FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Structured printing framework
Student:	Nilay Baranwal
Supervisor:	Ing. Konrad Siek, Ph.D.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2021/22

Instructions

Monitoring long-running processes takes the form of logs. However, the interface for printing is very rudimentary: printing a string. Sensible alternatives like graphs, diagrams, or tables require much extracurricular work.

The goal of this thesis is to create a framework to visualize data from running applications.

The framework will have 2 language-specific libraries that provide an API accepting data to display. API calls pass the data to a language-agnostic service that formats and displays it in a browser. The output format will be configurable and extendable. The display will show updates over time and accept data from multiple sources. Language-specific elements will be easily portable.

Milestones:
- gather functional and non-functional requirements
- review the space of existing solutions (logging utilities, LaaS)
- select technologies and design an architecture
- incrementally implement a prototype
- provide tests and documentation
- discuss the benefits of the solution

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague October 13, 2020

Bachelor's thesis

Down-Spiral: Structural Logging and Visualization

Nilay Baranwal

Department of Software Engineering
Supervisor: Ing. Konrad Siek, Ph.D.

February 14, 2021

Acknowledgements

I would like to thank and acknowledge my supervisor Mr. Konrad for all his support, guidance and valuable feedback throughout my thesis process and topic selection. This would not have been possible without him.
I would also like to thank my parents and sister for providing me with the constant support and the will power to continue. I would like to thank all friends and everyone present in my life for teaching me everything in and about life. I would also like to thank my roommate for listening to my countless proofreads, and pretending to be interested.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on February 14, 2021 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2021 Nilay Baranwal. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Baranwal, Nilay. *Down-Spiral: Structural Logging and Visualization.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021. Also available from: ⟨`https://github.com/alphasr/down-spiral`⟩.

Abstrakt

Logy v run-timu sledují využití systému, analýzu aplikací a pomáhají debagovat problémy. Většina logu se ukládá jako soubory nebo ekvivalentní databáze. Díky tomu je logování složité pro konkrétní případy, kdy se jedná o definované uživatelem datové typy nebo grafický výstup. Tato práce navrhuje řešení tohoto omezení logováním přímo v prohlížeči, které umožňuje vizualizovat logy jako tabulky, grafy, diagramy a vlastní HTML. To také přináší další výhody umožňující logování pomoci více procesů / více výstupů. Bakalářská práce analyzuje požadavky a poskytuje návrh takového systému, stejně jako důkaz implementace konceptu pomocí TypeScriptu, Node.js a ReactJS. Ukazujeme, že systém zlepšuje uživatelskou zkušenost, zatímco latence systému zůstává pro typické aplikace minimální.

Klíčová slova Logování, Webové aplikace, Webové služby

Abstract

Run-time logs keep track of system usage, application analysis, and help debug problems. Most logs are saves as files or equivalent databases. This makes logging complex for specific cases where user-defined data-types and graphical output are involved. This thesis proposes a solution to this limitation by enabling logging directly into the browser, which allows visualizing logs as tables, graphs, diagrams, and custom HTML. This also carries additional advantages of allowing multi-process/multi-output logging. The thesis analyzes the requirements and provides a design for such a system, as well as a proof of concept implementation using TypeScript, Node.js, and ReactJS. We demonstrate that the system improves user experience while the latency of the system remains minimal for typical applications.

Keywords Logging, Web applications, Web services

Contents

Introduction **1**

1 Related Work **5**
1.1 Logging . . . 5
1.2 Logging in Web Services . . . 6
1.3 Log V-management Systems . . . 7
1.4 Log Analysis . . . 10
1.4.1 Machine Learning . . . 10
1.4.2 Facility Space Exploration . . . 11
1.4.3 Cyber Security . . . 11
1.5 Conclusion . . . 11

2 Requirement analysis **13**
2.1 Use case analysis . . . 13
2.2 Non-Functional requirements analysis . . . 43

3 Design and implementation **45**
3.1 Design . . . 45
3.2 Implementation . . . 47
3.2.1 Technology Selection . . . 47
3.2.2 Implementation details . . . 50
3.2.2.1 Thin client . . . 50
3.2.2.2 React app . . . 52
3.3 Lessons learned . . . 55

4 Evaluation **57**
4.1 Correctness . . . 57
4.2 Tests . . . 57
4.2.1 Results . . . 58
4.2.2 Discussion . . . 59

5 Conclusion **63**
5.1 Impact 64
5.2 Future work 64

Bibliography **65**

A Acronyms **69**

B Contents of enclosed CD **71**

C Installation and user guide **73**
C.1 Installation Guide 73

List of Figures

1.1 Example of NEL handling connectivity loss [1]. 7

3.1 Architecture of down-spiral 46
3.2 Class Diagram of the Thin Client. 51
3.3 UML Sequence Diagram: sending data to the server. 53
3.4 Component Diagram of the react-app. 54
3.5 UML Sequence Diagram of updating data in the database and the application 56

4.1 Simple Printer short messages latency 59
4.2 Simple Printer long messages latency 60
4.3 Table Printer short messages latency 60
4.4 Table Printer long messages latency 61
4.5 HTML Printer short messages latency 61
4.6 HTML Printer long messages latency 62
4.7 Graph Printer short messages latency 62

List of Tables

3.1 Component message format. 55

4.1 Non-functional requirements 57
4.2 Use cases 58
4.3 Latency evaluation results. 59

Introduction

A well designed logging system is *sine qua non* for all programmers. Keeping logs can save a lot of time and effort. It is also an important and invaluable utility for many specific applications as well as the developer.

Log data is mainly used by developers to discover and analyze functional problems in their applications both during application development and during the maintenance phase. Filtering data logs and managing collection logs is of high significance [2].

Apart from the working programmer, logging is also a mainstay of major fields of research in computer science, including data logging are cyber security [3], machine learning [4], and facility space exploration [5] to name just a few. These fields use logs to collect historical information to discover rules of behavior within complex systems. While logging is omnipresent and useful, there are aspects of it that warrant consideration for improvement and upgradation to the latest technologies.

One significant limitation of data loggers is that the output format is very rudimentary: a string. While there may be attached metadata (eg. timestamps, host name), most loggers ultimately expect each logged event to be a line of text [6, 7, 8]. This gives logs generality, but makes certain types of information (eg. timing, graph traversal) difficult to follow at-a-glance, especially in real time. This is additionally compounded by the sheer number of logging events that often bombard the output. While infrastructure can be set up to visualize these types of logs, this requires a solution that parses the log and generates graphics. This solution can be either hand crafted (eg. an R or Python script) or a visualizer can be added to the stack [9, 10] [11]. In either case the programmer must set up additional software, often write non-reusable purpose-built scripts, and format the log output to match the input expected by the visualizer.

We suggest that integrating richer output and visualization directly into the logging process would be friendlier to the user, since the user could then view diagrams, graphs, and tables, etc. as output of the logger and view them

in real time. Web technologies have impacted the IT field in a positive way and has made it easier for developers to concentrate on UI/UX design. Web technologies can help with the visualization of collected or real-time data, which removes all the extra work and enables the user to monitor details and changes easily [10]. We see how the world is adapting more than ever to the web. This makes transition of display from the console or file to the modern web browser, that supports better documentation viewing experience in the form of an HTML document. Benefits also include interaction at run-time, when viewing the document, without the need to regenerate it for real time usage.

Another problem loggers face is the difficulty with which outputs from different processes are combined. While it is possible to combine outputs of logs from one host or operating system, either by appending to the same log file, or combining multiple log files on the basis of timestamps, this is more complex in case of

A more sophisticated client-server infrastructure would lend itself to combining inputs from multiple sources regardless of origin. This sort of infrastructure can be found in logging as web services (WS) [12] [13], where clients send requests to the logging server with logger information using a network protocol. However, while these work in WS environments, they do not easily extend to general use cases, since they native bindings in native languages, instead relying on the universality of network communication.

We suggest that a similar client-server architecture can be introduced to general purpose loggers by providing bindings. The architecture will allow composing multiple sources as well as logging to multiple outputs within a single process and centrally caching logs. Native bindings serving as user-facing facades will obscure the underlying network protocol from the programmer. This layer can also then provide transparent serialization of language-idiomatic objects into a format that is understandable to the logger. Since it is impossible to provide bindings in every single language, new bindings should be easy to write to allow quick introduction of the logging framework into new languages. Porting a thin client in the client-server logger is easier than porting an entire local logger, since the client is a a much smaller, much more flexible, and much less sophisticated code base [14].

Any suggestion towards improvement should not detract from existing capabilities that loggers posses. A logging system should be highly reliable, should scale in terms of inputs and multiple simultaneous inputs, and must maintain high availability. The logging system should also be completely decoupled from the parent system to ensure no blocking operations occur.

Given the discussion above we stipulate the following.

Thesis. *It is possible to propose an integrated structured logger and visualizer for native languages that displays events in the browser in real time. Such a browser should be user friendly to its target user base—programmers. I.e.,*

a. extendable—*the user can extend the capabilities of the logger by easily plugging in custom visualization modules,*

b. flexible input format—*users log data without worrying about underlying data format as the API takes care of the serialization,*

c. updated in real time—*the visualization is continuously updated as new events are being logged,*

d. usable anywhere in the program—*the user can send an event to a log anywhere in the code regardless of context (especially, does not depend on passing around a logger object),*

e. portable—*the logger provides native language bindings which are easily portable to other languages and the documentation contains a guide to porting bindings,*

f. configurable—*the user can be configure the logger system with custom views and parameters as per user preference,*

g. multiple simultaneous inputs—*the user can specify different instances of a logger to receive events from a single process,*

h. multiple simultaneous outputs—*the logger can receive events from multiple processes, possible on remote machines,*

i. batteries included—*the logger provides reasonable basic defaults for visualizing tables, graphs, and custom HTML. Provides out-of-box visualization support.*

This work examines the thesis statement and supports it by providing a design and a proof-of-concept implementation of a logger system satisfying the proposition. We do this as follows. First, in Chapter 1 we show that data loggers either have no visualization or if in-house this feature the solution usually requires enterprise grade tools, which are not user-friendly and require substantial amount of resources and configuration. We also go through the important applications of data logging in different fields. Next, in Chapter 2 we examine the requirements both functional and non-functional, to solve the problems. We describe the design of the solution in Chapter 3 and give an overview of the architecture of the data logging system. This chapter also contains details and justification behind the use of specific technology to solve the problem. We also provide the description of the implementation details of the thin-client architecture, and the data logging system. Then, in Chapter 4 we evaluate our implementation using the technologies we selected to satisfy the functional and non-functional. Finally, we discuss our conclusions in Chapter 5.

In addition, we also provide the following appendices that supplement this work. Appendix A lists and explains acronyms used in this text. Appendix B lists the contents of the accompanying CD. Appendix C provides an installation guide and usage manual for the software developed through this work.

Chapter 1

Related Work

Data logging is still the most appropriate way to record data for analysis and further pattern matching structures, which makes it so prominent in all fields involving IT. There is research in aspects of data logging and how it can impact other fields. We review latest and best practices related to logging in different fields with various applications.

1.1 Logging

Logging refers to systems that provide infrastructure to other systems to allow them to easily, concisely and effectively write debug information to a consistent sink. This sink is usually an append-only file on the local file system to which each new logging event is written as a new line in a semi-structured format (timestamps, process names, and log levels are usually attached to the payload automatically). Data Loggers have different features and some are dedicated to a specific language too. A data logger usually saves the data logs to a file or a database, these are common practices followed by default for most of them. There are many frameworks that provide such a mechanism. Among them Log4j,[1] Log4net,[2] Tinylog,[3] SLF4J,[4] Syslog,[5] LOGBack,[6] are a few data logging frameworks.

Log4j is an open source logging framework built in Java that can output logs to the console or database or a log file. Additionally it has different levels of logging and allows logging on a class-by-class basis. The Log4j 2 API provides the interface that applications should code to and provides adapter components required for implementers to create a custom logger implemen-

[1]https://logging.apache.org/log4j/2.x/
[2]https://logging.apache.org/log4net/
[3]https://tinylog.org/v2/
[4]http://www.slf4j.org/
[5]https://en.wikipedia.org/wiki/Syslog
[6]http://logback.qos.ch/

tation. Log4net is an open source framework for .NET runtime logging. It needs to be configured separately and logs are saved to a text file. Logback is an intended successor to Log4j. LogBack is an open framework built in Java similar to Log4j. LogBack has a native SLF4J API that allows it to switch back and forth between logback and other logging frameworks. TinyLog is a framework that can output logs to consoles or SQL database using JBDC or to a file, format and stream can be specified. TinyLog also has the capability of writing multiple instances of the application to the same file. SysLog(System Logging Protocol), is used to send system log or events to a specific server. It is used to collect data from multiple devices logs from different machines in a central location for monitoring and reviewing. Unix follows SysLog for internal log management.

Since the logger frameworks output to files or databases, they do not satisfy the problem set out by the thesis statement in thesis item i.

1.2 Logging in Web Services

Logging data is also a mainstay in context of web services, where the log is created on a file system that is local to the logger process, but remote to the processes that use the logger through a network API. It is an extension of the logger paradigm to distributed systems. Web services often require the logging and analysis of large amounts of data relating both to the service endpoints, as well as the health and status of the web services' network itself. In addition to this, for analysis and report messages associated with web service are logged entirely. In [12], the authors propose such a system that can log each activity of the web service instead of just logging headers and just the response. There are control points for web services that can be configured to flag messages, and this determines the logging instance that will used. Interesting finds in the paper include monitoring of network devices through logging, data logging node supporting a plurality of network devices and data logging service type, and a control point monitoring system to enable logging. This message logging system can be incorporated into a variety of network architectures, as it generates a SOAP fault for eventual receipt by the requester. The messages are classified into inbound and outbound, and logging policies are applied to filter out the messages.

NEL (Network Error Logging) is a planet-scale, client-side, network reliability measurement system [13]. NEL is implemented in Chrome and a W3C standard.[7] The reports generated are similar to web server logs, but include information about failed requests that never reached the serving infrastructure. This takes care of service inaccessibility, which could be due to DNS failure, DNS hijack, Service faulty/down to name a few. Specifically, NEL expands on the current approach of leveraging user reports, for failed requests to

[7]https://www.w3.org/standards/

the service. An example of this is given in Fig. 1.1. When a client successfully connects to the service, NEL is activated. If there is a failed request to the service, NEL report is sent to the NEL controller on the service side. NEL alone was not enough to diagnose this issue because it gave no information about the location of the problem other than the set of users affected. Other tools, like trace-route, are used to identify which network links were impacted and that most, if not all, of the Internet traffic in that country transits through one ISP.

