

University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Barłomiej Kozaryna
418338

Daniel Filimonow
417855

Andrzej Stalke
418461

Grzegorz Zaleski
418494

WebSocket based application with CQL TypeScript Driver running on C++ Seastar server for ScyllaDB

Bachelor's thesis
in COMPUTER SCIENCE

Supervisor:
mgr Michał Moźdzzonek

Warsaw, June 2022

Abstract

The purpose of this thesis is to describe the creation of an open-source Scylla database driver written in TypeScript, a server which would allow better communication between Scylla and the driver and also an example web application for accessing the database using the aforementioned. All of the components were developed as a part of this thesis.

Keywords

ScyllaDB, Seastar, C++, TypeScript, CQL, Driver, WebSocket, Databases, Terminal, Asynchronicity, Server

Thesis domain (Socrates-Erasmus subject area codes)

11.0 Mathematics, Informatics

Subject classification

Information systems - Data management systems - Middleware for databases - Database web servers

Tytuł pracy w języku polskim

Aplikacja bazująca na protokole WebSocket z TypeScriptowym sterownikiem uruchomiana na serwerze C++ Seastar dla ScyllaDB

Contents

Introduction	5
1. Terminology	7
2. Used technologies	9
2.1. Work Environment	9
2.2. React	10
2.3. TypeScript	10
2.4. C++	10
2.5. Seastar	11
2.6. Scylla	11
3. WebSocket Server	13
3.1. WebSocket Protocol	13
3.1.1. WebSocket communication process	13
3.1.2. WebSocket frame	14
3.2. Existing alternatives	15
3.3. Server Model	15
3.4. Implementation	15
3.4.1. Server	16
3.4.2. Connection	17
3.4.3. WebSocket Parser	18
3.5. WebSocket Secure	19
3.6. Integration with Scylla	19
4. TypeScript CQL Driver	21
4.1. Existing alternatives	21
4.2. Design	21
4.2.1. Frame Creator	21
4.2.2. Handshake Sender	22
4.2.3. Query Sender	22
4.2.4. Paging Handler	22
4.2.5. Prepare Sender	22
4.2.6. Execute Sender	23
4.2.7. Consistency Changer	23
4.2.8. Type Converter	23
4.2.9. Authentication Handler	23
4.2.10. Response Handler	23
4.2.11. Error Handler	23

4.3.	Implementation	24
4.3.1.	Technology	24
4.3.2.	TypeScript	24
4.3.3.	Development	24
4.3.4.	Challenges	26
5.	In-browser Scylla terminal	27
5.1.	Existing alternatives	27
5.2.	Designing project	27
5.2.1.	Composition	27
5.2.2.	Connection & authentication pop-up form	27
5.2.3.	Command history	28
5.2.4.	Input section	28
5.2.5.	Response displayer	28
5.3.	Implementation	28
5.3.1.	Technology	28
5.3.2.	Functional React with TypeScript	28
5.3.3.	useState module (State Hook)	29
5.3.4.	useEffect module (Effect Hook)	29
5.3.5.	useMemo module (Memory Hook)	29
5.3.6.	Material UI	29
5.3.7.	Components division	29
5.3.8.	Iterative development	30
5.3.9.	Challenges	31
5.3.10.	Visual appearance	31
6.	Division of work	37
	Bibliography	39
A.	The structure of the source code	41
A.1.	WebSocket Server	41
A.1.1.	Seastar	41
A.1.2.	ScyllaDB	41
A.2.	TypeScript CQL Driver	42
A.3.	In-browser Scylla terminal	42

Introduction

According to the World Economic Forum, at the beginning of 2020, the number of bytes in the digital universe was 40 times bigger than the number of stars in the observable universe. Moreover, it is expected that by 2025 human *data creation speed* will surpass 10^{18} bytes of data daily^[1]. With that in mind, one comes to realisation how great a part of our lives has data management become. To process large amount of information, people need fast, scalable and memory efficient solutions. Without them Discord, for instance, would not be able to transfer nearly 1 billion messages that users send every day (which is approximately 11 000 messages per second) and Netflix users could not watch their daily dose of *314 days* of video^[2].

First attempts to somehow store the digital data were navigational databases: Integrated Data Store and Information Management System, created in the 60's of the previous century. Later on, we could observe the raise of relational databases, with creation of the SQL, and object databases, then NoSQL databases, arriving with today's, trying to keep up with exploding demand, distributed or ML-driven databases.

This brings us to the September of 2015, when a small startup, Cloudbius Systems (now ScyllaDB), released an open source database, which was a rewritten implementation of already existing solutions. Their product became a huge success, significantly outperforming its archetype^[3]. Nowadays, their product is used by such *big players* as i.a. CERN, Discord, IBM, Intel or Samsung SDS^[4].

Our team was tasked by ScyllaDB with designing and implementing a driver for accessing aforementioned database. The main goal of the product is to enable a user to connect with Scylla using a specialised internet protocol, which allows to produce multiple apps which are going to deliver superior user experience. It requires from us adding content throughout many layers of the existing system. We have also created a minimalistic web application which enables users to access the database through their internet browsers.

The structure of this paper is as follows: firstly, we will introduce necessary terminology, explain certain concepts and introduce used technologies; secondly, we will describe the components we created and we will discuss our implementation; lastly, a brief insight in our workflow will be given alongside with some extra remarks.

Chapter 1

Terminology

Static typing

A property of programming languages to know types, contrary to the dynamic programming, at compile time. It enables the compiler to check if all types match each other to minimize the number of possible errors.

Framework

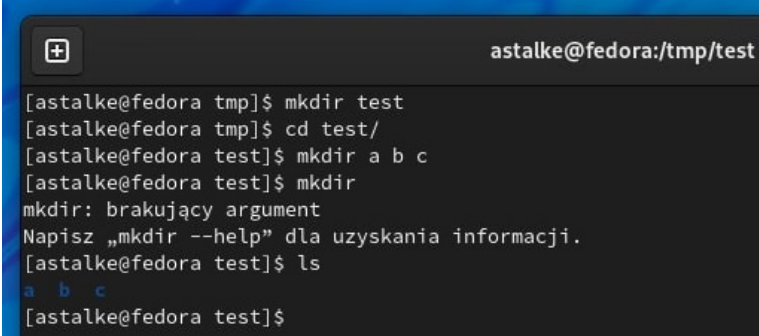
Framework in programming is a tool that provides ready-made components or solutions that are customized in order to speed up development. In most cases framework will be external module or library imported to project's source-code. In our project we used Material User Interface framework instead of pure CSS (Cascading Style Sheets) for front-end development.

Driver

The program responsible for intermediation between the client and the server. In our case it provides an interface to the terminal that enables sending queries to the Scylla database.

Terminal

Terminal is the interface through which user type commands for execution and then (usually) receives the response in the next line as a stream of characters. In the most cases terminal is a minimalist, dark-themed window with command prompt line. In our project we took inspiration from the Linux Terminal to design the terminal of the website from which user can access and use the database. Moreover, the terminal in our project contains more, advanced features in comparison to the regular ones from Linux systems.



```
astalke@fedora:/tmp/test
[astalke@fedora tmp]$ mkdir test
[astalke@fedora tmp]$ cd test/
[astalke@fedora test]$ mkdir a b c
[astalke@fedora test]$ mkdir
mkdir: brakujący argument
Napisz „mkdir --help” dla uzyskania informacji.
[astalke@fedora test]$ ls
a b c
[astalke@fedora test]$
```

Default terminal on Fedora Linux

Nonce

An arbitrary, most often random or pseudo-random, number that can be used only once during communication. Used in authentication processes to protect peers from replay attacks using old connections.

