
Inserting log				X	X	X			X	X	X	X	X	X	X	X	X					X	X	X	X	X		X
Filter highlight		X	X															X	X	X	X	X	X	X	X	X		X
Text filtering	X						X	X				X	X	X	X							X						
Grok recommendations				X	X	X																						
Managing log message patterns																						X	X				X	
Log message highlight																								X				X
Pattern export																									X	X		
Grok set selection							X	X																				
Filtered log messages counter																X	X											
Application mode display	X	X	X							X	X						X											

Table A.1: Feature covering matrix

Feature file	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
Application-mode-display.feature	X	X	X																									
Grok-recommendations.feature				X	X	X																						
Grok-set-selection.feature							X	X																				
Insert-log.feature									X	X	X																	
Log-filtering.feature												X	X	X	X													
Log-message-counter.feature																X	X											
Filter-highlight.feature																		X	X	X	X							
Log-message-pattern-operations.feature																						X	X				X	
Message-highlighting.feature																								X				X
Log-message-pattern-export.feature																									X	X		

Table A.2: Feature-test matrix