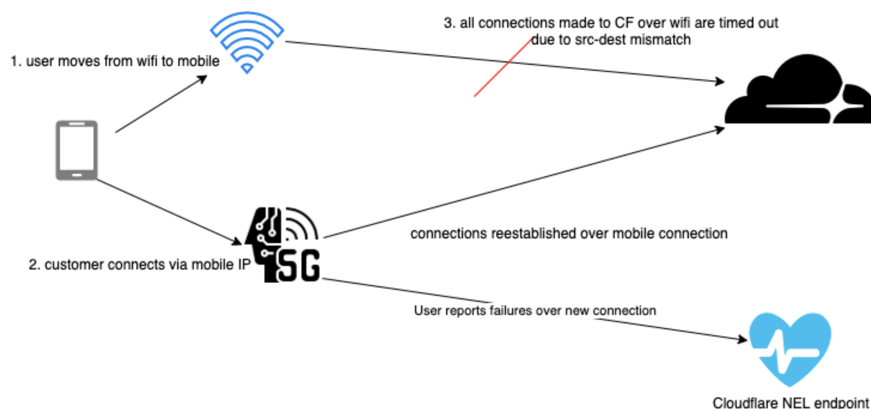


Figure 1.1: Example of NEL handling connectivity loss [1].

We reviewed web service loggers, and they behave in the similar way as local loggers which includes data logs as files or a database where the event payload is a key-value, and the value is a string, each of payloads ultimate lead to the same structure of a rudimentary string.

1.3 Log V-management Systems

Once a log is created, finding data in it can be difficult as there are often thousands of lines of text to comb through. There are some solutions that use the knowledge of either the logging format to color each line of the log or tools will provide visualization for log volume for administrators to be able to see what is going on in the system at a glance. Examples of visualization data loggers include GrayLog,[8] Nagios,[9] LOGalyze,[10] Elastic Stack,[11] Fluentd.[12]

GrayLog is developed in Java and supports specific configuration that enables the server-client communication. GrayLog is a whole log analysis suite that helps find trends, statistical deviations, and specific scenarios within a

[8] https://www.graylog.org/
[9] https://www.nagios.org/
[10] http://www.logalyze.com/
[11] https://www.elastic.co/elastic-stack
[12] https://www.fluentd.org/

system log. Clients push the log data to the server and it analyzes and stores log events in database. It also houses a log management dashboard and a web UI. Very basic GUI does not support advanced graphing functionality and user needs to use Grafana and/or Kibana to achieve this. Plugins and extensions are difficult to install. The main problem that GrayLog has, is it's an enterprise grade solution that doesn't seem to be made for individual programmers. It also a memory hog and needs at-least 2GB of RAM. GrayLog does not provide good enough visualization without significant configuration effort (required by thesis item i in the thesis statement). It also does not provide native bindings (required by thesis item e). Therefore, GrayLog does not satisfy our requirements. GrayLog stores all data logs in ElasticSearch, which gives it massive flexibility over a third-party search tool. This gives the user the power to create custom search queries. It has both options available for users i.e. open-source for individuals and enterprise for organization or individuals.

Nagios is an open source host/service/network monitoring system written in C, with an intuitive and simple-to-use interface. Configuring Nagios is difficult and can only be done using command-line. There are many features and modules to configure and select to tailor it to user's need, which makes it so complex to use. It is also prone to redundancy and has no out-of-the-box support for graphs and trends. Therefore, it does not satisfy the requirements in the thesis (thesis item i, thesis item a). Users are readily writing plugins for data loggers, for new languages. [13] They also wrote different plugins for graphs and trends. It is easy to write plugins and already available just a few lines long which makes it for casual programmers too. It's an enterprise grade tool so the configuration is complex, and it's difficult to construct a data logger tailored to the user's needs. This would require configuration of out many many small packages. Nagios is used by managers and system administrators including capacity planning, proactive alerting, service-level agreements, etc.

LOGalyze is an open source log analyzer and network monitor. It supports devices, windows hosts, and Linux/Unix servers with real-time event detection. From within the LOGalyze web interface, you can run dynamic reports and export them into Excel files, PDF, or other formats. In contrast it takes almost an hour to configure. Therefore, it does not satisfy the requirements in the thesis (thesis item e, thesis item g). Since LoGalyze, is a light-weight tool for data logging in a local setup it is very popular among users. It is an elegant alternative solution to enterprise grade data loggers. It also supports cross OS support.

Elastic Stack comprises of three projects which are Elasticsearch, Logstash, and Kibana. Elasticsearch is a distributed search engine with highly refined analytic capabilities, Logstash is a data-processing pipeline that collects data

[13] https://github.com/mpounsett/nagiosplugin/tree/master/nagiosplugin

and delivers it to Elasticsearch, and finally Kibana is a visualization platform built expressly for Elasticsearch. It is an enterprise grade solution needs technical expertise to be configured in the existing system as it has a steep learning curve. Kibana has no default logging dashboard. Also requires extensive management and maintenance. Therefore, it does not satisfy the requirements in the thesis (thesis item e). The ELK stack is an enterprise grade technology stack that is either used in some combination or using some modules for a specified purpose. ELK is extensive has three different main components and then multiple sub-component for each. The search client available in elastic in Python[14] and Rust.[15] It usually requires a team to set it up and maintain the whole ELK stack.

Fluentd is a cross platform open-source data collection software project originally developed at Treasure Data. All plugins decentralized and made in Ruby, which makes it easier to add custom plugins. It has built in parser for XML, JSON, CSV, and regular expressions. Fluentd has a simple design and is robust. Cons include a lack of support for multi-threading and it is very challenging to configure, and requires a lot of work and some expertise. Therefore, it does not satisfy the requirements in the thesis (thesis item e). Fluentd provides visualization using either Prometheus or Elasticsearch. Since elastic stack has built-in Kibana that can be used by Fluentd. The modules for visualization are easy to write and brief. The visualization supports graphs and trends with Kibana and Prometheus has built in 8 metric lists on which the user can filter and display data. All the metric lists are described in detail here [16] .

Apache Chainsaw can read any regular text log file, including those created by SLF4J and logback[17]. Chainsaw's initial configuration dialog makes it easy to help the user to process log file - they can can specify the log file you want to process, and then the format of the file, and Chainsaw will start tailing the file. Remote events are can be received using Log4j [6]. Chainsaw has a responsive GUI, and determine the frequency of updates which is an interesting feature as it can deal with real-time data. Color coding is also a feature, to classify logs depending on the the type of log event and it also has contains HTML based documentation. Therefore, it does not satisfy the requirements in the thesis (thesis item e, thesis item c, thesis item b). There is no utility to plot graphs and if the frequency of updates is high, it will difficult for the users to interpret the data. This creates a problem with real-time high frequency flow of data. Since it is free very casual programmers can also use it, and it provides the option of color coding the data to organize and filter it.

In [10], the authors propose the use of visualization for security experts to visually explore numerous types of log files through relevant representations

[14] https://github.com/elastic/elasticsearch-py
[15] https://github.com/elastic/elasticsearch-rs
[16] https://docs.fluentd.org/v/0.12/articles/monitoring-prometheus
[17] http://logback.qos.ch/

using *ELVIS* (Extensible Log VISualization). ELVIS is a security-oriented log visualization tool. It is implemented using standard web technologies: HTML5, JavaScript, CSS, SVG and the D3.chart[18] library for reusable and extensible charts. ELVIS is useful to visually explore security log files, allowing the user to quickly notice relevant facts. It has limitations which include that is not real-time and is used for viewing existing logs. Multiple sinks cannot be used for data logging. Therefore it does not satisfy the requirements extendable (thesis item a), multiple output sinks (thesis item h), and real-time visualization (thesis item c).

There are many data log analysis frameworks that are technically capable of solving the problem described in the introduction, these are very large frameworks with multiple modules and big resource requirements. These are usually not free to use since they are enterprise grade. They are aimed for big businesses, server clusters and large distributed environments with many simultaneously many running processes. The amount of configuration and expertise required to use them is preventative for most casual developers who want to examine medium or smaller systems and can be used by a very small segment. Therefore, even though these systems have many capabilities that satisfy the thesis statement, they fail to provide a good user experience for the intended use.

1.4 Log Analysis

A related field is analysis of existing logs to discover and analyze the properties of the log or the system. Two prominent fields that involve log analysis are: Machine Learning and Cyber Security. Although this work is not directly related to the problems described in the introduction we include it out of interest in applications of logging.

1.4.1 Machine Learning

Utilizing existing data logs in machine learning is very useful for improve upon key factors and make making few errors. Log analysis can help the developers understand the system better and can also observe behavioral changes in user. In [15] the authors discuss the effects of machine learning done on data logs. To main goal is to analyze listening data logged in speech processors. This data is used to predict early auditory and linguistic skills in toddlers [19] . One year cumulative listening time to speech in silence and in noise was analyzed. To conclude the author mentions that the listening environment can is huge influence on toddlers while growing up. A data logging system plays a crucial role role in predict outcomes related to the case.

[18] https://d3js.org/

[19] 0-2 years

1.4.2 Facility Space Exploration

Intelligent building facility space programs are influencing and modifying everyone's life with or without us knowing it. Data logs are especially useful in this case. Logged data can be used to create virtual environment or simulations, which can help us to analyze and modify current technologies [5] and set great safety standards so there is minimum damage. The author proposes the invention of a device that has built-in portable logger for room identification and dimensions in a facility. This paper provides us with an interesting view of how important a portable logger is in real world application.

1.4.3 Cyber Security

Paper [3] presents us with the case of cyber security and how data logs can be utilized, to detect intrusion. This paper documents two approaches to data sharing for the industrial control system IDS research community. First one, is network traffic data logs captured from a gas pipeline is presented. The gas pipeline data log was captured in a laboratory and includes artifacts of normal operation and cyber attacks. Second, an expandable virtual gas pipeline is presented which includes a human machine interface, programmable logic controller, Modbus/TCP communication, and a Simulink based gas pipeline model. The virtual gas pipeline provides the ability to model cyber-attacks and normal behavior. Thirty-five labelled data logs that recorded under cyber attacks for detection.

1.5 Conclusion

After an overview of the various fields and applications of loggers, we find that none of them completely satisfy the problem set out in the thesis statement. Of all the loggers we examined in Sec. 1.1 we found that none support visualization as they all output logs in text form or write to a database. In addition, most loggers are not interactive, in that their output is difficult or impossible to follow in real time.

While visualization can be added onto one of the existing loggers, our examination of Sec. 1.3 points to that either it is not sufficient on its own or requires extensive work to configure it. The examples reviewed bring to light the fact that data loggers that support visualization as an out-of-the-box feature are almost rare, not portable, and requires expertise.

Chapter 2

Requirement analysis

We have analyzed and reviewed existing frameworks and software, but none of them solve our problem. In preparation of solving this problem using current web technologies we analyze the requirements for a framework that tries to solve this problem and draw up a list of use cases and a list of non-functional requirements for our intended solution.

To paraphrase, the goal of this work is to to create a rich, capable, easy-to-use logger for use by programmers. Doing so translates to the following specific requirements. We will specify our requirements in detail in the form of use cases for functional requirements and in the form of a list of non-functional requirements.

2.1 Use case analysis

Use cases are described as a list of steps undertaken by system actors. The system has the following actors in all use cases: the *user* who is the user that uses the logger in their code, the *server* representing the remote part of the logging system, and the *browser*. The use cases provide examples of source code and results which are meant to be only illustrative rather than comprehensive specifications.

We sort our list of functional requirements into categories based on common functionality: logging events, logging types, custom modules, and more. The first group of requirements we present describes the behavior of the system in response to events being logged during a process's run-time. This universally includes initial setup of a connection the system, as well as sending events to the logger, which are then visualized in specific, various formats. This group includes UC 1, UC 1b, UC 1c, UC 5, and UC 6.

The second group discusses logging to different output formats such as tables, HTML, graphs and many more. This universally includes initial setup of a connection the system, as well as sending events to the logger, which are

then visualized in specific, various formats. The group includes UC 2, UC 2b, UC 3, UC 4, UC 7, and UC 7b.

The third group discusses extensibility and user created plugin support for data logging and setup of the view in the browser. This universally includes initial setup of a connection the system, as well as sending events to the logger, which are then visualized in specific, various formats. This group includes UC 8 and UC 8b .

Use Case 1: Simple logging

1. *User* writes a long running program, mocked up by this pseudo-code:

```
for i in 0.. {
    result := run long running task
}
```

2. *User* imports package or library in language appropriate manner

```
import logging library
for i in 0.. {
    result := run long running task
}
```

3. *User* configures loggers

```
import logging library
configure_logger("localhost", 3000, "results", SIMPLE_PRINTER)
for i in 0.. {
    result := run long running task
}
```

4. *User* logs data from running program

```
import logging library
configure_logger("localhost", 3000, "results", SIMPLE_PRINTER)
for i in 0.. {
    result := run long running task
    result_label := concat("result ", i)
    result_line = concat(result_label, ": ", result)
    log("results", result_line)
}
```

5. *User* turns on browser pointed at the logging server at address indicating a concrete logger `"localhost:3000/simple-printer"` and GUI has **ID ”results”** on a scroll-able place with select and delete button.

6. *User* starts program.

7. *Program* runs and sends configuration for ”results” in the form of a JSON object to server at `localhost:3000`.

8. *Server* receives configuration information, performs configuration.