(Communication) Protocol

Set of rules that describe communication between entities. The protocol defines, among others, the syntax, the semantics and ways of synchronisation. Protocols can be implemented and layered.

HTTP

A protocol functioning in the application layer of hypermedia transfer systems. It is the foundation of data transfer in the World Wide Web where hypertext documents are one of the most common types of data. It was introduced in 1991.

TCP

Transmission Control Protocol is a connection-oriented, working on raw octets, protocol functioning in the transport layer. Its main qualities are reliability, error checking and keeping the order of the data sent.

SSL/TLS

Secure Sockets Layer and its successor Transport Layer Security are cybersecurity protocols created to ensure network safety. TLS encrypts the data in a way that in most cases prevents 'man in the middle' attacks. It typically relies on a trusted third-party certificate authority to establish the authenticity of the sides.

CQL

Cassandra Query Language is a language for Cassandra database resembling SQL (Structured Query Language) structure. The syntax is almost the same, it differs by the nature of the targeted products. A relational database provides features that NoSQL does not and the other way around. ScyllaDB is a NoSQL database based on Cassandra. Cassandra Query Language is then the scripting language that is used to communicate with ScyllaDB.

Chapter 2

Used technologies

2.1. Work Environment

Fedora Linux

We were all working on Fedora Linux (version 34 or 35). As this operating system is the one Scylla runs on the best and it was advised from our Scylla supervisor to use this OS. Due to the fact that we all already had different systems on our computers we employed solutions such as Docker, Dual-boot or VirtualBox. It is also important to note that the task to obtain working and stable Fedora OS was far from trivial as each of us needed a different solution to get a working station (because each of us primarily uses different computer with different operating system, varying from MacOS to Manjaro Linux).

Another challenge arose from the fact that Scylla consumes a lot of Random Access Memory (RAM), this forced us to make certain decisions in choosing other tools as that prerequisite rendered some ideas useless. That heavily affected Grzegorz who needed to move from VirtualBox and then dual-boot system to a separate computer with more RAM available.

Fortunately in the end we all were equipped with the working Fedora system and Scylla server.

Visual Studio Code

As our coding workspace we all chose to use Visual Studio Code with selected plugins. The reason for choosing this text editor was simple. Each one of us could:

- Select plugins to make the environment as comfortable as possible.
- Work with different programming languages, switching between C++ and TypeScript was smooth.
- Run programs without the necessity of extra computational power

These (and much more) advantages make VSC the most popular tool for developers all over the world.

WireShark

For testing reasons, we were also using Wireshark which is a very convenient and easy-to-use tool. It captures data packages sent over the network and enables their decoding. It enabled us to track queries and responses between the driver and Scylla.

Internet Browsers

We used Chrome and Firefox - the two most popular browsers - to work with data packages sent and received by server, driver and terminal. The same tool was used to dynamically see the terminal from the perspective of a regular user and to conduct real-life tests on the terminal.

Github

We used github.com for storing and keeping back-up copy of our work, tracking changes in any set of files of our projects, coordinating work amidst team members collaboratively developing different features.

2.2. React

One of the three most popular open-source frontend frameworks. We chose React as a tool for terminal development because we consider it the cleanest among other frameworks and we have a lot of experience with it.

2.3. TypeScript

Multiparadigm programming language created by Microsoft that extends functionalities of JavaScript. It compiles to the previously mentioned language. Furthermore, a program written in JavaScript is also a valid TypeScript program and all JavaScript libraries are compatible with TypeScript code. Moreover, it allows for static typing, therefore providing much more safety to the language which is known as one of the most unsafe languages. Its popularity is growing year to year because of multiple reasons, code written in TypeScript is safer and clearer than the JavaScript one. Moreover, software engineers outside of the project are more likely to understand, debug and add new features to it. Safety, clarity, and compatibility with terminal and various multiple libraries was the reason to choose TypeScript over any other language in implementation of our cql driver, working with binary data is much more convenient when having an extra layer of security and access to one of the biggest open-source code bases. This decision was consulted with and approved by the Principal Software Engineer from ScyllaDB.

2.4. C++

General-purpose programming language offering functional, generic and object-oriented features in addition to tools allowing for low-level memory manipulation. It is mostly used as a compiled language (in case of our project also). C++ is appreciated mainly for its high performance given the amount of facilities for the programmer.

C++20

Latest, stable version of C++ programming language, standardised by The International Organization for Standardization, published in December 2020. It introduces i.a. coroutines, pack-expansions in lambda init-captures, constraints and concepts^[6].

2.5. Seastar

High performance, event-driven, open source C++ framework, designed for creating non-blocking and asynchronous server applications^[15]. Used techniques are, to name a few: threads, shared memory, mapped files.

2.6. Scylla

Scylla is an open source NoSQL database^[5]. It is implemented as a distributed wide-column database, compatible with Apache Cassandra, however achieving significantly higher throughputs^[3]. This database is written using mainly C++20 and the Seastar framework. Not only does it implement Cassandra's protocols, but also the Amazon DynamoDB API. Scylla uses horizontal partitions of data, called shards, on each node, which makes each CPU core manage different subset of data.

Chapter 3

WebSocket Server

3.1. WebSocket Protocol

Although ubiquitous, hyper text transfer protocol (HTTP) is not always the best choice for client-server communication, for its request-response methods and redundant headers. With help then comes a full duplex alternative - the WebSocket Protocol, which offers lower overheads than the HTTP. WebSocket operates on a single TCP connection and can work on the same ports as HTTP (80 and 443). To achieve compatibility, a HTTP upgrade header is send as a WebSocket Handshake. Afterwards a two-way message framing communication follows. This protocol offers also its encrypted version called WebSocket Secure. The WebSocket protocol is standardised by the IETF as RFC 6455^[7].

3.1.1. WebSocket communication process

The protocol can be divided into two parts: handshake and data transfer. The first one is straightforward: client sends a HTTP upgrade frame containing the following untypical headers:

- Host - the hostname to enable reaching an agreement with the server
- Sec-WebSocket-Key - nonce
- Origin - protection against unauthorized cross-origin use of the server
- Sec-WebSocket-Protocol - which subprotocols the client accepts
- Sec-WebSocket-Version

Then the server, if it is willing to communicate, responds with the last HTTP frame in successful communication containing the following untypical headers:

- Sec-WebSocket-Protocol - server has to choose one of the subprotocols suggested by the client or fail
- Sec-WebSocket-Accept - client nonce transformed accordingly

and status line:

```
HTTP/1.1 101 Switching Protocols
```

After a successful handshake the second part begins, which means that frames can be exchanged in full-duplex between the client and the server. The communication ends if one of the sides sends a *CLOSE* frame (in practice also when the TCP connection timeouts). It has to be noted that our implementation requires the client to always choose exactly one subprotocol offered by the server.

