JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Patrick Nolte
k01618097

Submitted at
Institute of Networks and Security (INS)

Supervisor and First Examiner
Univ.-Prof. Dr. **René Mayrhofer**

April 14, 2021

SECURE EXPORT OF CHAT HISTORY FROM WIRE

Bachelor Thesis

to obtain the academic degree of

Bachelor of Science

in the Bachelor's Program

Computer Science

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

JOHANNES KEPLER
UNIVERSITY LINZ

Sworn Declaration

I hereby declare under oath that the submitted Diploma Thesis has been written solely by me without any third-party assistance, information other than provided sources or aids have not been used and those used have been fully documented. Sources for literal, paraphrased and cited quotes have been accurately credited.

The submitted document here present is identical to the electronically submitted text document.

Linz, April 14, 2021

JOHANNES KEPLER
UNIVERSITY LINZ

Abstract

This Bachelor Thesis is about the development of the secure export of chat history from the messenger app Wire. Wire is an end-to-end encrypting audio/video/chat service for various platforms. The aim of this Thesis is to expand the open source Android client in such a way that a secure export of an entire (group-) conversation, including the media it contains, is possible. Additional reference is given for restrictions such as time-limited messages. The export is done as a Zip file, which contains the messages in an XML document as well as the media files. Additionally, an HTML-Viewer can be included to view the exported data.

The complete code can be found at GitHub[8].

Zusammenfassung

Diese Bachelorarbeit handelt von der Entwicklung eines sicheren Exports der Chat-Historie der Messanger-App Wire. Wire ist ein Ende-zu-Ende verschlüsselter Audio/Video/Chat-Dienst für verschiedenste Platformen. Das Ziel dieser Arbeit ist es, den Android-Client so zu erweitern, dass es möglich ist, vollständige (Gruppen-) Konversationen, Mediendateien eingeschlossen, sicher zu exportieren. Außerdem wird Bezug genommen auf Einschränkungen wie Zeitlimitierte Nachrichten. Der Export wird als Zip-Datei vorgenommen, welche die Nachrichten in einem XML-Dokument, sowie die Mediendateien enthält. Desweiteren kann ein HTML-Viewer hinzugefügt werden, welche es ermöglicht, die exportierten Daten anzusehen.

Der vollständige Code kann auf GitHub[8] gefunden werden.

JOHANNES KEPLER
UNIVERSITY LINZ

Contents

JOHANNES KEPLER
UNIVERSITY LINZ

List of Figures

List of Tables

1 Motivation

The majority of the population today is using messenger services everyday. This leads to a lot of conversations being handled by these services. Sometimes even messages that would normally be sent by email or letter are sent with these services. This raises the need to secure these messages to access them later. In addition, these messages should remain available even if the device is changed or breaks. Most messenger services provide continued access by providing the possibility to create backups. But these backups can only exchange messages between the same service on different devices and sometimes do not even support different operating systems. Additionally, they are limited to the messenger service that created the backup and are not designed to be used by other services.
Because of this limitation an export service is a very good feature to have. It provides the possibility to view conversations independently of the service used to send them. Additionally, the format of the export is openly accessible and can be implemented by other messenger services to enable the import of messages sent by other messengers.

In this Bachelor Thesis, I have implemented the feature to export conversations made with Wire. The data is exported in a way that is easily accessible outside of the service itself. Additionally, the media data included in the conversation is also exported.

2 Comparison with other systems

To get an overview of the current status I would like to take a look at the current export features of some messenger services. I chose Elements, Facebook Messenger, Line, Signal, Skype, Telegram, WhatsApp and Wire.

2.1 Current state of the export feature

2.1.1 Elements

Elements currently does not have an export feature.
Instead, the encryption keys can be saved (encrypted) on the device and imported on other devices. This enables the user to restore messages from the server with these keys. The server (that could be self hosted) is storing the "backup" of the messages. The compatibility with other services is provided by using the Matrix open standard for interoperable, decentralised, real-time communication over ip. Unfortunately this means, that it is currently not possible to export old messages without writing a custom tool for this.
I only found one third-party tool, named Matrix Recorder [10], that provides something similar. It registers as a regular client at the server, receives all messages and stores them with the possibility to export them to HTML. Unfortunately it is not easy to set up and is currently not able to save messages sent before the setup of the tool, because they were encrypted with the keys for other devices and, until now, it is not possible to import them at the moment. Additionally, this tool is no longer actively developed (last update 3 years ago).

2.1.2 Facebook Messenger

Facebook supports the download of all messages (and other data they have on you) in the settings on their website. The chat history can be requested as JSON or HTML data. When the request was processed the data can be downloaded at the website. The download is a ZIP file containing the messages in the specified format and all files sent. [3] The European Union has a law, the General Data Protection Regulation, that requires companies to enable every person to request the data the company has about the person. To avoid a lot of support requests, bigger companies often

provide the possibility for the user to download the data directly. Facebook makes the export of the messages accessible in the same menu as a lot of other download possibilities about data the company has, so it looks like the export of the messages was implemented to follow this requirement, especially because it is not really easily available. On the other side the possibility to not only download the data in JSON but also as HTML is provided, so it is easier for most people to access the conversations and read them and the export contains the messages of the other people in the conversation, which would not be required by the law.