9. *Program* continues running and sends data to `"results"` as JSON.
10. Server receives data, performs visualization.
11. *Browser* appends a line of text showing `"result n:  ..."` where `"..."` indicates the result of the long running task and `"n"` is the number 0 for the first line, and `1, 2, 3, ...` for successive lines. Example after 4 iterations:

```
result 0: asia
result 1: basia
result 2: casia
result 3: dasia
```

12. Proceed to 10.

Use Case 1 Extensions:

Use Case 1b: Turning it on later

1. *User* writes a long running program, mocked up by this pseudo-code

```
for i in 0..3 {
    result := run long running task
}
```

2. *User* imports package or library in language appropriate manner

```
import logging library

for i in 0..3 {
    result := run long running task
}
```

3. *User* configures loggers

```
import logging library
configure_logger("localhost", 3000, "results", SIMPLE_PRINTER)
for i in 0..3 {
    result := run long running task
}
```

4. *User* logs data from running program

```
import logging library
configure_logger("localhost", 3000, "results", SIMPLE_PRINTER)
for i in 0..3 {
    result := run long running task
    result_label := concat("result ", i)
    result_line = concat(result_label, ": ", result)
    log("results", result_line)
}
```

5. *User* starts the program.
6. Program runs and sends configuration for `results` to server at `"localhost:3000/simple-Printer"` as JSON.
7. *Server* receives configuration information, performs configuration.
8. *Program* continues running and sends data to `"results"` as JSON.

9. *Server* receives data, performs visualization.

10. Proceed to 6 until program performs all its iterations

10.a *User* allows program to finish execution completely

11. *User* turns on browser pointed at the logging server at address indicating a concrete logger `"localhost:3000/simple-printer"` and GUI has **ID "results"** on a scroll-able place with select and delete button.

12. The *Browser* displays the data logged until that point.

```
result 0: asia
result 1: basia
result 2: casia
result 3: dasia
```

13 . Proceed to 9.

Use Case 1c: Log level

---

1. *User* writes a long running program, mocked up by this pseudo-code:

```
for i in 0..3 {
    result := run long running task
}
```

2. *User* imports package or library in language appropriate manner

```
import logging library

for i in 0..3 {
    result := run long running task
}
```

3. *User* configures loggers with a higher minimum log level (this works for any logger). Available log levels from lowest to highest: `DEBUG, LOG (default), WARN, CRITICAL`

```
import logging library
configure_logger("localhost", 3000, "results",
                 SIMPLE_PRINTER, LogLevel.WARN)
for i in 0..3 {
    result := run long running task
}
```

4. *User* logs data from running program with different log levels: warn for `even i's` and log for `odd i's`:

```
import logging library
configure_logger("localhost", 3000, "results",
                 SIMPLE_PRINTER, LogLevel.WARN)
for i in 0..3 {
    result := run long running task
    result_label := concat("result ", i)
    result_line = concat(result_label, ": ", result)
    if i is odd {
        log("results", result_line)
    } else {
        warn("results", result_line)
    }
}
```

5. *User* turns on browser pointed at the logging server at address indicating a concrete logger `"localhost:3000/simple-printer"` and GUI has **ID "results"** on a scroll-able place with select and delete button.
6. *User* starts program.
7. *Program* runs and sends configuration for `"results"` to server at `localhost:3000` as JSON.
8. *Server* receives configuration information, performs configuration.
9. *Program* continues running and sends data to `"results"` as JSON.
10. *Server* receives data, performs visualization.
11. For `even i's` the browser appends a line of text showing `"result n: ..."` where `"..."` indicates the result of the long running task and `"n"`is the number 0 for the first line, and `1`, `2`, `3`, `...` for successive lines. For odd `i`'s the result is omitted, because its log level is below warn. Example after 4 iterations:

```
result 1: basia
result 3: dasia
```

12. Proceed to 9

Use Case 2: Logging to table

1. *User* writes a long running program, mocked up by this pseudo-code:

```
for i in 0.. {
    result := run long running task
}
```

2. *User* imports package or library in language appropriate manner

```
import logging library
for i in 0.. {
    result := run long running task
}
```

3. *User* configures loggers

```
import logging library
configure_logger("localhost", 3000, "results", TABLE)
for i in 0.. {
    result := run long running task
}
```

4. *User* logs data from running program, passes a dictionary-like structure containing the number of the iteration for column "i" and the value of the result to column "result"

```
import logging library
configure_logger("localhost", 3000, "results", TABLE)
for i in 0.. {
    result := run long running task
    log("results", ["i": i, "result": result])
 }
```

5. *User* turns on browser pointed at the logging server at address indicating a concrete logger `"localhost:3000/table-printer"` and GUI has **ID "results"** on a scroll-able place with select and delete button.

6. *User* starts program.

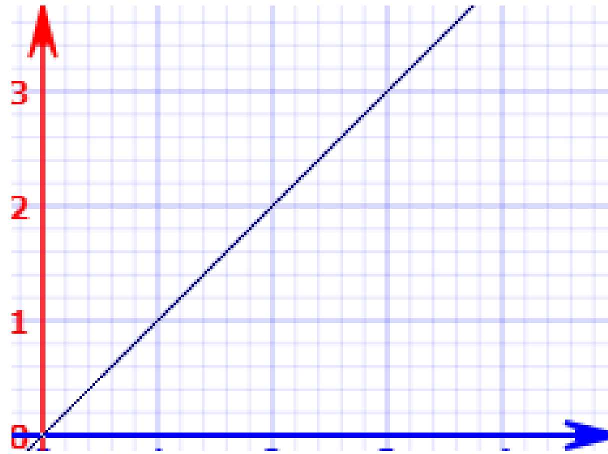
7. *Program* runs and sends configuration for "results" to server at `localhost:3000` as JSON.

8. *Server* receives configuration information, performs configuration.

9. *Program* continues running and sends data to `"results"` as JSON.

10. Server receives data, performs visualization.

11. If there was no data, the browser displays two column headers: `"i"` and `"result"` (in that order). Then, the browser appends a row containing a number:



n	result
0	asia
1	basia
2	casia
3	dasia

12. Proceed to 10.

Use Case 2 Extensions:

Use Case 2b: Logging to table with heterogeneous columns

1. *User* writes a long running program, mocked up by this pseudo-code

```
for i in 0..3 {
    result := run long running task
}
```

2. *User* imports package or library in language appropriate manner

```
import logging library
for i in 0..3 {
    start := current time
    result := run long running task
    duration := concat(current time - start, "s")
}
```

3. *User* configures loggers to display data as a table

```
import logging library
configure_logger("localhost", 3000, "results", TABLE)
for i in 0..3 {
    start := current time
    result := run long running task
    duration := concat(current time - start, "s")
}
```

4. *User* logs data from running program, passes a dictionary-like structure containing the number of the iteration for `column "i"` and the value of the result to column `"result"`, if `i is odd`, in addition, the dictionary contains a `column "elapsed"` containing the duration value

```
import logging library
configure_logger("localhost", 3000, "results", TABLE)
for i in 0.. {
    start := current time
    result := run long running task
    duration := concat(current time - start, "s")
    if i is even {
        log("results", ["i": i, "result": result, ""])
    }
    if i is odd {
```

```
        log("results", ["i": i, "elapsed": duration,
        "result": result, ""])
    }
}
```

5. *User* turns on browser pointed at the logging server at address indicating a concrete logger `"localhost:3000/table-printer"` and GUI has **ID "results"** on a scroll-able place with select and delete button.

6. User starts program.

7. Program runs and sends configuration for `results` to server at `"localhost:3000/table-printer"` as JSON.

8. *Server* receives configuration information, performs configuration.

9. *Program* continues running and sends data to data to `"results"` containing two columns as JSON.

10. Server receives data, performs visualization.

11. If there was no data, the browser displays two column headers: `"i"` and `"result"` (in that order). Then, the browser appends a row containing a number and the result data in respective columns: `"n"` and `"result"`. Example for 1st iteration:

n	**result**
0	asia

12 *Program* continues running and sends data to `"results"` containing three columns.

13. *Server* receives data, performs visualization.

14. The *Browser* adds another column header: `"elapsed"` (mind order). Then, the browser appends a row containing a number and the result data in respective columns: `"n"`, `"elapsed"` and `"result"`. Example after 4 iterations:

n	**elapsed**	**result**
0		asia
1	42s	basia
2		casia
3	13s	basia

15. Proceed to 9.

Use Case 3: Logging to graph

1. *User* writes a long running program, mocked up by this pseudo-code:

```
for i in 0.. {
    result := run long running task
}
```

2. *User* imports package or library in language appropriate manner

```
import logging library
for i in 0.. {
    result := run long running task
}
```

3. *User* configures loggers, the user has different options for graph plotting. `BAR`, `SCATTER`, `RADAR`, `LINE`, `DOUGHNUT`, `POLAR_AREA`, `BUBBLE` are default available options for graph types.

```
import logging library
configure_logger("localhost", 3000, "results", GRAPH,
                 SCATTERPLOT)
for i in 0.. {
    result := run long running task
}
```

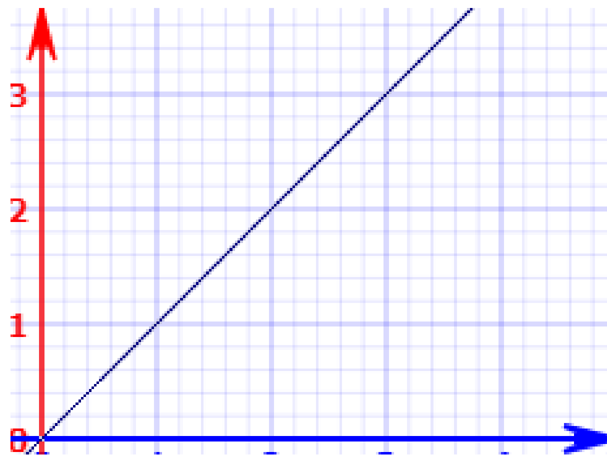
4. *User* logs data from running program, passes a dictionary-like structure containing the number of the iteration for coordinates ”x” and ”y” and the value of the result to as a ”label”

```
import logging library
configure_logger("localhost", 3000, "results", GRAPH,
                 SCATTERPLOT)
for i in 0.. {
    result := run long running task
    log("results", ["x": i, "y": i, "label": result])
}
```

5. *User* turns on browser pointed at the logging server at address indicating a concrete logger `"localhost:3000/graph-printer"` and GUI has **ID ”results”** on a scroll-able place with select and delete button.

6. *User* starts program.

7. *Program* runs and sends configuration for ”results” to server at `localhost:3000` as JSON.
8. *Server* receives configuration information, performs configuration.
9. *Program* continues running and sends data to `"results"` as JSON.
10. Server receives data, performs visualization.
11. *Browser* adds a point to the scatter-plot at coordinates (0,0), (1,1), ... with the appropriate label. Example after 4 iterations:



12. Proceed to 10.

Use Case 4: Logging custom HTML

1. *User* writes a long running program, mocked up by this pseudo-code:

```
for i in 0.. {
    result := run long running task
}
```

2. *User* imports package or library in language appropriate manner

```
import logging library
for i in 0.. {
    result := run long running task
}
```

3. *User* configures loggers

```
import logging library
configure_logger("localhost", 3000, "results", HTML)
for i in 0.. {
    result := run long running task
}
```

4. *User* logs data from running program

```
import logging library
configure_logger("localhost", 3000, "results", HTML)
for i in 0.. {
    result := run long running task
    result_line = concat("<s>", result, "</s>")
    log("results", result_line)
}
```

5. *User* turns on browser pointed at the logging server at address indicating a concrete logger `"localhost:3000/html-printer"` and GUI has **ID ”results”** on a scroll-able place with select and delete button.

6. *User* starts program.

7. *Program* runs and sends configuration for ”results” to server at `localhost:3000` as JSON.

8. *Server* receives configuration information, performs configuration.

9. *Program* continues running and sends data to `"results"` as JSON.
10. Server receives data, performs visualization.
11. *Browser* displays successive values of results, formatting them to be striken out as per the s HTML tag. Example after 4 iterations:

    ~~`asia`~~
    ~~`basia`~~
    ~~`casia`~~
    ~~`dasia`~~

12. Proceed to 10.

Use Case 5: Logging multiple lines at once

1. *User* writes a long running program, mocked up by this pseudo-code:

```
for i in 0..10 {
    result := run long running task
    start := current time
}
```

2. *User* imports package or library in language appropriate manner

```
import logging library
for i in 0.. {
    result := run long running task
    start := current time
}
```

3. *User* configures loggers

```
import logging library
configure_logger("localhost", 3000, "results", TABLE)
for i in 0.. {
    result := run long running task
    start := current time
    duration := concat(current time - start, "s")
}
```

4. *User* writes code to collect results into a single data structure

```
import logging library
configure_logger("localhost", 3000, "results", TABLE)
results := List()
for i in 0.. {
    result := run long running task
    start := current time
    duration := concat(current time - start, "s")
    results.append(["i": i, "result": result,
                    "elapsed": duration])
}
```

5. *User* logs result collection all at once:

```
import logging library
configure_logger("localhost", 3000, "results", TABLE)
```

```
results := List()
for i in 0.. {
    result := run long running task
    start := current time
    duration := concat(current time - start, "s")
    results.append(["i": i, "result": result,
                    "elapsed": duration])
}
log_all("results", results)
```

6. *User* turns on browser pointed at the logging server at address indicating a concrete logger `"localhost:3000/table-printer"` and GUI has **ID "results"** on a scroll-able place with select and delete button.

7. *User* starts program.

8. *Program* runs and sends configuration for "results" to server at `localhost:3000` as JSON.

9. *Server* receives configuration information, performs configuration.

10. *Program* continues running and sends data to "results" containing three columns and four rows.

11. Server receives data, performs visualization.

12. *Browser* prints the data all at once as the appropriate format :

n	**elapsed**	**result**
0	6s	asia
1	42s	basia
2	180s	casia
3	13s	dasia

13. Proceed to 11.

Use Case 6: Logging to multiple sinks

1. *User* writes a long running program, mocked up by this pseudo-code:

```
for i in 0.. {
    result := run long running task
    start := current time
    duration := concat(current time - start, "s")
}
```

2. *User* imports package or library in language appropriate manner

```
import logging library
for i in 0.. {
    start := current time
    result := run long running task
    duration := concat(current time - start, "s")
}
```

3. *User* configures loggers to display data as a table and as a bar plot:

```
import logging library
configure_logger("localhost", 3000, "results", TABLE)
configure_logger("localhost", 3000, "elapsed_time",
                 GRAPH, BAR_PLOT, "n", "seconds")
for i in 0.. {
    start := current time
    result := run long running task
    duration := concat(current time - start, "s")
}
```

4. *User* logs data from running program to two loggers:

```
import logging library
configure_logger("localhost", 3000, "results", SIMPLE_PRINTER)
for i in 0.. {
    start := current time
    result := run long running task
    duration := concat(current time - start, "s")
    log("results", ["i": i, "result": result])
    log("elapsed_time", ["i": i, "elapsed_time": duration])
}
```

5. *User* turns on browser pointed at the logging server at address indicating a concrete logger `"localhost:3000/table-printer"` and `"localhost:3000/graph-printer"`, for table and graph respectively. GUI has **ID "results"** and **ID "elapsed_time"** on a scroll-able place with select and delete button for simple printer and graph.