3.1.2. WebSocket frame

Aforementioned frames have the following format (taken from the RFC 6455 documentation)^[7]:

```

0                                     1                                     2                                     3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+--+--+--+-----+--+-----+-----+-----+-----+-----+
|F|R|R|R| opcode|M| Payload len |      Extended payload length |
|I|S|S|S|   (4) |A|       (7) |               (16/64)             |
|N|V|V|V|         |S|           |    (if payload len==126/127)    |
| |1|2|3|         |K|           |                                  |
+--+--+--+-----+--+-----+-----+-----+-----+-----+
|          Extended payload length continued, if payload len == 127 |
+ - - - - - + - - - - - +-----+-----+-----+-----+-----+
|                                                              |Masking-key, if MASK set to 1 |
+-----+-----+-----+-----+-----+-----+-----+-----+
| Masking-key (continued)                   |          Payload Data |
+-----+-----+-----+-----+-----+-----+-----+-----+
:                      Payload Data continued ...                :
+ - - - - - + - - - - - +-----+-----+-----+-----+-----+
|                      Payload Data continued ...                |
+-----+-----+-----+-----+-----+-----+-----+-----+

```

Unapparent keywords:

- FIN - indicates whether the frame is a final fragment of a message
- RSV? - extension flags, which are ignored in our server
- opcode - code indicating the type of the frame
- Masking-key - all data sent from the client to the server is masked using this key

Types of frames important for our server:

- 0x1 - UTF-8 text data
- 0x2 - Binary data
- 0x8 - Close frame

The length of a frame depends on the length of payload data. In theory it is possible to send frames up to $2^{63} - 1$ bytes in length.

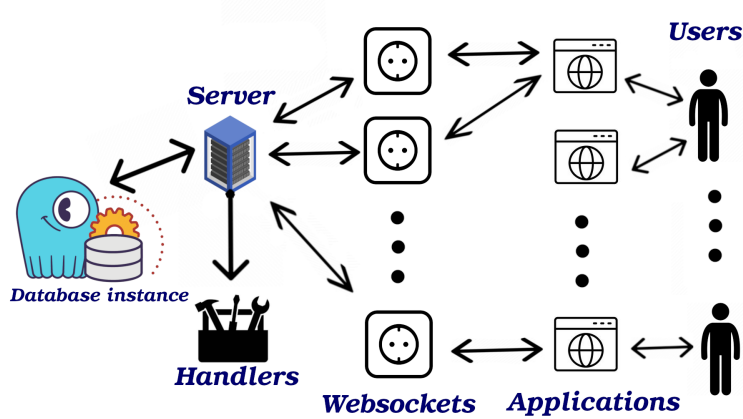
3.2. Existing alternatives

Implementation was preceded by a research from which we tried to learn about already existing, publicly available libraries for communication over WebSocket in different languages, but especially C++.

Among many, Rust (tungstenite)^[8], Python (WebSockets)^[9], C++ (boost/beast ^[10], websocket++^[11]), C (libwebsockets)^[12] have community created solutions which are quite popular. There are also languages like SWI-Prolog^[13] or Haskell^[14] which offer access to official packages.

Even though we used the python's WebSockets library in the debugging process, the aforementioned libraries would be difficult to integrate with Scylla, would unnecessarily increase latency or do not offer features as coroutines. Hence, a decision was made to implement it using the Seastar framework which is already used by Scylla, is reasonably fast and offers all necessary features, i.a. those available in C++20.

3.3. Server Model



Simplified server model

When the server is instantiated it has to have at least one handler registered to know how to perform data transfer part of communication with clients. A handler is a function which accepts input and output streams and returns a `seastar::future<>`. It is recommended for it to hand over the control to the Seastar backend while waiting for data from the input stream. The server keeps handlers inside an ordered map `subprotocol → handler`. Afterwards the server has to be told to listen on a specific socket.

When a client and the server perform a successful handshake an instance of `Connection` class is created, has its subprotocol assigned, and from this moment, asynchronously, manages the data exchange with clients applications through WebSockets.

3.4. Implementation

In this section we will explain the API and structure of classes located in the seastar source files on the path: `include/seastar/websocket/server.hh`. Cases of trivial fields or methods will not be discussed.

Our server implements a subset of the WebSocket protocol sufficient enough to meet Scylla's needs. Nonetheless the software and protocols are 'alive', therefore adequate updates might

be required in the future. Moreover it could be reconsidered to implement an algorithm to pick a subprotocol from a set suggested by the client or start supporting additional frame flags

3.4.1. Server

The class is declared at the beginning of the file to make it visible to other classes which are cyclically dependent.

Public Methods

```
* void listen(socket_address addr)
* void listen(socket_address addr, listen_options lo)
```

Both methods can be used to supply the server with sockets on which it can listen on and the second one specifically allows the user to specify options, which are in the first case a singleton flag - 'reuse_address'. Server can listen on multiple sockets. It stores them in a protected vector '_listeners'. Further management is delegated to the 'do_accepts' method.

```
* future<> stop()
```

This method is used to stop the server. It sets the protected field '_stopped', frees allocated sockets, closes all connections and returns `seastar::future<>` so other actions can be performed by the backend while connections are being closed. The mentioned flag is important for stopping the 'do_accepts' process.

```
* void register_handler(std::string&& name, handler_t handler)
* bool is_handler_registered(std::string const& name)
```

The first method takes a name of the handler (and therefore also a subprotocol) as an rvalue `std::string` and a handler, which is a function taking references to 'input_stream<char>' and 'output_stream<char>' and returning a `seastar::future<>`.

The second one checks whether a handler/subprotocol with given name has already been registered. Handlers are stored inside a private map '_handlers' so both actions are performed in $O(n \log n)$ where n is the number of registered handlers.

Protected Fields

```
* void do_accepts(int which)
* future<> do_accept_one(int which)
```

The main asynchronous loop using seastar is built in method 'do_until'. The future returned by it, is saved to a private field '_accept_fut' so it can be waited on in the stop method. Calling the first method setups the loop which works until '_stopped' flag is set and performs single accepts on the which's socket in '_listeners' asynchronously creating new instances of Connection class which are 'befriended' to our server.

Private Fields

```
* boost::intrusive::list<connection> _connections
```

List which new connections will sign into. This implementation of lists is chosen because of thread safety and to suffer less from cache thrashing,

3.4.2. Connection

```
* boost::intrusive::list_base_hook<>
```

The Connection class must inherit from 'boost::intrusive::list_base_hook' in order to work with 'boost::intrusive::list'.

```
* connection_source_impl
* connection_sink_impl
```

Seastar's input and output streams internally use special classes 'data_source'^[17] and 'data_sink'^[16]. They require classes 'connection_source_impl' and 'connection_sink_impl' to be implemented in order to work with our server.

Public Methods

```
* future<> process()
```

This method is called after accepting a new connection. It calls 'read_loop()' and 'response_loop()' and wait for their completion.

```
* void shutdown()
```

The 'shutdown()' method shuts down the TCP connection with the client.

```
* future<> close()
```

This method is used to gracefully close the connection. It sends a *CLOSE* frame and closes the connection.

Private Methods

```
* future<> close(bool send_close)
```

This method is used to close the connection. If 'send_close' is set to true, then it sends *CLOSE* frame before closing the connection.

```
* future<> handle_ping()
* future<> handle_pong()
```

PING and *PONG* frames are used to check if the other side of the connection is responsive. These methods are responsible for handling them.