6. *User* starts program.

7. *Program* runs and sends configuration for `"results"` and `"elapsed_time"` to server at `localhost:3000` as JSON.

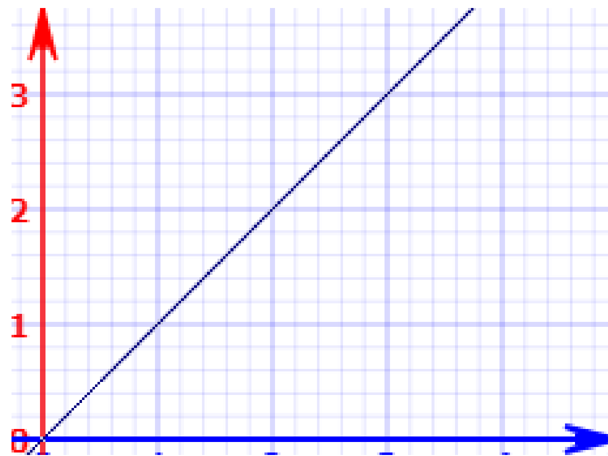
8. *Server* receives configuration information, performs configuration.

9. *Program* running and sends data to `"results"` and `"elapsed_time"` .

10. *Server* receives data, performs visualization.

11. *Browser* updates the table and the graph simultaneously.

n	result
0	asia
1	basia
2	casia
3	dasia



12. Proceed to 10.

Use Case 7: Grid layout

1. *User* writes a long running program, mocked up by this pseudo-code:

```
for i in 0.. {
    start := current time
    result := run long running task
    duration := concat(current time - start, "s")
}
```

2. *User* imports package or library in language appropriate manner

```
import logging library

for i in 0.. {
    start := current time
    result := run long running task
    duration := concat(current time - start, "s")
}
```

3. *User* configures loggers to display data as a table and as a bar plot:

```
import logging library
configure_logger("localhost", 3000, "results", TABLE)
configure_logger("localhost", 3000, "elapsed_time",
                 GRAPH, BAR_PLOT, "n", "seconds")
for i in 0.. {
    start := current time
    result := run long running task
    duration := concat(current time - start, "s")
}
```

4. *User* configures a combined view for multiple loggers, specifying which loggers to include and the number of columns (2) in the grid:

```
import logging library
configure_logger("localhost", 3000, "results", TABLE)
configure_logger("localhost", 3000, "elapsed_time",
                 GRAPH, BAR_PLOT, "n", "seconds")
configure_view("localhost", 3000,   "results_and_elapsed_time",
                 GRID, 2, "results", "elapsed_time")
for i in 0.. {
    start := current time
    result := run long running task
    duration := concat(current time - start, "s")
```

```
    log("results", ["i": i, "result": result])
    log("elapsed_time", ["i": i, "elapsed_time": duration])
}
```

5. *User* turns on browser pointed at the logging server at address indicating both a concrete logger `"localhost:3000/log=results"` and `"localhost:3000/log=elapsed_time"` are updated simultaneously.

6. *User* turns on another browser pointed at the logging server at address indicating a view `"localhost:3000/combined-printer"`. GUI has **ID "results_and_elapsed_time"** on a scroll-able place with select and delete button for simple printer and graph.

7. *User* turns on browser pointed at the logging server at address indicating a concrete logger `"localhost:3000/table-printer"` and `"localhost:3000/graph-printer"`, for table and graph respectively. GUI has **ID "results"** and **ID "elapsed_time"** on a scroll-able place with select and delete button for simple printer and graph.

8. *User* starts program.

9. *Program* runs and sends configuration for `"results"` and `"elapsed_time"` to server at `localhost:3000` as JSON.

10. *Server* receives configuration information, performs configuration.

11. *Program* continues running and sends data to `"results"` as JSON.

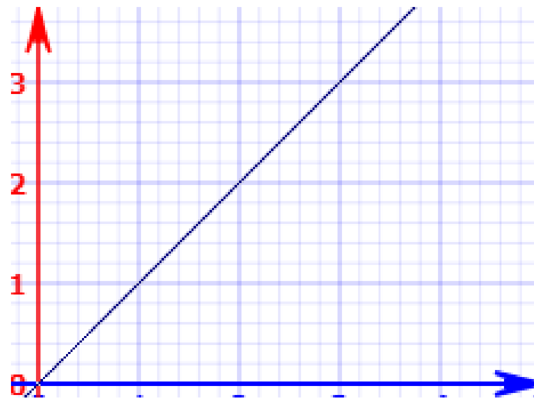
12. *Server* receives data, performs visualization.

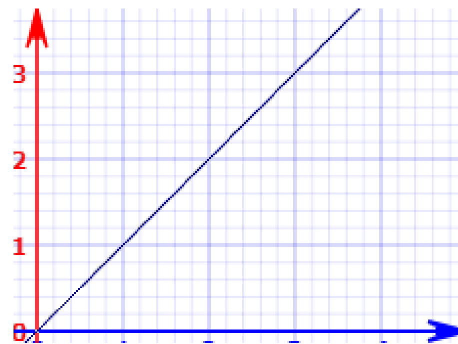
13. *Browser* pointed at "results" updates it's table.

n	**result**
0	asia
1	basia
2	casia
3	dasia

14. *Browser* pointer at "results_and_elapsed_time" is updated simultaneously.

15. *Browser* pointed at "results_and_elapsed_time" updates it's graph.





n	result
0	asia
1	basia
2	casia
3	dasia

16. Proceed to 12.

Use Case 7 Extensions:

Use Case 7b: Grid layout with more sinks than columns

1. *User* writes a long running program, mocked up by this pseudo-code

```
for i in 0.. {
    start := current time
    result := run long running task
    duration := concat(current time - start, "s")
}
```

2. *User* imports package or library in language appropriate manner

```
import logging library
for i in 0.. {
    start := current time
    result := run long running task
    duration := concat(current time - start, "s")
}
```

3. *User* configures loggers to display data as a table

```
import logging library
configure_logger("localhost", 3000, "results", TABLE)
configure_logger("localhost", 3000, "elapsed_time",
                 GRAPH, BAR_PLOT, "n", "seconds")
for i in 0..3 {
    start := current time
    result := run long running task
    duration := concat(current time - start, "s")
}
```

4. *User* configures a combined view for multiple loggers, specifying which loggers to include and the number of `columns (1) in the grid`:

```
import logging library
configure_logger("localhost", 3000, "results", TABLE)
configure_logger("localhost", 3000, "elapsed_time",
                 GRAPH, BAR_PLOT, "n", "seconds")
configure_view("localhost", 3000,
                "results_and_elapsed_time", GRID, 1,
                "elapsed_time", "results")
for i in 0.. {
    start := current time
```

```
    result := run long running task
    duration := concat(current time - start, "s")
}
```

5. *User* logs data from running program to ”results” and ”elapsed time”

```
import logging library
configure_logger("localhost", 3000, "results", TABLE)
configure_logger("localhost", 3000, "elapsed_time",
                 GRAPH, BAR_PLOT, "n", "seconds")
configure_view("localhost", 3000,
"results_and_elapsed_time", GRID, 2, "elapsed_time", "results")
for i in 0.. {
    start := current time
    result := run long running task
    duration := concat(current time - start, "s")
    log("results", ["i": i, "result": result])
    log("elapsed_time", ["i": i, "elapsed_time": duration])
}
```

6. *User* turns on browser pointed at the logging server at address indicating a concrete logger `"localhost:3000/table-printer"`

   and `"localhost:3000/graph-printer"`, for table and graph respectively. GUI has **ID ”results”** and **ID ”elapsed_time”** on a scroll-able place with select and delete button for simple printer and graph.

7. *User* turns on browser pointed at the logging server at address indicating a concrete logger `"localhost:3000/simple-printer"`. GUI has **ID ”results”** on a scroll-able place with select and delete button for simple printer.

8. *User* turns on browser pointed at the logging server at address indicating a concrete logger `"localhost:3000/graph-printer"`. GUI has **ID ”elapsed_time”** on a scroll-able place with select and delete button for simple printer.

9. *User* turns on another browser pointed at the logging server at address indicating a view `"localhost:3000/combined-printer"`. GUI has `ID "results_and_elapsed_time"` on a scroll-able place with select and delete button for simple printer and graph.

10. User starts program.

11. *Program* runs and sends configuration for ”results” and `"elapsed_time"` to server at `localhost:3000` as JSON.

12. *Server* receives configuration information, performs configuration.

13. *Program* continues running and sends data to data to `"results"` and `"elapsed_time"`.

14. Server receives data, performs visualization.

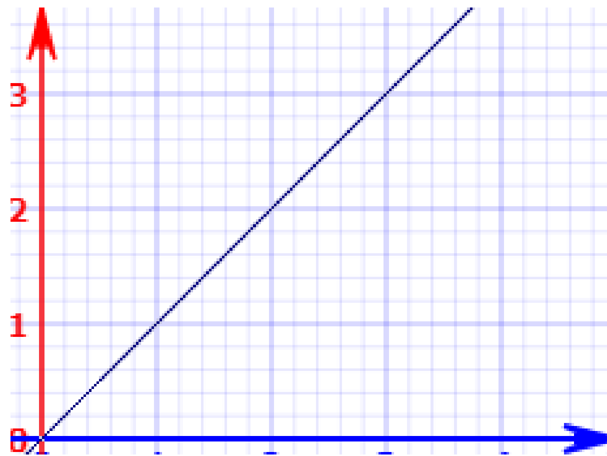
15. The browser pointed at ”results” updates it’s table. Example after 4 iterations:

n	**result**
0	asia
1	basia
2	casia
3	dasia

16. *Program* continues running and sends data to `"elapsed_time"`.

17. *Server* receives data, performs visualization.

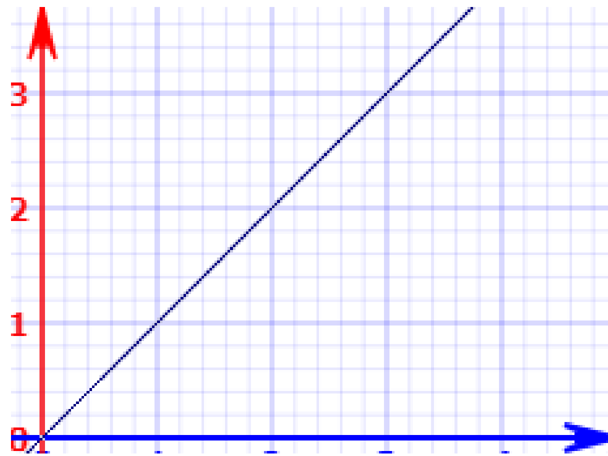
18. *Browser* pointed at `"elapsed_time"` updates it’s graph. Example after 4 iterations:



19. The browser pointed at ”results_and_elapsed_time” updates it’s graph. Example after 4 iterations (mind order):

n	**result**
0	asia
1	basia
2	casia
3	dasia

15. Proceed to 13.



Use Case 8: Custom logger

1. *User* writes a long running program, mocked up by this pseudo-code:

```
for i in 0.. {
    start := current time
    result := run long running task
}
```

2. *User* imports package or library in language appropriate manner

```
import logging library
for i in 0.. {
    start := current time
    result := run long running task
}
```

3. *User* configures loggers to display data using a custom logger named `"my_logger"`.

```
import logging library
configure_logger("localhost", 3000, "results",
                 CUSTOM, "my_logger")
for i in 0.. {
    start := current time
    result := run long running task
}
```

4. *User* writes a module in JS/TS implementing a specific API.

```
import logging library
configure_logger("localhost", 3000, "results", TABLE)
for i in 0.. {
    result := run long running task
    log("results", ["i": i, "result": result])
}
```

5. *User* turns on browser pointed at the logging server at address indicating a concrete logger `"localhost:3000/custom-printer"` and GUI has **ID "results"** on a scroll-able place with select and delete button.

6. *User* starts program.

7. *Program* runs and sends configuration for `"results"` to server at `localhost:3000` as JSON. At this point `"my_logger"` is fully installed and operational.

8. *Server* receives configuration information, performs configuration. If "my_logger" cannot be used, the call to the server returns an exception and halts the program.

9. *Program* continues running and sends data to `"results"`.

10. *Server* receives data, performs visualization using function provided by `"my_logger"` module.

11. The browser pointed at "results" updates it's visualization. Example after 4 iterations:

```
hello: ["n": 0, "result": "asia" ]
hello: ["n": 1, "result": "basia"]
hello: ["n": 2, "result": "casia"]
hello: ["n": 3, "result": "dasia"]
```

12. Proceed to 10.

Use Case 8 Extensions:

Use Case 8b: Custom view composer

1. *User* writes a long running program, mocked up by this pseudo-code

```
for i in 0.. {
    start := current time
    result := run long running task
    duration := concat(current time - start, "s")
}
```

2. *User* imports package or library in language appropriate manner

```
for i in 0.. {
    start := current time
    result := run long running task
    duration := concat(current time - start, "s")
}
```

3. *User* configures loggers to to display data as a table and as a bar plot:

```
import logging library
configure_logger("localhost", 3000, "results", TABLE)
configure_logger("localhost", 3000, "elapsed_time",
                 GRAPH, BAR_PLOT, "n", "seconds")
for i in 0.. {
    start := current time
    result := run long running task
    duration := concat(current time - start, "s")
}
```

4. *User* configures a combined view for multiple loggers using a custom view composer called "`my_view`" :

```
import logging library
configure_logger("localhost", 3000, "results", TABLE)
configure_logger("localhost", 3000, "elapsed_time",
                 GRAPH, BAR_PLOT, "n", "seconds")
configure_logger("localhost", 3000,
                 "results_and_elapsed_time",
                 CUSTOM, "results", "elapsed_time")
for i in 0..3 {
    start := current time
    result := run long running task
```

```
    duration := concat(current time - start, "s")
}
```

4a. Potentially the custom view composer can have configuration options:

```
configure_view("localhost", 3000,  ...,
               "results_and_elapsed_time", CUSTOM,
               "my_view", "results", "elapsed_time")
```

5. *User* logs data from running program to both "`results`" and "`elapsed_time`".

```
import logging library
configure_logger("localhost", 3000, "results", TABLE)
configure_logger("localhost", 3000, "elapsed_time",
                 GRAPH, BAR_PLOT, "n", "seconds")
configure_view("localhost", 3000,  "results_and_elapsed_time",
               CUSTOM, "my_view", "results", "elapsed_time")
for i in 0..3 {
    start := current time
    result := run long running task
    duration := concat(current time - start, "s")
    log("results", ["i": i, "result": result])
    log("elapsed_time", ["i": i, "elapsed": duration])
}
```

6. *User* writes a module (eg. Node module, git repo) in JS/TS implementing a specific API. The user can expand the data interface to suit the custom data. The rest is already configured.

```
export interface ICustomPrinterPayload {
    sessionId: string; //unique id for every session
   data: any[]; // each object must have unique key id: string
}
```

7. User turns on browser pointed at the logging server at address indicating a concrete logger "`localhost:3000/custom-printer`" and GUI has **ID "results_and_elapsed_time"** on a scroll-able place with select and delete button.

8. *User* starts program.

9. *Program* runs and sends configuration for "`results_and_elapsed_time`" to server at "`localhost:3000`" as JSON. At this point "`my_view`" is fully installed and operational.

9a. If "my_view" cannot be used, the call to the server returns an exception and halts the program.

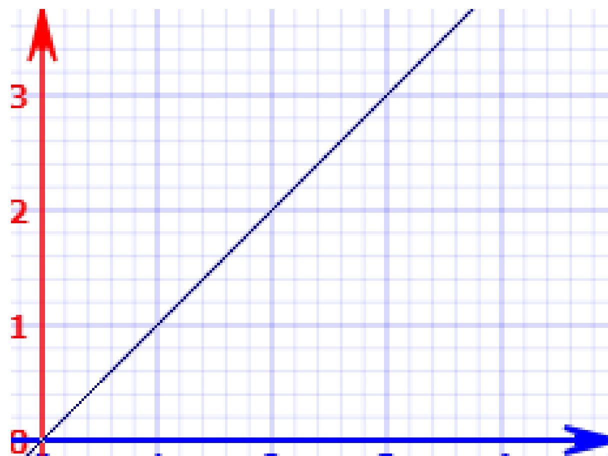
10. *Server* receives configuration information, performs configuration.

11. *Program* continues running and sends data to "results" and "elapsed_time".

12. Server receives data, performs visualization using "results" and "elapsed_time" loggers and composes the results using the by "my_view" module.

13. The browser pointed at "results_and_elapsed_time" updates it's visualization. Example after 4 iterations (mind order):

n	result
0	asia
1	basia
2	casia
3	dasia



15. Proceed to 11.

2.2 Non-Functional requirements analysis

This section describes the system's operating capabilities and constraints. Types of non-functional requirements are scalability capacity, availability, reliability, recover-ability, data integrity, etc. We consider all non-functional requirements before we implement or design any software i.e. in the early stages of the development process. We go through them in detail to help us choose the best tech stack to reach the goals we set out in the thesis statement.

NFR1: Portability/Bindings *Native bindings* Logging is invoked by the programmer in the native language. While language agnostic logging is technically possible, invoking such loggers is expected to be cumbersome, require boilerplate code, and require third-party bindings. A logging web service can get data from Restful API, but boilerplate code would be needed to compose HTTP requests and, in most programming languages, a third-party binding would have to send them to the service. The thin-client should have minimum layering and should be robust. While the scope of this work we will implement two language specific bindings as a proof of concept and provide a tutorial for future implementers.

NFR2: Quick binding development Logging is secondary to the business logic of the programmer's application. Thus, it should not get in the way of the business logic and the code that implements it. Therefore, using the logger should not require more than adding a single line to the code, and should not require modifications to existing code, including specifically passing around a state object to the sites where the logger is called.

NFR3: UTF-8 support To support developers and users from all countries around the world, and support multiple languages to give better context. The logger should support UTF-8, it is a Unicode Standard and extends US ASCII[20].

NFR4: Callable anywhere Logging can be done anywhere in the code regardless of context: the programmer can call a function that logs something anywhere in their code. Specifically, the programmer should not have to pass additional logger objects to the sites where logging occurs. Otherwise the programmer would be faced with the burden of modifying the structure of their own code to make logging possible which runs against the general requirements.

NFR5: Agile The logging system needs to be easy to install and configure. The bindings should be native and the data logging system should pro-

[20]https://en.wikipedia.org/wiki/UTF-8

vide setup to run locally or remotely. If the logging system is not agile it may only have appeal to a very small segment.

Existing methods of data logging are very limited in terms of what can be expressed through it. We propose that modern loggers should use the browser as output, which is both more powerful in what it can render, and less alien than command-line to an unaccustomed user. We discuss the design further in the next chapter 3.

Chapter **3**

Design and implementation

In this chapter we describe a proof-of-concept system that serves as a solution to the problem defined in the introduction. The chapter consists of two main sections. In chapter 3 we set out a general overview and an architecture of the system. In section 3.2 we provide concrete technical details of the server-side and client-side components of the system.

3.1 Design

Since one of the base assumptions of this thesis is that a logger system can leverage the browser to provide adequate data visualization facilities, our design is built around that assumption. In order to provide dynamic data to a browser, the system must be equipped with the ability to serve web content. Since the system is meant to provide native bindings to multiple languages, it is best to provide a client-server architecture, where the server coordinates the communication between the client process and the browser. Since the logger is meant to provide easily written bindings for a growing number of languages, it must ensure that the portion of its architecture written in native language is small as well. Therefore we make the decision to concentrate the functionality of the system in the server and make the client thin. The client-server-browser three-component design brings about many other advantages as the server can serve data to multiple browsers and receive logging information from multiple clients at once. In addition one client can potentially communicate with multiple loggers at once. We present the details of the architecture below in Fig. 3.1 and describe each component in detail below.

Client Processes A client process is the program written and executed by the user which contains calls to the thin client API. The API allows configuration and sending data to be logged. Multiple client processes can use the same sink or multiple sinks to send data to the server for visualization via the *thin client*.

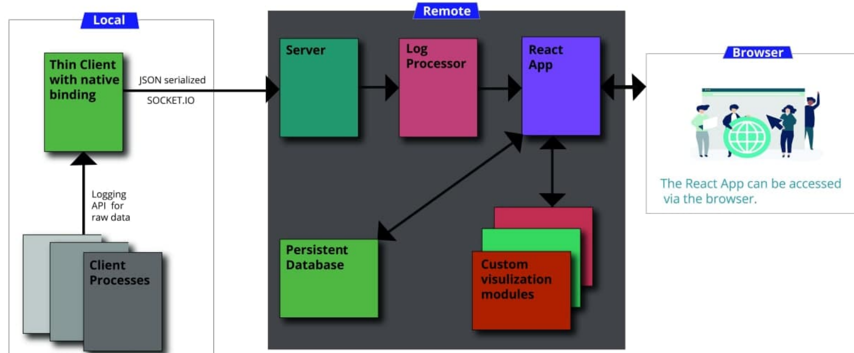


Figure 3.1: Architecture of down-spiral

Thin Client with native binding The thin-client is responsible for accepting raw data from the client processes, and serializing them to JSON, and sending it to the *server*. The thin client has bindings in the native language which makes it a hassle free experience integrating it in the code unlike third-party libraries which require extra steps. The thin-client receives configuration from client processes and then connect to that socket. The thin-client then emits data to a specified event when the client process sends data logs.

Server The server is responsible for accepting requests from the thin client, decoding them, and orchestrating processing by referring to the logger. It is also responsible for logging events in *persistent storage*. The communication with the server is real-time, event-based, and asynchronous (fire and forget). The communication with the thin client is done through Socket.IO, through an asynchronous event based network protocol.

The protocol has only one type of message. It always originates from the client and is received by the server (semantically the server is listen-only). The message announces a logging event to the server. The message is a JSON structure which specifies the ID of the logger, the type of the logger (eg. simple, table, graph), the payload of the event in the appropriate format for its type.

Log Processor The log processor is responsible for accepting request from the server and binding the data of type any to the specified logger type function in the react-app. The communication between the log processor and the server is asynchronous and event based. The communication with server is done via socket-io-client, protocol has logger type and payload in JSON. Log processor catches the emitted event (via the server), processes it accordingly.

React App The react-app is responsible for saving data to the database and also rendering it in the browser. The component on which the function has been called, dispatches a call to Redux-Saga. It simultaneously updates the component state in the persistent database and the Redux app sate. Redux updates the component, React uses the Virtual DOM to render an HTML tree virtually first, and then, every time a state changes and we get a new HTML tree that needs to be taken to the browser's DOM, instead of writing the whole new tree React will only write the difference between the new tree and the previous tree.

Persistent Database The persistent database collects log events: it stores messages between sessions and provides recall for application state. The database reflects the state of the application and every time some changes are made Redux-Saga fires a call and updates the database in parallel with the application. The database is upgraded if needed via an upgrade saga, database instances is closed, and instance is removed from the store when the app requests for the database to close (typically when the app itself is closed).

Browser The browser is used to access the react-app, and see visualizations of different types of data logs. Web based GUI is used for easy navigation. The changes in the browser are updated by the react-app using, a virtual DOM system.

Custom Visualization Modules **down-spiral** supports custom view and custom visualization user can set the variables in mentioned in the guide and also also have custom HTML views. User needs to set structure of the payload sent in the component in the react-app and it will either display as key-value pair or any custom view if set by the user.

3.2 Implementation

We present *Down-spiral*, an complete implementation of our data logger system. Down-spiral follows the design from the previous section and consists of two parts: a client-side binding library and a remote part. Down-spiral also communicates with a browser. We provide two native language thin clients: one in Java and one in Python. We use a React App written in TypeScript and running on a Node.js server to implement the remote components. We describe the technology selection in more detail below.

3.2.1 Technology Selection

It is becoming ever more difficult to clarify the right technology alternatives because the number of technologies is increasing and technologies are becoming very complex. Technology selection describes the process of making a

choice between a number of alternatives. If technology selection is done properly, the selected technologies should be able to move through the complete development process and lead to solutions of identified problems.

Node.js[21] Node.js is an open-source, cross-platform, back-end JavaScript run-time environment that runs on the Chrome V8 engine and executes JavaScript code outside a web browser. We use Node.js to implement our server component. Nodejs has asynchronous processing which makes it a good candidate for real-time data transfer as a back-end server and also has native support for JSON. Asynchronous processing allows requests to be processed without blocking (non-blocking I/O) the thread. So after a request is processed, it can push out a callback and continue serving requests. That helps Node.js make the most of single threading, resulting in short response time and concurrent processing. Node.js, being an open-source project, encourages support and contribution aimed at the improvement and adoption of the platform. Due to JavaScript environment there is there is seamless JSON support. It also has great repository of libraries within its package manager: NPM.[22] The choice has given a limitation that we cannot use web-sockets or other protocols for transfer of data, a work around it is using a JavaScript library with for event based communication, we use Socket.IO in our case. Although we have this limitation we also have an advantage introduced, we can use the React App to handle DOM manipulation and to communicate with the browser. The server uses Node.js for listening to requests.

Socket.IO [16] is a framework for real-time event-based client-server communication. We use it to implement the communication layer between the local and remote components of the system and for internal communication between the server and the log processor. We use Socket.IO module for the thin-client, the server, and also the log processor to emit payload for event based communication.

The server is listening and catches data from the thin client event, and sends it to get processed to Log Processor API according to predefined event outcome. The thin client is in the native language that serializes data logs to JSON and then, sends the JSON to the server using Socket.IO module, emits the data to an event. Socket.IO is used both in the thin-client and the server. It supports implementation various languages like C++, Java, and Python, to name a few. The consequences include support for thin-client available.

ReactJS[23] is a highly used open-source JavaScript Library. It helps in creating impressive web apps that require minimal effort and coding. We

[21] https://nodejs.org/
[22] https://www.npmjs.com/
[23] https://reactjs.org/

use ReactJs for front-end view rendered in the browser. We choose React App to render data on the browser, which ReactJS does using virtual DOM manipulation. DOM (document object model) is a logical structure of documents in HTML, XHTML, or XML formats. Web browsers are using layout engines to transform or parse the representation HTML-syntax into a document object model, which we can see in browsers. Uni-directional flow code in Reactjs ensures stable code. It allows for direct work with components and uses downward data binding to ensure that changes in child structures don't affect their parents. ReactJS was the first open-source project by Facebook, which ensure that it uses all advantages of free access – a lot of useful applications and additional tools from off-company developers. We use TypeScript instead of JavaScript while creating the react-app. TypeScript is a statically typed super-set of JavaScript. React introduced hooks for component state management which leads us to choose Redux which simplifies storing and managing component states in large applications with many dynamic elements where it becomes increasingly difficult.

TypeScript[24] extends JavaScript by adding types to the language. We use TypeScript as an alternative to JavaScript for the react-app. TypeScript is a programming language developed and maintained by Microsoft. TypeScript is designed for the development of large applications and transpiles to JavaScript. TypeScript supports concepts from class-based object-oriented programming like classes, interfaces, inheritance, and more. Researchers found that TypeScript detects 15% of common bugs at the compile stage [17] . Although it requires trans-piling into JavaScript and cannot be directly run by the browser, it can be configured easily using the configuration file. TypeScript is well documented and popular among the users.

Redux[25] is a library which provides a data store, and React-Redux provides the glue between React and Redux. We used them to store and manage the state of the react-app. It stores application state in a single object and allows every component to access application state without dealing with child components or using any callbacks. Redux also preserves the functionality of uni-directional flow code similar to React. Redux is extensible via middle-ware only has no out-of-the-box solution for dealing with side-effects.