Protected Methods

```
* future<> read_loop()
```

This method is responsible for starting the WebSocket connection by calling the 'read_http_upgrade_request()' method, starting the handler and calling 'read_one()' in a loop.

* `future<> read_one()`

This method is used to read data from the client and send it to the WebSocket parser. The received messages are sent to the stream to which the handler is connected.

* `future<> read_http_upgrade_request()`

Before the WebSocket connection can be established, the HTTP upgrade request must be processed. This method is responsible for that and for sending a reply.

* `future<> response_loop()`

This method is responsible for sending data from the `'_output_buffer'` queue to the client.

* `void on_new_connection()`

The server must know that a new connection has been established. This method is used to add the connection to the server's connection array.

* `future<> send_data(opcodes opcode, temporary_buffer<char>&& buff)`

The WebSocket protocol requires data to be wrapped in a frame. This method is used to wrap data stored in the *buff* argument in a frame type specified by the *opcode* argument. The created frame is immediately sent to the client.

3.4.3. WebSocket Parser

This class is made as an object oriented solution to parse incoming frames and store all the necessary data for future non-blocking parsing.

Public Methods

* `future<consumption_result_t> operator()(temporary_buffer<char> data)`

Since TCP packets are delivered as a stream of octets, the 'read' method may not be able to read the entire WebSocket frame at once. To deal with this problem, seastar's input stream^[18] offers 'consume' method that requires *operator()(temporary_buffer<char> data)* to be implemented. It is called to process a new data packet from the client and return whether the WebSocket frame has been processed completely.

* `bool is_valid()`

This method returns *true*, if the parser is still in a valid state and more data can be parsed.

* `opcodes opcode()`

This method returns the type of the frame parsed by the parser.

Private Methods

* `void remove_mask(buff_t& p, size_t n)`

Data in client's WebSocket frame must be masked. This method removes the mask from the data.

Private Fields

* `parsing_state _state`

This field describes the current stage of parsing a frame. There are three states:

- Processing flags and payload data
- Payload length and mask
- Payload

* `connection_state _cstate`

It holds whether the parser is in valid state. There are three states:

- Valid - the parser can accept more data
- Eof - the connection is closed
- Error - an error has occurred, parser is now invalid

* `sstring _buffer`

This field holds the unprocessed data.

* `std::unique_ptr<frame_header> _header`

This field holds the WebSocket header of the processed frame.

* `buff_t _result`

This field holds the payload received in the WebSocket frame.

3.5. WebSocket Secure

In the same way that HTTPS (Hypertext Transfer Protocol Secure) uses SSL/TLS to fix up HTTP vulnerabilities to capturing and changing sent data, WSS (WebSocket Secure) offers an encrypted version of WS. As a bonus part of the project, we added a simple and minimalistic implementation of WSS to our Seastar contribution. It was attained by clever inheritance and usage of solutions already built in the framework.

A secure version of the server has nearly the same interface. The only difference is that during the instantiation it has to create a shared pointer for a certificate, and the listening method has to be given paths to files with a certificate and a corresponding key.

Testing the WSS version of the server was problematic because creating certificates and keys leads to errors about self-signed certificates, which cannot be turned off in certain libraries.

3.6. Integration with Scylla

Integration of the the WebSocket server with ScyllaDB was done by creating a new class 'websocket::controller' that inherits from the class 'protocol_server' and creating an instance of this object in the main function of ScyllaDB. The created object must also be registered as a protocol server. To make it more configurable, the server address and port were added to the configuration file.

Implementation of websocket::controller class

The source code of this class is located in the ScyllaDB source code in the 'websocket' sub-directory. The only important method is *future<> controller::start_server()*. It starts the WebSocket server, registers the *CQL* subprotocol handler and starts listening on configured address and port. The handler is very simple, it works as a bridge between the client and built-in *CQL* server.

Chapter 4

TypeScript CQL Driver

The second and the biggest part of the whole project was a CQL driver that communicated with database using WebSocket protocol. It was implemented using the fourth version of CQL Binary Protocol as it is the newest version that the Scylla database handles. This driver offers a set of functionalities. After providing a human-readable message to certain command it translates the whole message to a list of bytes that are easily understood by the database. Moreover, it decodes all responses from the database to let user know its meaning.

4.1. Existing alternatives

After doing some research we found out that there exists CQL driver for web applications. However it did not use WebSockets and was written for other database also based on Apache Cassandra. Therefore we realized that writing CQL driver over WebSockets on a Scylla database is a great opportunity to create something non-existing and contribute to open source community.

4.2. Design

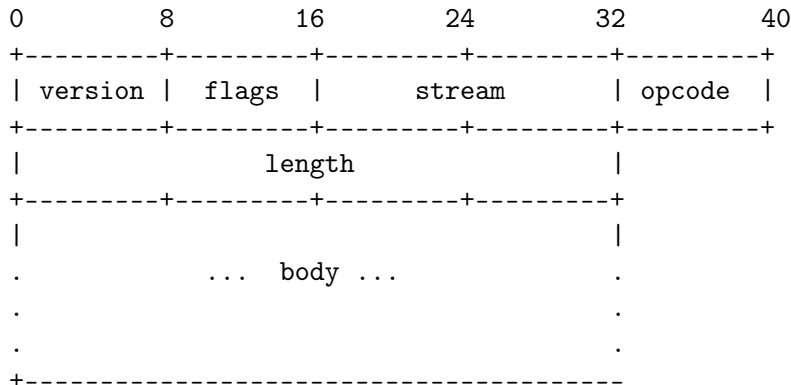
After familiarising with protocol specification the driver was planned to be made of couple semi-independent modules listed as Frame Creator, Query Sender, Paging Handler, Handshake Sender, Execute Sender, Prepare Sender, Consistency Changer, Type Converter, Authentication Handler, Response Handler and Error Handler.

4.2.1. Frame Creator

Most basic yet most important module is Frame creator. Its whole purpose is to create empty frame from scratch. It consists of 6 components:

- Version - Version of the protocol.
- Flags - Number providing extra information about the frame.
- Stream - Number of the message, it is needed to permit asynchronous usage.
- Opcode - Number indicating type of message.
- Length - Body length.
- Body - Message content.

Preview of the frame (numbers indicate number of bits each part has):



4.2.2. Handshake Sender

This module creates and sends handshake message. It starts whole conversation with the database. Without sending it, database would respond with errors indicating that this message was not sent and ask for delivering a handshake.

4.2.3. Query Sender

Query Sender is a very important module, without it the CQL Driver could not access any data from the database. It would make whole driver utterly useless. Query Sender takes an input from the user and at the beginning, creates empty frame and then proceeds to translate received message to binary data that is understandable by Scylla.

4.2.4. Paging Handler

Paging Handler allows to divide response from the database to multiple pages, each having arbitrary, maximal number of rows. It has wide amount of available operations. At the beginning it provides previously mentioned paging mode. User can turn it on and provide desired number of rows that will come with each page. Of course this mode can be turned off in a single API call. When the paging is on, every query request will result in a single page from the database. Driver provides the current page number to the user to indicate him on which page he currently is. Moreover it shows whether previous or next page exist. If so user can ask for adjacent page. Then the request will be send to the Scylla database and if there are no internal errors on the database side, the new page will be delivered to user. It is worth to mention that ids of previous pages are stored to provide quick lookup to pages that were previously delivered, without it, asking for previous pages would be impossible as the CQL protocol does not allow for previous page request as.