This leaves us with two popular middle-wares used with Redux, Redux-Thunk and Redux-Saga. We choose Redux-Saga for our case.

[24]https://en.wikipedia.org/wiki/TypeScript
[25]https://redux.js.org/
[26]https://redux-saga.js.org/

Redux-Saga[26] is a middle-ware library used to allow a Redux store to interact with resources outside of itself asynchronously. This includes making HTTP requests to external services, accessing browser storage, and executing I/O operations. These operations are also known as side effects.

Redux Saga helps to organize these side effects in a way that is easier to manage [18]. The main benefit that Redux-Saga has in comparison to Redux-Thunk is there no callback hell which means we can avoid passing functions and calling them, additionally call and put methods return JavaScript objects. This makes it also easier to test asynchronous data flow [19]. Redux-Saga also is used to fetch current state from database and also for updating the database with current state. Redux-Saga is used as middle-ware to fire requests to the database and Redux asynchronously.

SQLite[27] is a simple databse management system which we use for persistent data base—log events and changes in the react-app are saved and updated into an SQLite database. SQLite is famous for its great feature of zero-configuration, which means no complex setup or administration is needed. SQLite has a file based system that makes it easily portable. It is free and easy implementation. It requires no extra configuration or space because it a server-less installation and also done in just a few minutes. SQLite is fast and usually there are no problems with the speed of data retrieval or data itself.

The selected technology allows us to achieve the goals set out in the thesis statement.

3.2.2 Implementation details

We provide a complete overview of the implementation of the thin client and remote system components of Down-spiral. We describe the structure of the thin client as a class diagram and the remote server as a component diagram and provide descriptions of the role of each component and class in brief in the text. We also describe the interactions among various components of the system with sequence diagrams.

3.2.2.1 Thin client

The thin client is responsible for providing native language bindings and serializing logged data into JSON using Socket.IO library. It also sends the JSON payload to the server. The structure of the thin client is presented in the class diagram in Fig. 3.2. It describes the Java implementation specifically. The Python bindings are similar.

[27] https://www.sqlite.org/index.html

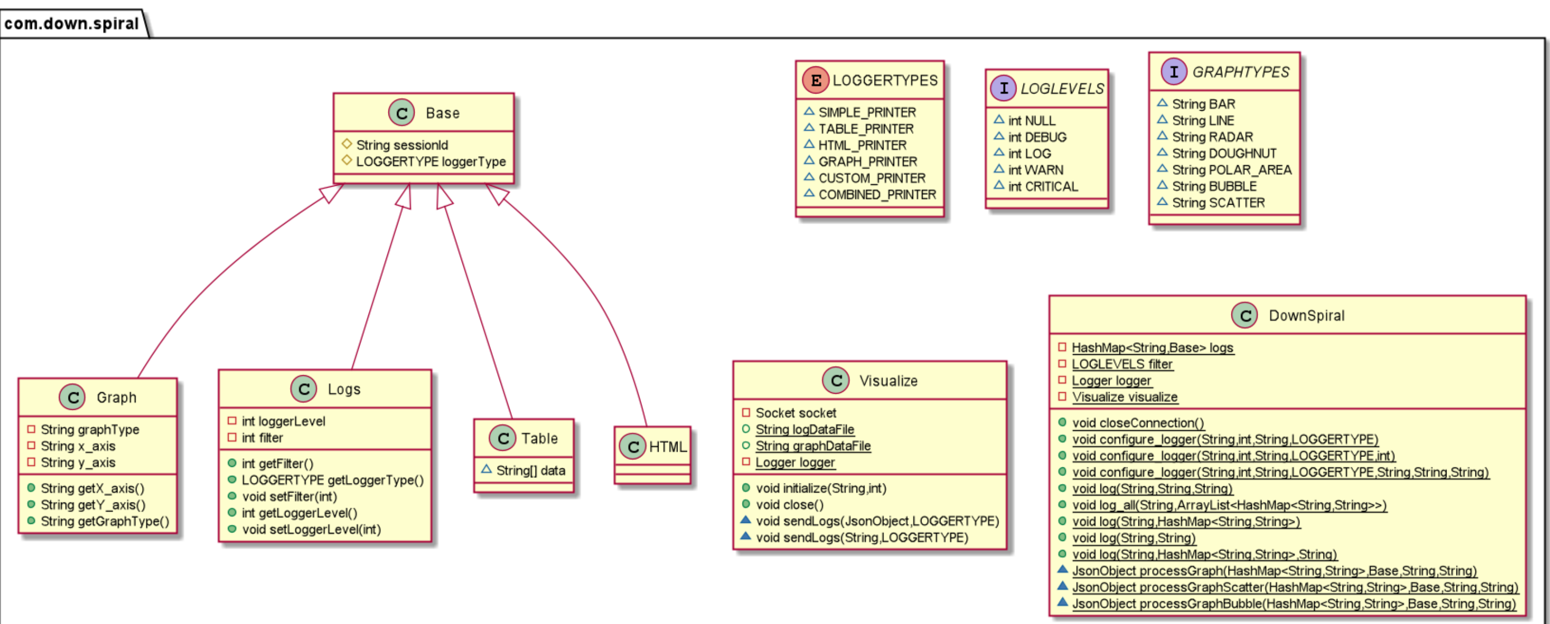


Figure 3.2: Class Diagram of the Thin Client.

The main class of the thin client is `DownSpiral`. It acts as a facade and exposes functions for configuring loggers and logging. It is also responsible for serialization to JSON. `DownSpiral` methods send the serialized JSON payload to the server using functions from the `Visualize` class. `Visualize` uses Socket.IO to create and close connections, and also to emit events.

The loggers can be configured in terms of the type of their output and minimum logging levels. The thin client has different logger types (which are represented by `LOGGERTYPES`) and log levels (represented by the interface `LOGLEVELS`). Graph output is additionally configured by specifying the type of graphs (represented by `GRAPHTYPES`). The specific logger is instantiated as subclasses of the `Base` superclass. The `Base` class is used to store the session ID and logger type and opens the connection using `Visualize` class. Its subclasses define output-specific behaviors: the `Logs` class have different filters to monitor different log levels and `Graph` class has functions to get the x-axis and y-axis values and also a getter for a graph type.

The interaction between the Client Process, thin client, and the Server is presented in Fig. 3.3.

3.2.2.2 React app

The react-app is responsible for setting the state of the application and rendering it the browser. It also saves and updates the current state of the application in the database. We present the broad structure and behavior of the react app as a component diagram in Fig. 3.4. The main component of the remote-system for browser and database interaction is the react-app (represented by `React App` in the figure). It renders the current state store in the Redux store (represented by `Redux Store`) and also triggers crud functionality in the database.

Redux is a state container for JavaScript apps. The React app calls a function after receiving an event from a Socket.IO client (represented by `Socket.IO API`). This action is caught in a Redux saga (represented by `Redux Saga`), it fires two events: it updates the Redux state and the database (represented by `DB`) with the new changes. The redux components is responsible, for setting the state of application in the react-app. The redux-saga, middle-ware triggers events in database and the redux store state. The role of the database is to manage and create data persistence. The react-app also updates it's state from the database.

The components responsible for parsing log messages are the set of printer components: graph printer, custom printer, simple printer, HTML printer, table printer, and combined printer. They all accept JSON messages through Socket.IO and parse them into an appropriate representation, to which the payload is then set. This then triggers changes in the view and the payload gets rendered in the browser. We show the specific JSON formats for each of these components in Tab. 3.1.

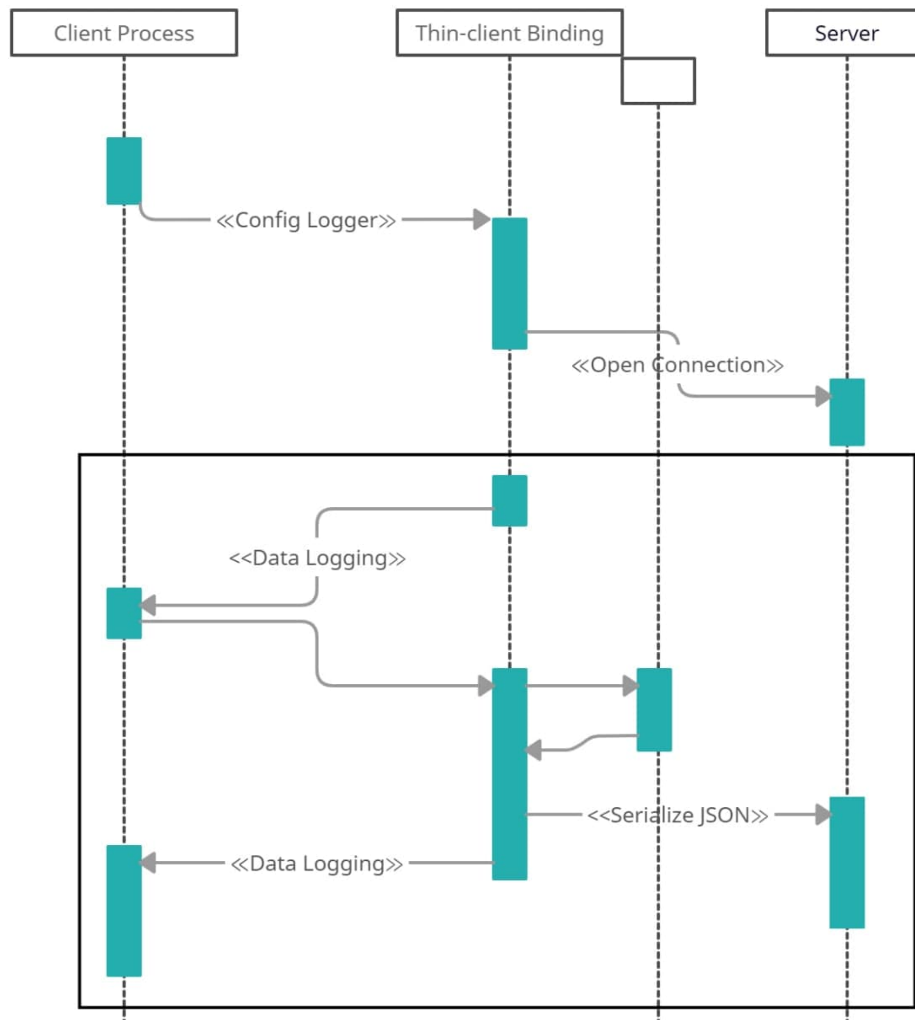


Figure 3.3: UML Sequence Diagram: sending data to the server.

The interaction between the Server, Log Processor, React, Redux-Saga and the Database is presented in Fig. 3.5. `Server` sends the payload as JSON to the `Logger`, it receives the payload and binds it with the `React App`. React app triggers middle-ware `Redux Saga`. Middle-ware appends the payload sent by the server in redux-store and Database. Redux store updates the react-app then makes changes in the components according to the payload.

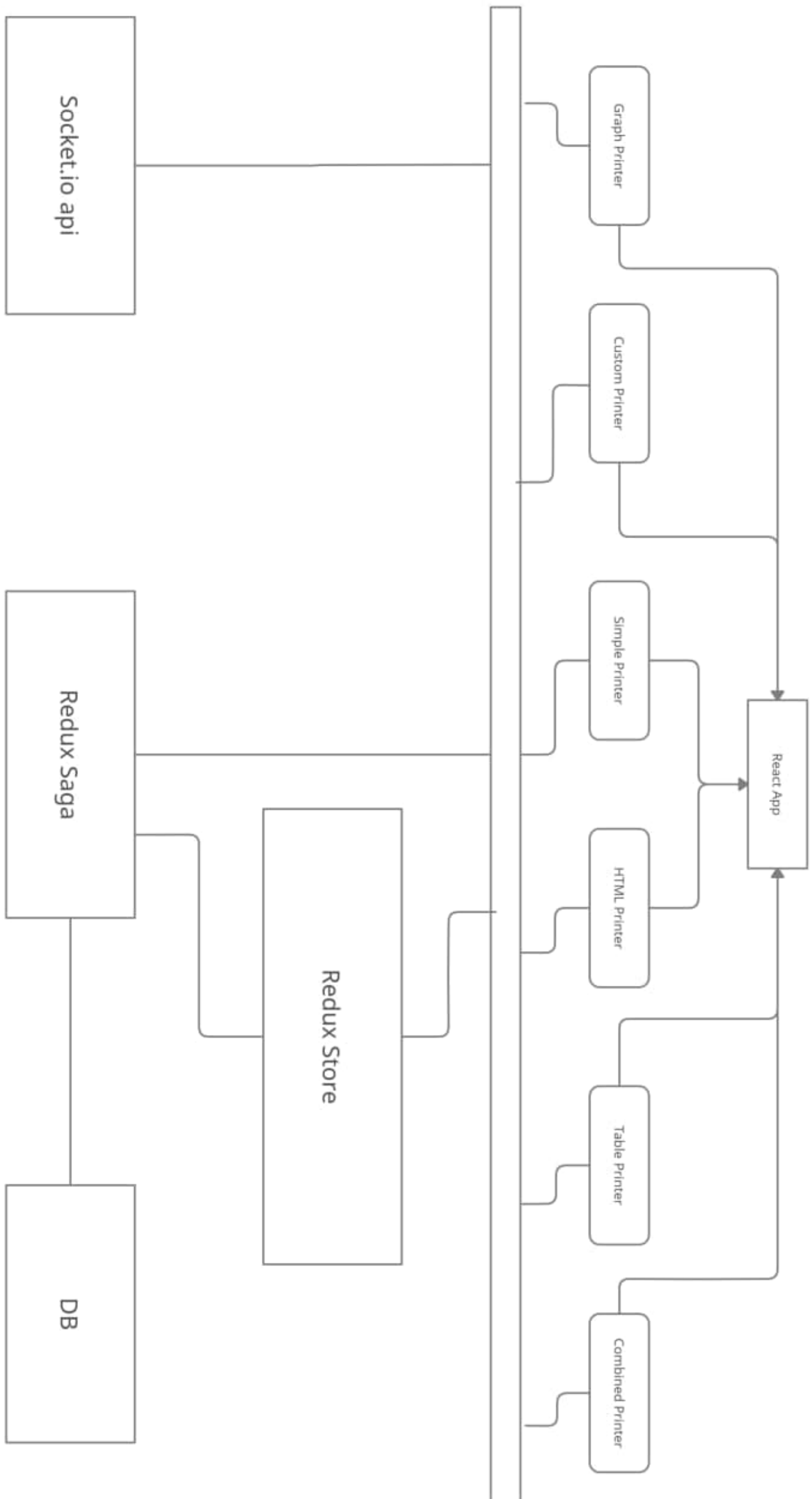


Figure 3.4: Component Diagram of the react-app.

Component	Socket.IO Event	JSON format
Table Printer	TABLE_PRINTER	{ sessionId: string, header: string[], data: any[] }
HTML Printer	HTML_PRINTER	{ sessionId: string, header: string[], data: any[] }
Graph Printer	GRAPH_PRINTER	{ sessionId: string, type: string, labels: string[], datasets: { label:string, data: any[] } }
Simple Printer	SIMPLE_PRINTER	{ sessionId: string, data: { id: string, resultLable:string resultValue:string }[] }
Combined Printer	COMBINED_PRINTER	{ sessionId: string, grid: number, combinedViewsPayload: { type: string, sessionId: string }[] }
Custom Printer	CUSTOM_PRINTER	{ sessionId: string, grid: number, combinedViewsPayload: { type: string, sessionId: string }[] }

Table 3.1: Component message format.

3.3 Lessons learned

During the implementation of the thesis project the author had to navigate around specific engineering problems. Initially, I selected Play framework [20], for the remote system. As it is built on Akka [21], and is an open source web-application framework. Since Play framework has cross-platform and native in Scala and Java, it was ideal candidate but after implementing a few use case scenarios, it became clear it had some problems and limitations that wouldn't meet the requirements. Problems encountered: plugins are not stable, reloading library code doesn't work, and application startup time grows with application size. This lead us to choose ReactJS for the web-application part. It has well maintained libraries because it is updated by Facebook, and

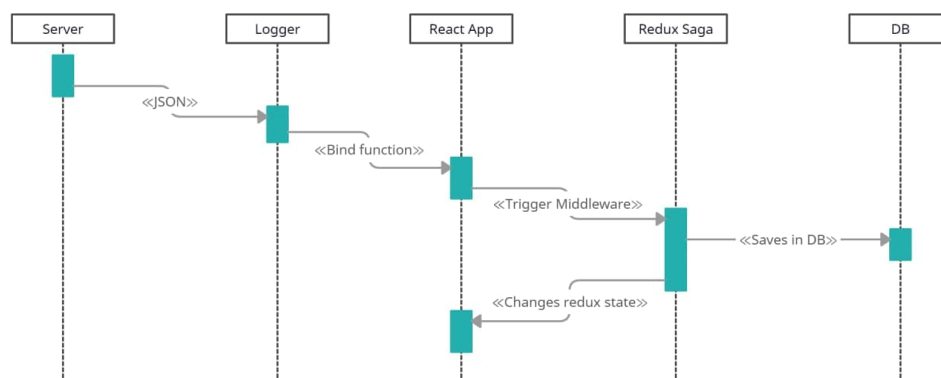


Figure 3.5: UML Sequence Diagram of updating data in the database and the application

has backward compatible which is another feature Play framework does not support. ReactJs is cross platform and has JavaScript which makes it ideal to deal with JSON. We justify our choice in section 3.2.1 .

Interesting bugs found and fixed One of the interesting bugs found was when setting data in react-app in the `Table component`. Since most tables use the procedure of setting the header before the table is formed, but we wanted it to be done simultaneously which makes it dynamic and easy to map new columns and data to it. When the mapping was done dynamically the react-app was firing a warning in the console for white-spaces as column value. Due to dynamic mapping, all empty element cells had a white space. This was fixed using grids for dynamic mapping of Table that allows real-time mapping and handles exceptions.

The bug was that the data wasn't set in the redux-state for the react-app. Since the logging data is passed to the browser via the state, it was not possible to render the data in the browser. The Node.js server forwards the request to react-app and then the react-app sets it but this could lead to problems as the requests are asynchronous, so we used socket-io client to use solve this problem. As socket-io-client is asynchronous and can catch events from the server, and can also call the function in the react-app, which then updates the view in the browser using the virtual DOM, and also saves the data in the database.

Chapter **4**

Evaluation

4.1 Correctness

We verify the correctness of the solution and look for all requirements we have fulfilled both functional and non-functional, also list discrepancies if any. If any case has not been covered or fulfilled we list out reasons and explain how extension of work can be carried out.

We fulfill all the requirements but there is still a lot for improvement in the solution. We can minimize the packaging size in react-app to make the renders more smooth and seamless. The thin-client can also be made to configure a batch of custom modules and with parallel implementation also in the react-app. The section 5.2 expands on the scope of how to expand on it.

4.2 Tests

Latency is the time between a logger receives a message and the message is rendered in the browser. latency is crucial for logging systems, both for those that are suppose to run in real time, and loggers in general, as loggers are not supposed to slow down the programs they are logging for. In this section we characterize Down-spiral's latency.

NFR	Name	State
NFR1	Portability/Bindings	Fulfilled
NFR2	Quick binding development	Fulfilled
NFR3	UTF-8 support	Fulfilled
NFR4	Callable anywhere	Fulfilled
NFR5	Agile	Fulfilled

Table 4.1: Non-functional requirements

UC	Name	State
1	Simple logging	Fulfilled
1b	Turning it on later	Fulfilled
1c	Log level	Fulfilled
2	Logging to table	Fulfilled
2b	Logging to table with heteROGENEOUS COLUMNS	Fulfilled
3	Logging to graph	Fulfilled
4	Logging to custom HTML	Fulfilled
5	Logging multiple lines at ONCE	Fulfilled
6	Logging multiple sinks	Fulfilled
7	Grid layout	Fulfilled
7b	Grid layout with more sinks than columns	Fulfilled
8	Custom logger	Fulfilled
8b	Custom view composer	Fulfilled

Table 4.2: Use cases

The benchmarks were performed using the following testing environment: Intel 6700HQ processor 2.60 GHz, 20 GB RAM DDR4 2133MHz, NVIDIA 965M, 128GB SSD M.2 EVO 960 SAMSUNG, 64-bit OS, and running on windows 10 Professional. All components(thin client, server and react app) were running in parallel on this system configuration. The client and server were both running on a single machine and communicated via loop-back. Running the server locally allowed us to compare system time recorded on both components.

We group the tests into two parts which are short messages (∼10 characters) and long messages (∼2000 characters) texts are generated automatically and are random. We tests for simple printer, table printer, graph printer, and HTML printer. We send 100 messages in sequence to each printer and measure the latency separately for each message. We test the lag time between the transmission of each payload from the client process until the time of rendering in the browser. We record the time in the client process when the payload is sent to the thin client, and also when the react-app sets and updates the component in the virtual DOM, i.e. render in the browser. Time was logged using `console.log` in the react-app and `System.out` in Java.

4.2.1 Results

We collect the aggregated observations in Tab. 4.3 showing the minimum, maximum and mean time in each benchmark group, as well as the standard deviation within the group. All results are in milliseconds. We also plot the distribution of the results for each group using violin graphs in Figures 4.1–4.7.

Benchmark	Workload	Latency [ms]			
		min	max	mean	stddev
Simple printer	short messages	1	155	5.3	35
Simple printer	long messages	2	1155	118.75	115.09
Table printer	short messages	4	259	6.67	25.50
Table printer	long messages	3	1355	166.3	176.3
Graph printer	short messages	3	250	6.57	24.60
HTML printer	short messages	1	166	5	62.8
HTML printer	long messages	2	1255	16.59	125.09

Table 4.3: Latency evaluation results.

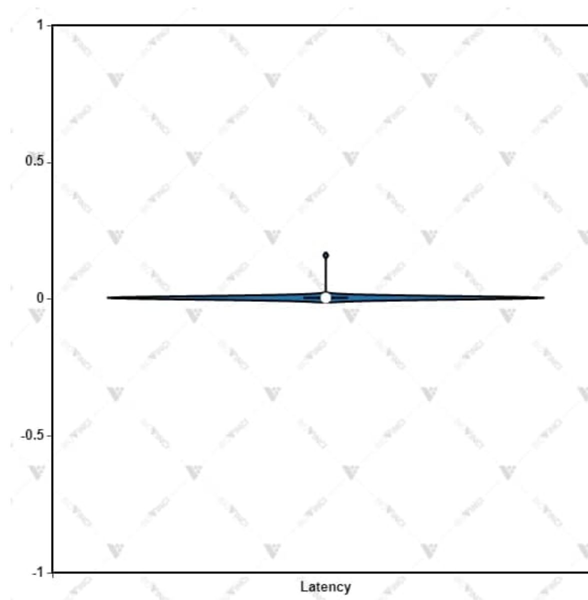


Figure 4.1: Simple Printer short messages latency

4.2.2 Discussion

Overall, the results of the evaluation show that the logger has relatively low latency, between 5 and 166ms on average, depending on workload and printer. The results show that the maximum latency can exceed one second for long messages in the case of the simple printer and the table printer. We additionally see in figures 4.1–4.7 that these are outliers with respect to the general population. The analysis of the results in detail shows that the outlier is always the first message sent, and the analysis of the system shows that this delay stems from the setup that the logger undergoes when it receives its first message.

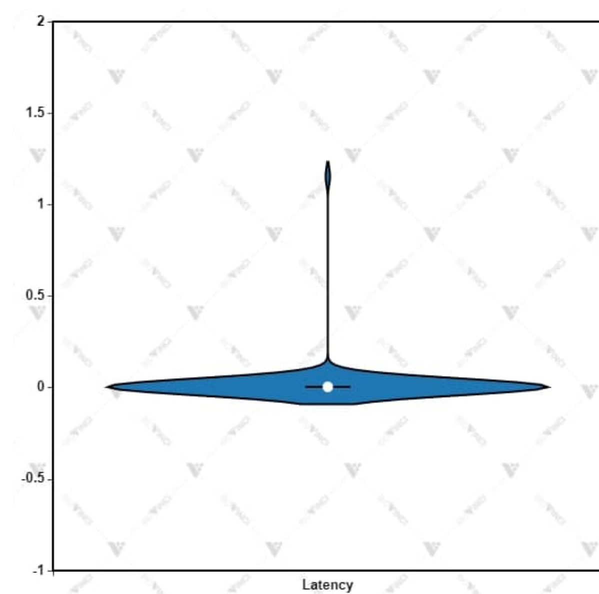


Figure 4.2: Simple Printer long messages latency

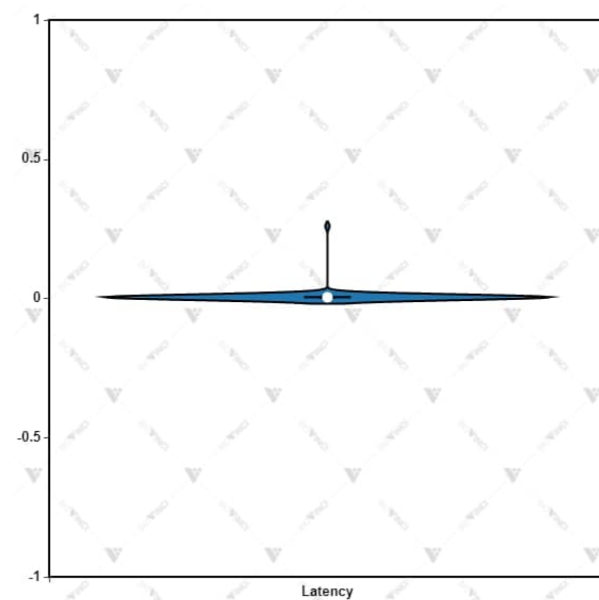


Figure 4.3: Table Printer short messages latency

In general we see that loggers perform better, both in terms of absolute values and in terms of latency distribution when the payload is small. If the payload is 200× larger, mean latency increases between three- and thirty-fold. We also see from the violin plots and the standard distribution, that larger

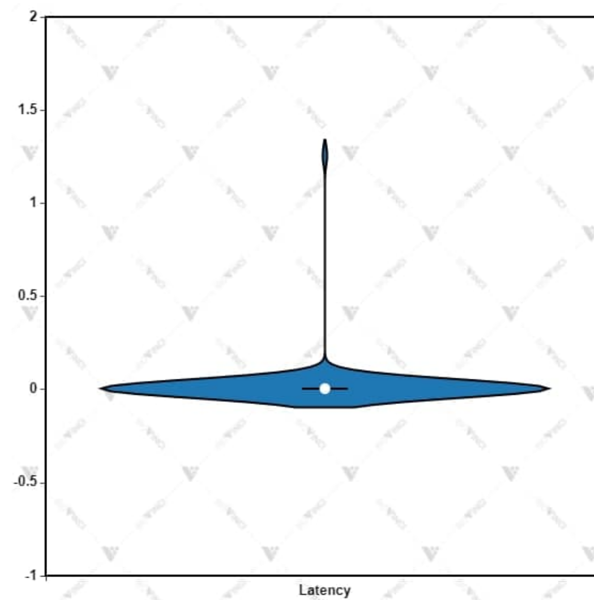


Figure 4.4: Table Printer long messages latency

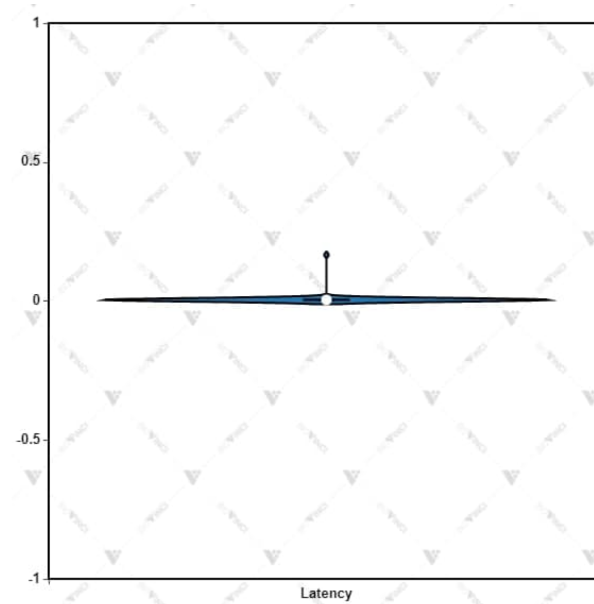


Figure 4.5: HTML Printer short messages latency

messages scatter the distribution of the results. However, in absolute terms, these differences are not larger and unnoticeable for a human operator.