4.2.5. Prepare Sender

This module is very similar to the Query Sender. It allows user to prepare certain query. Preparing is about sending query message but without executing it. It also allows for incomplete query format (User can indicate that certain fields will be delivered while executing). Main purpose of preparing queries is to greatly improve efficiency of multiple request that are very similar to each other. It saves a lot of time and resources that are required to parse whole request over and over again. The driver gets message needed to be prepared, converts

it to binary format and sends it. In response database should deliver id of prepared queried that is necessary to execute it.

4.2.6. Execute Sender

Execution Sender is quite a simple module. It executed the query prepared before. As the arguments, this sender takes the id of the prepared query and the list of arguments that will be placed in the blanked places of the prepared query and sends proper message to the database indicating that the user wants to execute the query. Paging also covers execute messages. After receiving a response it is possible to move through all available pages resulting from the query prepared before.

4.2.7. Consistency Changer

Consistency Changer provides possibility of changing consistency level. This variable indicates requirements for success of certain operation. After providing a consistency level to the driver, it will be used in future requests. At the beginning it is set to 'All'.

4.2.8. Type Converter

Type converter is a very important module. It allows for two sided conversion between human readable input and binary form. This module is used in almost every other one as conversions are required on almost every stop. There are over 20 distinct CQL types that require proper handling. Without possibility to convert a message to CQL form executing queries would be impossible and without other side conversion presenting results of queries would be non-readable.

4.2.9. Authentication Handler

Authentication Handler sends credentials to the Scylla when it requires authentication. Password requirement can be set in Scylla configuration file. Driver enables option to properly authenticate, it accepts login and password and sends it directly to the database.

4.2.10. Response Handler

This module handles all responses received from the database. User should provide setter of variable that will hold results coming from database. It is required as the driver listens to the WebSocket with Scylla and directly modifies the result instead of letting user wait for the response. It is useful as sometimes driver will respond instantly in case of an error.

4.2.11. Error Handler

When CQL frame has the 'ERROR' opcode, Error Handler comes handy. It translates error message written in binary form to readable format while adding extra information that is not explicitly encoded. Moreover handler provides data to the user that an error occurred. It is very useful as the message can be shown in red. This should be clear sign to the user that something went wrong.

4.3. Implementation

4.3.1. Technology

Whole driver was written entirely in TypeScript, there were no necessity for any other language. It allows for an easy access from the website. But driver does not need to be run in the browser. It might as well be build without it. This way we could use plenty of modules that made whole development process easier as well as test our product without a browser.

4.3.2. TypeScript

Use of TypeScript as previously mentioned allowed us to use a lot of different modules. The most important one is a WebSocket module. There was no need to write distinct WebSocket driver for web development when the solution already existed. Moreover it became a great ground truth as it could be assumed that it is correct when testing WebSocket handler. Without it there was a chance that in both WebSocket drivers some malfunction would exist. Other modules mostly regarded conversion and formatting issues. TypeScript is quite limited in terms of type abundance. It was necessary to provide data structure that could easily handle multiple operations on binary data as the driver required so. Furthermore introduction of big int made everything easier as big int in opposite of default number type was able to represent eight byte variables. Moreover using TypeScript allowed us to write cleaner code as everything has to be typed. It greatly reduced amount of runtime errors as almost everything was previously checked during type checking.

4.3.3. Development

Whole process followed very iterative trait.

- **Proxy Server** - At the beginning proxy server in python was implemented. It was needed to communicate with Scylla as the database previously provided only direct communication via TCP. Proxy server had a very straightforward design. It connected to database and waited for WebSocket clients to join and send communicates that would be forwarded directly to Scylla. After proxy server was created it was possible to actually send something to Scylla.
- **Auxiliary Functions** - Afterwards all extra functions were implemented which have been used later in all other modules. Mainly they were focused about creating and decoding binary objects to facilitate handling of frame objects. Also consistency level handling was implemented at this point.
- **Creating frames** - In the next step simple frame factory was created. Its main purpose was to test communication with database as well as providing a base for more advanced creating modules. Ensuring that Scylla was getting correct frames was critical to any further development.

- **Handshake** - After sending correct frames all returning messages consisted of an error message wrapped in CQL frame showing that CQL connection has not been initialized with handshake. To move forward it was necessary to create handshake frames. After their implementation it was possible to proceed further and focus on actual request from database.
- **Query** - Implementation of sending queries was not complicated. All that was needed was to wrap query body inserted by user into CQL frame with addition of some parameters. User was able to create keyspaces, tables, insert data into them as well as asking for content. However received data was non readable.
- **Showing results** - Therefore it was needed to decode all received information. This however was very time consuming, there were multiple types of results. Each one of them needed to be handled. After successful implementation of protocol concerning results and type conversion it was possible to get human readable format of all communicates coming from the database as well as tables coming as a result of 'Select' queries. At this point it was possible to communicate with Scylla and actually work on it. However there were yet some features to be implemented.
- **Paging** - First of those features was paging. It allowed to ask for certain page sizes and traverse whole table smoothly. Extra data was attached to each query to provide database information about user expectations. Also location of next pages started to be retrieved from database results. This turned into a fine module which functionalities exceeded those coming from other CQL drivers.
- **Prepare & Execute** - Another great functionality was to provide possibility to prepare and execute queries. Prepare was very simple to implement as the body of prepare message was almost the same as the query one. However execute was a bit harder. Mostly it needed conversion between input provided by user to binary representation which was not trivial in some cases. Therefore type conversion module was greatly enhanced to enable this feature.
- **Authentication** - Last but not least was giving possibility to log into databases that required login and password. If specified in Scylla configuration file, proper authentication is required. If our driver would not allow to access such databases it would be as good as useless. Authentication is possible in our driver in two scenarios. First of them is trying to authenticate alongside handshake. Therefore if credentials are not required they will not be sent. However if information about their necessity comes they will instantly be delivered to database. Second one is to just send authentication message in any moment. If user already has been authorized, Scylla will respond with error message. Otherwise an attempt to authenticate will be made. Any non successful try will be responded with specific error communicate.

4.3.4. Challenges

Implementation of the CQL driver brought a lot of challenges:

- **WebSockets** - One of them was the issue with WebSockets. Each browsers provide WebSockets to some extend but their implementations, functionalities and behaviours differ. However without any browser it is impossible to use WebSockets. Fortunately there is WebSocket library that enabled driver to work even without browser. But an issue arose, including WebSocket library conflicted with native WebSockets built in to browser. After hours of research a solution has been found and implemented. CQL driver started to recognize if environment belonged to the browser or not.
- **Types** - Another challenge was in managing all required data types. A lot of time was spent to find all details regarding TypeScript types and their limitations as well as build in classes. Afterwards research was made to include all libraries that could help in proper conversions. Unfortunately a lot of libraries were very limited. It resulted in huge amount of work during implementation of type conversions.
- **Debugging** - Although the biggest challenge was in finding bugs. Sometimes the frame was incorrect and response from the database was not precise enough to understand it. Debugging code that runs in the web browsers is very hard due to nature of environment. There are no breakpoints to easily access variables in a specific moment and check their value. Of course there is an brute-force option to write certain information to the web console and compare expected results to actual ones. However it results in a huge amount of non organized data and finding error becomes tedious with this approach. A better option is to track actual data frames with Wireshark at the beginning of debugging process. With proper filtering a lot of information could be found and the most important, there was a view of the exact data that Scylla received. Sometimes bugs were very visible when looking at binary frame in Wireshark and entirely not existent on the driver side. Thanks to this tool debugging challenge was not as distressing as it could be.

Chapter 5

In-browser Scylla terminal

The third, the most front-end part of the project was a browser terminal for the entire project. Here user connects to the Scylla database, sends commands, modifies or requests data and receives it on the conveniently and user-friendly designed response displayer.

We based our project on the terminal from Linux systems. As the main traits of this project, we chose minimalism, user-friendliness, easiness of operations. Dark-mode was chosen rather than light-mode for the colour theme for the projects as it generally regarded than less tiring for eyes and preferred by users. Dark theme is also the default and most commonly used colour scheme of virtually any IDE and terminal.

5.1. Existing alternatives

We did some research on similar projects, terminal templates or ideas and existing node packages, however the best we found was this one react-terminal npm package^[20].

Firstly we decided to incorporate it to our project, however it quickly proved to be difficult to modify and with very limited possibilities for developing new features. Therefore we decided to write our terminal in React with TypeScript and Material UI from scratch.

Another advantage of such way is that we - the developers - can better understand it as it is easier to modify each underlying subpart of the project.

5.2. Designing project

5.2.1. Composition

The browser terminal is divided into three main parts - command history, input section, server's response displayer - and a pop-up form which at the beginning asks the user to provide information for authorisation and/or connection.

5.2.2. Connection & authentication pop-up form

This part is launched at the beginning, and while this form is displayed the rest of the page is overshadowed and disabled. There user can insert address and port for server with Scylla database to which they intend to connect to, username and password for authorisation. Then query is sent to the server to check if given credentials are correct. In the meanwhile loading icon is displayed instead of the form. Upon successful verification application is connected to the database and the user can access the main part, otherwise respective error is displayed and user is asked to re-enter credentials.

5.2.3. Command history

The top part of the application. Shows ten of the lastly used commands. User can quickly re-enter commands from the history to the input part using up and down arrows. History can be cleaned by typing *Clear* (not case sensitive).

5.2.4. Input section

This is the place where user can write CQL commands which will be subsequently send to the server and executed there.

Input allows user to switch between two types of formatting **short** (default, can be set by typing *short* in the input) - commands will be send after pressing enter, and **long** (is set by typing *long* in the input) commands must end with semicolon, otherwise pressing enter will append endlime, allows user to insert multi-line commands.

Instead of typing command user can also copy-paste it from external source or access history of commands with up and down arrow keys, then press enter to re-enter chosen command.

Another feature controlled from this part is paging control, *PAGING ON n* will set database to divide large responses into chunks (called **pages**) of *n* rows, respectively *PAGING OFF* (default one) will turn off and make portions of data obtained from database to show up no matter how large they are.

5.2.5. Response displayer

Displays the latest response from the server. Executing another CQL code will result in receiving new response which will be shown instead of the previous one. Displays works with two type of responses: single line responses and table response. For table response (if paging is enabled) application provides user with buttons for moving between pages (set of rows from larger response) on the bottom.

Displayer automatically detects if the response is an error and shows it in red colour and bolder font.

5.3. Implementation

5.3.1. Technology

Entire terminal is made in React with TypeScript, for CSS feature we used Material UI framework. All connection to the server/database is done through the already mention driver module.

5.3.2. Functional React with TypeScript

Employing React was the obvious choice as it is widely considered the most popular technology for such matter so there is a plethora of resources, manuals, libraries and frameworks available online.

We chose functional React - not the objective one - as it is more modern, works better with external libraries (especially Hooks proved to be a great tool), and all of us prefer the functional paradigm rather than the objective one.

The last important choice made here was to use TypeScript or the regular JavaScript, we went with the first option as it gives a better handle on controlling dataflow, it is more readable

and (as we give the project to the Scylla's software family) will be easier use in the further development.

5.3.3. `useState` module (State Hook)

The most important hook in the terminal, it dynamically stores data received from the user and driver. It work perfectly smooth with the React components, as it automatically update the displayed information on the front website. Apart from storing every variable used in the terminal it is used to connect raw HTML with Material UI CSS classes resulting in pleasant for eyes' of user website structure.

Another alternative would be Redux, however it was decided that State Hook is a much more elegant, readable and comfortable to use solution.

5.3.4. `useEffect` module (Effect Hook)

This hook is used to dynamically check and update components after new response from the server arrives or user inserts new command into terminal. We chose it as it makes instant updates on elements after given condition occurs and does not require constant refreshing to check the condition.

5.3.5. `useMemo` module (Memory Hook)

This hook is used only once - to hold once initialised driver object - however it is of a vital role as it prevents React to re-render and therefore costly re-initialise (and probably lose some data in the process) the driver each time something in the application is done.

5.3.6. Material UI

We chose this framework as it is simple and easy to use and works smoothly with State Hook and React. Framework detailed description and documentation is available at their website^[21].

5.3.7. Components division

Terminal is build from the hierarchy of React component with the main functions and connection to the driver being defined in the Terminal component and then passed down to other elements:

- **Launch Form Component**

Component is displayed only when connections to the database has not been initialised yet or it requires further authorisation. When this element is on all other functionalities of the terminal are disabled.

This components inherits functions to commence connection to the database server, holders and modifiers for username, password, address and port. Launch Form on its own consists of just two elements displayed interchangeably - loading animation from react-loading-icons module (more information at module's page^[22]) while application awaits response for the query and form when user is asked for the credentials.

After submitting form the component does some basic check-up if they are not null, and in such cases does not move forward with the query, instead displays message to the user to fix their input.

- **Terminal History Component**

This simple elements inherits access to just the list of last used commands from the main component. It then creates a table from ten most recent ones to display on the top of the screen. Thanks to Effect Hook it does everything smoothly, dynamically and automatically after any new command is entered to the terminal.

- **Input Component**

This components inherits command modifying function, its value and currently set keypace name. Input components then divides further into a keypace displayer on top - information in which user currently is, this information dynamically changes upon executing command to move to another keypace. Second part is the textarea - probably the most important piece of the whole front-end application. Here user insert commands for further executing at the terminal (like clearing history or switching between input format), or in the most - to be sent to the driver and then server.

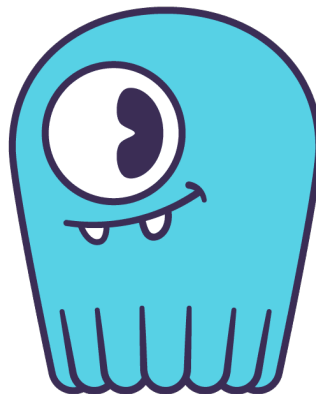
There is no need for passing input format indicator to here as parsing and analysing input is done in the main component.