The speed results demonstrate the logging lag is marginal in milliseconds

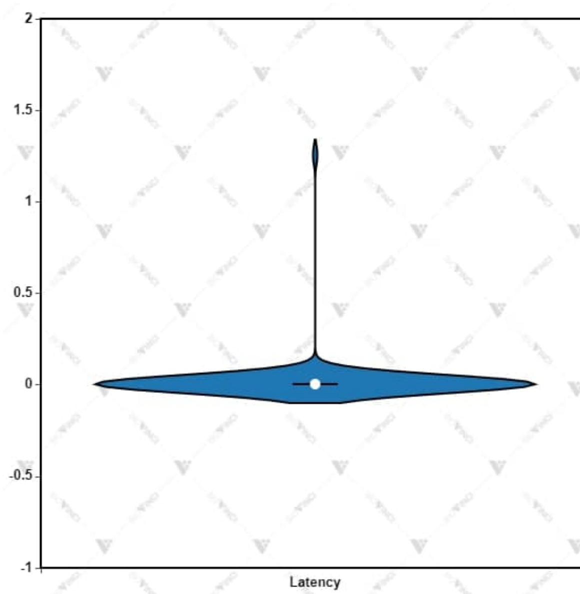


Figure 4.6: HTML Printer long messages latency

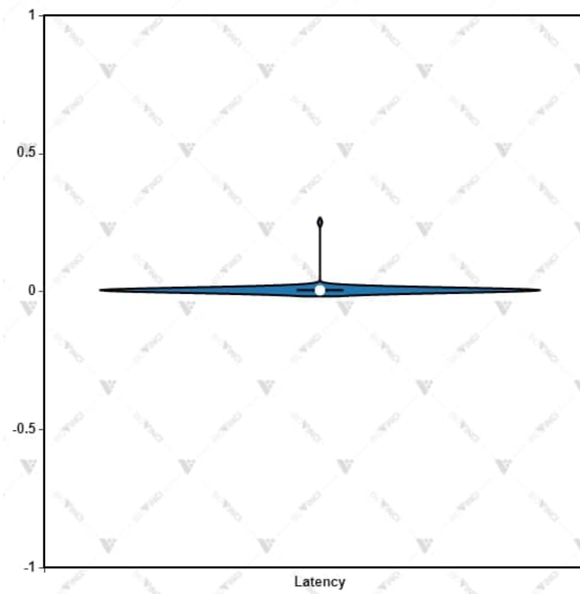


Figure 4.7: Graph Printer short messages latency

and doesn't affect the rendering or the order of the payload and therefore the Down-spiral is fit for purpose.

Chapter 5

Conclusion

As discussed in the introduction, a well designed logging system is a must have for programmers. Use of logging makes the process efficient and ensure no redundancy. Significant data loggers can output or store to a data persistence option but the format is mostly rudimentary is just a string, although it can metadata attached to it. Visualization is limited in data loggers, and most of them either require extensive setup or further external support to configured. Since, logging is secondary to the user is doing in the code it makes visualization of logs a tedious task. Real-time logging makes it easy for the user to track changes and all that we reviewed loggers lack this feature. Custom configuration is difficult and almost impossible in major data loggers.

We set goals to make a portable logger that has out-of-box support for visualization. The logger were also to support multiple input sources and multiple output sinks, so the user can log different types of data in real-time and from any process. Logging can be made more user friendly using web technologies and rendering the output in the browser. Also, to provide the user with customization options that require no intervention in the main source code.

Down-spiral has out-of-box support for visualization, using a react-app that communicates with a browser in real-time to monitor and update the state of the application and its sub-components. Using Socket.IO we made the data logging procedure asynchronous and real-time. We also used a database for persistence of the application and reflecting its current state, which preserves the functionality of loggers like Log4J and SLF4J. The code is open-source and available publicly at GitHub with documentation such as installation guide and manual, and also a thin-client binding guide. The users can either use thin-client already available to them or create a new one. The data logs can now easily be visualized using this data logger, which will make debugging and tracking changes in real-time easy to track and record. No going through boring command-line interface to find what you need, switch to the web browser.

5.1 Impact

The introduction of Down-spiral gives the user the power of visualization of data logs in the browser. This gives leeway for more exploration in the field of data logging visualizers and data logging itself. It also reflects upon how important data logging is to solve real world problems. The user now has an option available with a portable logger, to visualize data. That needs no configuration aside from importing the library in the language. Down-spiral has built-in support for different types visualizations like tables, graphs, custom HTML and more. Since, down-spiral is real-time it makes it easier to track changes through logs and also to view multiple large data-sets.

5.2 Future work

Although Down-spiral fulfills all the non-functional requirements and Use-Cases, there is some extension to the work that can be done.

Generate Bindings The features that were out of scope generating bindings for thin-clients automatically for other languages. The bindings for thin-clients has to be in native the language. It also requires a support library available from libraries[28] in Socket.IO otherwise a similar alternative.

Customization Module Extension The custom module feature available on the remote-system, can be extended and made more dynamic. It shouldn't allow `DangerouslySetInnerHTML` practice, which is to manipulate the DOM manually in a react-app. If the domain is manipulated manually it bypasses the virtual DOM which makes ReactJS code stable.

[28] https://socket.io/docs/v3/index.html

Bibliography

[1] Cloudflare. Understanding network error logging. 2020, [Online; retrieved January 31, 2021]. Available from: https://support.cloudflare.com/hc/en-us/articles/360050691831-Understanding-Network-Error-Logging

[2] Yang, L.; Phipps, D. Controlling collection of debugging data. July 7 2009, US Patent 7,559,055.

[3] Morris, T. H.; Thornton, Z.; et al. Industrial control system simulation and data logging for intrusion detection system research. *7th annual southeastern cyber security summit*, 2015: pp. 3–4.

[4] Bosch, N.; Bosch, J. Software logs for machine learning in a DevOps environment. In *SEAA'20: the 46th Euromicro Conference on Software Engineering and Advanced Applications*, Aug 2020, pp. 29–33, doi: 10.1109/SEAA51224.2020.00016.

[5] Averbuch, A. J.; Brauer, R. L.; et al. Facility space data logging device. Oct. 26 1993, US Patent 5,256,908.

[6] Apache Software Foundation. Elastic Stack for data logging and visualization. 2001, [Online; retrieved January 31, 2021]. Available from: https://logging.apache.org/log4j/2.x/

[7] Apache Software Foundation. What is Apache Log4net. [Online; retrieved January 31, 2021]. Available from: https://logging.apache.org/log4net/

[8] Allman, E. Syslog is a standard for sending and receiving notification messages–in a particular format–from various network devices. 2001, [Online; retrieved January 31, 2021]. Available from: https://en.wikipedia.org/wiki/Syslog

[9] Gülcü, C. The simple logging facade for Java: SLF4J. 2021, [Online; retrieved January 30, 2021]. Available from: http://www.slf4j.org/index.html

[10] Humphries, C.; Prigent, N.; et al. Elvis: Extensible log visualization. In *Proceedings of the Tenth Workshop on Visualization for Cyber Security*, 2013, pp. 9–16.

[11] B.V., E. Elastic Stack for data logging and visualization. 2012, [Online; retrieved January 31, 2021]. Available from: https://www.elastic.co/

[12] Martinez, F.; Toth, P. K. Web-services-based data logging system including multiple data logging service types. Sept. 25 2007, US Patent 7,275,104.

[13] Burnett, S.; Chen, L.; et al. Network Error Logging: Client-side measurement of end-to-end web service reliability. In *NSDI'20: the 17th USENIX Symposium on Networked Systems Design and Implementation*, 2020, pp. 985–998.

[14] Duane, N. Porting Log4J2 to .NET. Mail Archives for the Apache Log4J User Group. [Online; retrieved February 13, 2021]. Available from: https://tinyurl.com/y9hfbq5z

[15] Guerzoni, L.; Cuda, D. Speech processor data logging helps in predicting early linguistic outcomes in implanted children. *International Journal of Pediatric Otorhinolaryngology*, volume 101, 2017: pp. 81 – 86, ISSN 0165-5876, doi:https://doi.org/10.1016/j.ijporl.2017.07.026. Available from: http://www.sciencedirect.com/science/article/pii/S0165587617303397

[16] Automatic Inc. Socket.IO. 2015, [Online; retrieved February 5, 2021]. Available from: https://socket.io/

[17] Alexsoft. The good and the bad of TypeScript. 2020, [Online; retrieved February 5, 2021]. Available from: https://www.altexsoft.com/blog/typescript-pros-and-cons/

[18] Chim, N. A high level introduction to Redux Saga. 2020, [Online; retrieved February 5, 2021]. Available from: https://www.loginradius.com/blog/async/introduction-to-redux-saga/

[19] Rosenfield, S. Redux-Thunk vs. Redux-Saga. 2018, [Online; retrieved February 5, 2021]. Available from: https://medium.com/@shoshanarosenfield/redux-thunk-vs-redux-saga-93fe82878b2d

[20] Lightbend. Play: The high velocity web framework for Java and Scala. 2007, [Online; retrieved February 12, 2021]. Available from: https://www.playframework.com/

[21] Lightbend. Akka. 2020, [Online; retrieved February 12, 2021]. Available from: https://doc.akka.io/

Appendix A

Acronyms

GUI Graphical user interface

XML Extensible markup language

DOM Document Object Model

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

JSON JavaScript Object Notation

JDBC Java Database Connectivity

SQL Structured Query Language

I/O Input and Output

IE Internet Explorer

CSV Comma-Separated Values

XML Extensible Markup Language

Appendix B

Contents of enclosed CD

```
readme.md ........................ the file with CD contents description
src ...................................... the directory of source codes
├── down-sprial ............................. implementation sources
└── thin-clients ............. both thin-client implementation sources
    ├── thin-client-java ......................... java implementation
    └── thin-client-python ................... python implementation
thesis-master ............................. the thesis text directory
├── thesis.pdf ......................... the thesis text in PDF format
├── thesis.ps ........................... the thesis text in PS format
└── latex-source-code ................... the thesis latex source code
```

Appendix C

Installation and user guide

C.1 Installation Guide

Perquisites

- NPM package manager
- NPM package serve installed globally 'npm install -g serve'

Installation

- Clone the repo and change to the working directory.

```
cd */down-spiral/frontend
```

- Install all dependencies

```
npm install
```

- Start React App

```
npm serve-build
```

- Run the Node.js server

```
npm run server
```

Usage

Types of loggers

1. Table Printer Table printer is a logger that logs the data as HTML Table.

    - Accepts a json payload of type:

      ```
      {SessionId: string, header: string[], data: any[]}
      ```

    - Socket.io event for Table Printer:

      ```
      TABLE_PRINTER
      ```

    - Complete Usage

      ```
          <code>.emit('TABLE_PRINTER', {
       SessionId: string,
       header: string[],
       data: any[]`
       }
      )</code>
      ```

2. HTML Printer HTML printer is a logger that logs the data as custom HTML.

    Usage

    - Accepts a json payload of type:

      ```
      {SessionId: string, htmlPayload: string}
      ```

    - Socket.io event for HTML Printer:

      ```
      HTML_PRINTER
      ```

    - Complete Usage

      ```
      .emit('HTML_PRINTER', {SessionId: string, htmlPayload: string})
      ```

3. Graph Printer GRAPH printer is a logger that logs the data as Graph.

    `Supported Graph formats`

    a) BAR

b) SCATTER
c) RADAR
d) LINE
e) DOUGHNUT
f) POLAR-AREA
g) BUBBLE

Usage

- Accepts a json payload of type:

```
        {
SessionId: string,
type:string,labels: string[],
datasets:{
label:string,
data: any[]
 }
}
```

- Socket.io event for Table Printer:

```
GRAPH_PRINTER
```

- Complete Usage

```
  .emit('GRAPH_PRINTER', {
 SessionId: string,
 type:string,
 labels: string[],
 datasets:{
  label:string,
  data: any[]
  }
 }
)
```

4. Simple Printer SIMPLE printer is a logger that logs the data as Key value pairs.

Usage

- Accepts a json payload of type:

```
{
SessionId: string,
data: {id: string, resultLable:string, resultValue:string} []
}
```

- Socket.io event for HTML Printer:

```
SIMPLE_PRINTER
```

- Complete Usage

```
.emit('SIMPLE_PRINTER', {
SessionId: string,
data: {id: string, resultLable:string, resultValue:string} []
}
```

5. Combined Printer COMBINED printer is a logger that logs the data from other chosen printers.

   Usage

   - Accepts a json payload of type:

```
{
sessionId: string;
grid: number;
combinedViewsPayload: { type: string; sessionId: string }[];
}
Here type can be any other logger type, passed with respective sessionId.
```

   - Socket.io event for HTML Printer:

```
COMBINED_PRINTER
```

   - Complete Usage

```
.emit('COMBINED_PRINTER',{
sessionId: string;
grid: number;
combinedViewsPayload: { type: string; sessionId: string }[];
}
)
```

6. Custom Printer CUSTOM printer is a logger that's defined by the user.In CUSTOM_PRINTER user defines the json object and it's mapping with JSX in the file CustomPrinter.JSX

   Usage

   - Define your custom json in the interface

     ```
     ICustomPrinterPayload { sessionId: string, data: any[] }
     ```

     expand data object to your need, must have a unique id.
   - Map your objects to JSX Component

     ```
     Note: Redux state mapping is already configured to the interface
     ```

   - Socket.io event for HTML Printer:

     ```
     CUSTOM_PRINTER
     ```

   - Complete Usage
     - Define your payload for the JSON object here
     - This object will be the expected payload by reducer
     - Unique sessionId is required for every new session
     - expand data property of the interface to define your JSON object
     - example:- data: id: string; date: Date; name: string;[] above example is defining data to be an array of id: string; date: Date; name: string;

   ```
   .emit('CUSTOM_PRINTER',{
   sessionId: string;
   data: any[];
   }
   )
   ```