- **Server Response Component**

This component consist only (bar the banner saying *Response* and auxiliary formatter for showing error) of two different structures to display both types of responses from the driver. First would be for simple one-liner response (which can also be the error message), the other - **Table Displayer structure** - is more complex as it displays tabled response, considering the tools for the paging feature (next/previous page, current page counter, moving response to the next chunk etc.).

- **Logo Component**

This element is just a simple watermark of ScyllaDB logo, it is permanently displayed in the bottom right corner with 20% opacity as not to cover anything on the application.



5.3.8. Iterative development

Build the terminal was lengthy and challenging process, therefore we employed iterative development. Firstly - without even connecting terminal to the database - we built a first semi-working front-end application, which consisted of a simple terminal with poorly working input (without format options and history editing), commands history and response part that

just printed out whatever it received. However, what really matters, it worked, it fulfilled its basic, most primitive goal, we tested it and then connected to the driver and database (via temporal, auxiliary proxy server). After that we just kept repeating the pattern of adding new features (such as table displayer, paging, functionality of keyspace, launch form etc.), testing them, presenting them to our supervisor from Scylla on our weekly and then repeating this work pattern. The last feature to add (not counting testing everything thoroughly) was implementing the authentication feature to the launch panel, which conveniently take care of handling multiple failed attempt of connecting to the database.

5.3.9. Challenges

The first, probably the greatest, challenge was the integration of the parts we did separately. Each found bug or working incorrectly feature needed to be widely analysed to find the place of origin (which part of the application malfunctioned) and then find the best approach to fix the project. This was usually solved together quite swiftly and smoothly as we had an excellent teamwork and great communication between members.

Second challenge, I think would be organising work. Project was quite large, lasted almost a year, and in the meantime all of us had a university duties to attend. However, thanks to the good planning, sticking to the plan, iterative development, regular meetings when we discussed current progress and issues, we moved forward every week and finally finished everything we had to do.

Last challenge would be frontend design as it requires not only raw, analytical planning but it also needs to be looking attractive and pleasant to average user who may have virtually none programming knowledge. This required a series of consultation, changes in website's appearance and manual testing to check whether all elements fit properly, and are displayed neatly and visibly.

In the end we can say, that we coped with every issue we faced and created a satisfying result.

5.3.10. Visual appearance

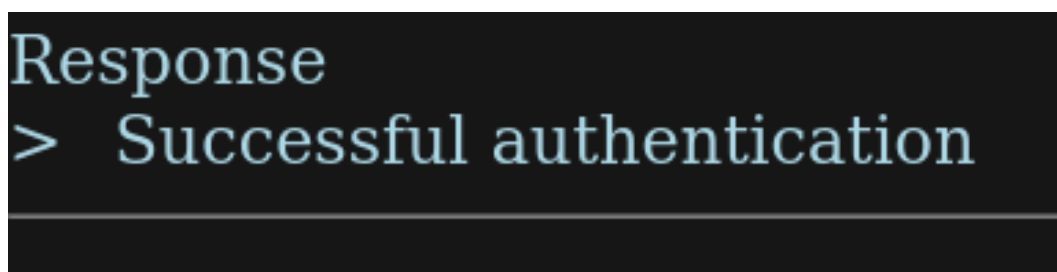
Developing the visual appearance of the application was a complex problem, because it consisted not only of designing something that works properly, but also is pleasant for eyes of regular user - something that cannot be measured objectively. Of course we made sure that things that are objectively negative - such as overlapping elements, unreadable font colour, flickering screen, disappearing text are not happening in any case. We kept all main elements in the same, dark-themed colour palette, with navy background and lightcyan font.

A dark blue rounded rectangle with a black border. It contains four labels: 'Address:', 'Port:', 'Username:', and 'Password:'. Each label is followed by a light gray input field. The 'Address' field contains 'localhost' and the 'Port' field contains '8222'. The 'Username' and 'Password' fields are empty. At the bottom center is a light gray button labeled 'Connect'.

Launch from

A dark blue rounded rectangle with a black border. It contains two labels: 'Username:' and 'Password:'. Each label is followed by a light gray input field. The 'Username' field contains 'test' and the 'Password' field contains four black dots. Below the input fields is the text 'Authorisation failed!' in red. At the bottom center is a light gray button labeled 'Authorise'.

Reathorisation form

A dark gray rectangular area with a thin horizontal line near the bottom. It contains the word 'Response' in a large, light blue serif font. Below it is a light blue prompt character '>' followed by the text 'Successful authentication' in the same light blue serif font.

Single-line response

```
Response
> Syntax Error: line 1:0 no viable alternative at input 'wrong'
```

```
Response
> Invalid: Keyspace 'keyspace_name' does not exist
```

Displaying errors from response

Response						
key	bootstrapped	broadcast_address	cluster_name	cql_version	data_center	gc
local	COMPLETED	127.0.0.1		3.3.1	datacenter1	

Tables response

host_id	li
4-4fe5-8c65-4de95b419358	

Scylla DB logo (transparent enough so the background is easily readable)

```
History
> select * from system.local
> dsdssdsd
> dssd
> select * from system.local
> TEST 2137
> select * from system.local
> dsdssdsd
> wrong command
> select * from system.local;
> select * from system.local
```

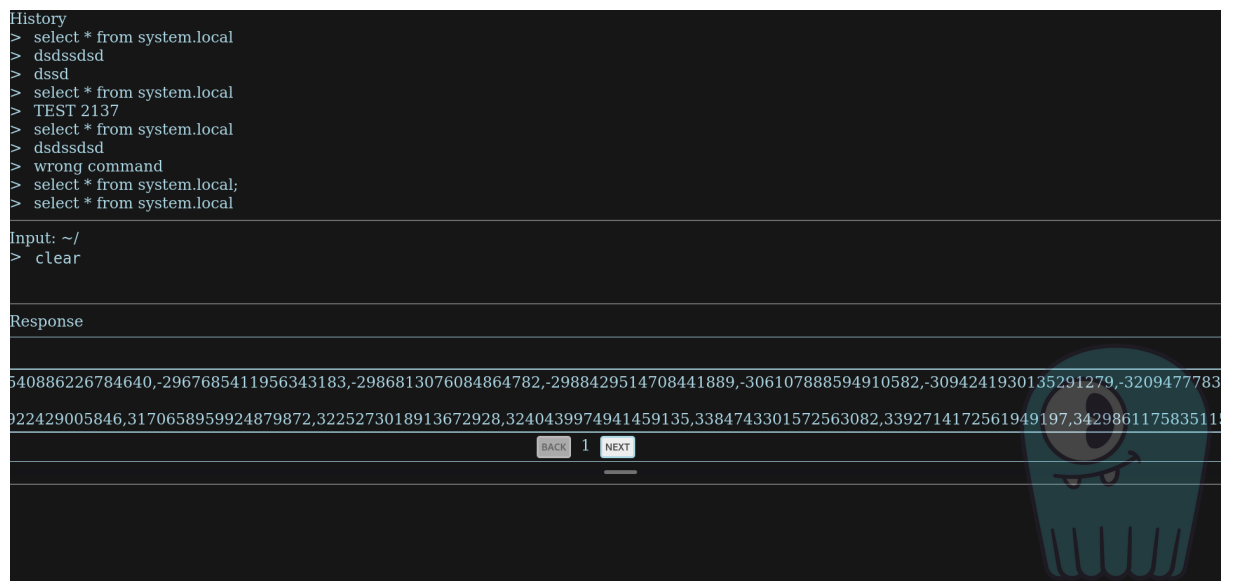
Command history

```
> use cycling

Input: ~/cycling/
>

Response
> Changed keypace to cycling
```

Input component (with *cycling* keypace set)



[View of the entire application](#)

Chapter 6

Division of work

Before starting the project we had divided it between the team members so that everyone could specialise in one area, however there was constant communication amongst us, both during designing and implementation processes to fill in the knowledge gaps and experience gaps and make the final integration seamless. The Final Bachelor's paper was written together, everyone at first described the topics they were working on in order to provide the best quality of report and then a group review was performed.

Andrzej Stalke

Alongside Bartłomiej, Andrzej designed and implemented the WebSocket server described in this paper. He also took care of the whole integration process which enabled ScyllaDB to use the aforementioned server.

Bartłomiej Kozaryna

Together with Andrzej, Bartłomiej was responsible for the server implementation. He participated in the general structure and the first parser implementation and afterwards was more focused on developing tests using seastar and the TLS server version. Moreover he undertook the code review for driver and terminal and added a few minor stylistic touch ups to the front-end.

Daniel Filimonow

Daniel designed and implemented the CQL driver in TypeScript based on the protocol specification^[19].

Grzegorz Zaleski

Grzegorz was the main creator of the terminal part of the application. Due to the nature of how driver and terminal work together many part of the terminal were consulted and implemented with Daniel.

Bibliography

- [1] Jeff Desjardins: How much data is generated each day? [online] <https://www.weforum.org/agenda/2019/04/how-much-data-is-generated-each-day-cf4bddf29f/> [Access: 01.06.2022]
- [2] Statista Research Department: Media usage in an internet minute as of August 2021 [online] <https://www.statista.com/statistics/195140/new-user-generated-content-uploaded-by-users-per-minute/#:~:text=Media%20usage%20in%20an%20online%20minute%202020&text=A%20lot%20of%20things%20happen,are%20streamed%20by%20users%20worldwide> [Access: 01.06.2022]
- [3] ScyllaDB Benchmarks [online] <https://www.scylladb.com/product/benchmarks/> [Access: 01.06.2022]
- [4] ScyllaDB Clients [online] <https://www.scylladb.com/users/> [Access: 01.06.2022]
- [5] ScyllaDB Technology [online] <https://www.scylladb.com/product/technology/> [Access: 01.06.2022]
- [6] C++20 Information [online] <https://en.cppreference.com/w/cpp/20> [Access: 01.06.2022]
- [7] RFC 6455 [online] <https://datatracker.ietf.org/doc/html/rfc6455> [Access: 01.06.2022]
- [8] Rust WebSocket Library [online] <https://crates.io/crates/tungstenite> [Access: 01.06.2022]
- [9] Python WebSocket Library [online] <https://websockets.readthedocs.io/en/stable/> [Access: 01.06.2022]
- [10] C++ Boost WebSocket Library [online] https://www.boost.org/doc/libs/1_79_0/libs/beast/doc/html/beast/using_websocket.html [Access: 01.06.2022]
- [11] WebSocket++ Library [online] <https://docs.websocketpp.org/> [Access: 01.06.2022]
- [12] C WebSocket Library [online] <https://libwebsockets.org/> [Access: 01.06.2022]
- [13] Prolog WebSocket Library [online] <https://www.swi-prolog.org/pldoc/man?section=websocket> [Access: 01.06.2022]
- [14] Haskell WebSocket Library [online] <https://hackage.haskell.org/package/websockets-0.12.7.3/docs/Network-WebSockets.html> [Access: 01.06.2022]

- [15] Seastar Documentation [online] <http://docs.seastar.io/master/index.html>
[Access: 03.06.2022]
- [16] Documentation of data_sink class [online] http://docs.seastar.io/master/classseastar_1_1data__sink.html [Access: 03.06.2022]
- [17] Documentation of data_source class [online] http://docs.seastar.io/master/classseastar_1_1data__source.html [Access: 03.06.2022]
- [18] Documentation of input_stream class [online] http://docs.seastar.io/master/classseastar_1_1data__source.html [Access: 03.06.2022]
- [19] CQL Specification [online] https://github.com/apache/cassandra/blob/trunk/doc/native_protocol_v4.spec [Access: 01.06.2022]
- [20] React Terminal Library [online] <https://www.npmjs.com/package/react-terminal>
[Access: 01.06.2022]
- [21] Material UI Website [online] <https://mui.com/> [Access: 01.06.2022]
- [22] React Icons Library [online] <https://www.npmjs.com/package/react-loading-icons>
[Access: 01.06.2022]
- [23] ScyllaDB Main Repository [online] <https://github.com/scylladb/scylla>
[Access: 03.06.2022]
- [24] Seastar Main Repository [online] <https://github.com/scylladb/seastar>
[Access: 03.06.2022]
- [25] Repository with CQL Driver [online] <https://github.com/dfilimonow/CQL-Driver>
[Access: 03.06.2022]
- [26] Repository with Web Terminal [online] <https://github.com/gbzaleski/ZPP-ScyllaDB-Front>
[Access: 03.06.2022]

Appendix A

The structure of the source code

A.1. WebSocket Server

A.1.1. Seastar

- seastar/
 - CMakeLists.txt - It was modified to add new files to the project.
 - include/seastar/websocket/server.hh - Header file of the server code.
 - src/websocket/server.cc - Source code of the server.
 - demos
 - * websocket_demo.cc - Example WS echo server
 - * websocket_secure_demo.cc - Example WSS echo server
 - tests/unit/websocket_test.cc - Tests

A.1.2. ScyllaDB

- scylla/
 - CMakeLists.txt - It was modified to add new files to the project.
 - configure.py - It was modified in order to add new files to the project.
 - conf/scylla.yaml - The configuration file of the ScyllaDB. It was modified to add the ability to configure the port and the address of the server.
 - db - Files in this directory were modified in order to add the new configuration options.
 - main.cc - The main function of the ScyllaDB was modified to start the WebSocket server.
 - websocket - Files in this directory were created in order to start the WebScket server.

A.2. TypeScript CQL Driver

- src/
 - Driver.ts - Representation of the whole driver.
 - utils - Directory consisting of multiple utility functions.
 - cql-types - Directory consisting of files that provide CQL types and their conversions.
 - functions - Files in this directory are used directly by the driver for its method implementations.

A.3. In-browser Scylla terminal

- src/
 - main.ts, App.tsx - Main react files wrapping entire project together.
 - consts.js - Set of global constants used in throughout the project.
 - assets/logo.webp - Picture of ScyllaDB logo used as watermark on the project.
 - components/
 - * Terminal.tsx - Main component containing entire terminal with its subcomponents.
 - * LaunchForm.tsx - Component with form for logging-in and authenticating connections.
 - * TerminalHistory.tsx - Component displaying history of used commands.
 - * Input.ts - Component where user writes CQL and terminal commands.
 - * ServerResponse.tsx - Component which receives, parses and displays responses sent by the database server through driver.
 - * TableDisplay.tsx - Auxiliary component for displaying table responses.