2.1.3 Line

Line has multiple possibilities to export the chat history. One is to simply export the messages as text on the device. Additionally, they developed a tool called "UltData" to provide the possibility to export data. When using "UltData" it is possible to export not only the messages but also photos, videos, audio, documents, contacts and the call history. This tool needs the device to be plugged to a pc with USB debugging enabled. [14]

2.1.4 Signal

The messenger service Signal does not support the export of conversations directly.
However, it is possible to create an encrypted backup within the app and this backup can be decrypted with third-party tools like "signal-back" [13]. The backup is secured by a 30 character passphrase/key. The data must then be further processed to be easily human readable.
The desktop app uses an encrypted SQLite database that would need a similar processing. [1, 11]

2.1.5 Skype

Skype does support the export of the chat history. They have a website, where a TAR file can be requested containing all messages and/or files, with files being available for 30 days after they were sent. The requested TAR file can then be downloaded from the website and contains the requested files and a JSON file with all messages. [12]

2.1.6 Telegram

Telegram does not support the export of conversations in their app, but the desktop application has this feature.
The export may include photos, videos, voice messages, video messages, stickers, GIFs and other files.

The export is done in form of a folder containing an HTML document with the conversation data and some subfolders with additional files to display the HTML site and the (media-) files in other folders.

2.1.7 WhatsApp

In WhatsApp the export is supported. The exported data is sent by email and thus limited to the 40000 newest messages. If media data is included only the newest 10000 messages are exported next to the media files. [4, 5]
However, it is also possible to use the backup data, similar to Signal. The backup is readable with third party software and the decryption needs the Android account email and the corresponding key file. There are a lot of commercial tools to do this like "Wondershare MobileTrans - WhatsApp Transfer" [15] and some free tools like this "WhatsApp Viewer" [7].

2.1.8 Wire

Wire currently does not support the export of conversations.
It is possible to decrypt the backup with third-party tools and the passphrase used at the backup creation, but the backup does not contain any media files.
My Bachelor Thesis changes this and adds the export feature to the Android app. The export I implemented supports all current message types that may be sent, including text messages as well as audio files, video files, images and documents. [2]

2.2 Advantages

The advantages of an implemented export are prominent. First it allows exchange of messages between different messenger services as soon as all agree on one format or implement the import/-export of the other messenger services. Another advantage is the possibility to access exported data independently of the service used to send the messages. This enables the possibility to switch the messenger service more easily.
The export of messages and import into other messengers could be a first step towards the currently by the European Union targeted possibility of message exchange between different messenger services. It enables users to have more freedom in the choice of the messenger they would like to use, because the conversations held with another messenger are no longer lost when switching the service.

2.3 (Security-) problems

There are some major problems with the export of conversations. One big issue is the agreement between all messenger services to use the same format to enable the import into other messengers. Currently there is no uniform system available. Most services providing an export use different formats or a format that is not able to reconstruct the conversation completely. An example is the handling of exports by different services. Telegram provides the conversation as a web page and is focused to be an easily human readable system. WhatsApp on the other side exports a simple text file, where only minimal information is provided. My implementation in Wire exports the conversation as an XML document with all data needed to fully reconstruct the conversation. Additionally, it is possible to include a web page to display the data, similar to the approach of Telegram.
Another problem is the missing encryption of exported data. As soon as the data is exported, everyone with access to the files can read them. This is particularly true for the WhatsApp export, which is sent unencrypted by email.
The missing encryption cannot be avoided, if the goal is to make the conversation easily readable in a web browser, but the ZIP file could be encrypted so that it could savely be stored on the mobile device and transferred. Only at the target device, it would need to be decrypted directly before it should be viewed.
Another point is the handling of messages with an expire time. At the moment all messages that are already expired are encrypted and cannot be exported. All messages that will expire are exported normally and the *HTML-Viewer* encrypts them when they should be displayed. This means, that they are still inside the **chats.xml** and can be extracted there. It is debatable if it would be better to encrypt them directly when exporting, even if it means that they are no longer readable in the export even before the time has expired.

2.4 Comparison of exports and (cloud) backups

The main difference between a local export and a backup is that the backup is not intended to be read by the user. Additionally, it is not a goal of a backup to enable the user to exchange the data between different services. It is designed to be used to make sure that the data is not lost and can be recovered. The export on the other hand is mainly used to either create the possibility to import the data within another device and/or service and may not only be used by the service itself, or it is used to enable the user to read the data independently of any service, just to have the possibility to access the data in a readable format. If the export is designed to be imported it could be used as a backup because it would enable the user to restore the data. If it is exported to be an independent, readable version for the user to read, it could be exported to be used as a backup, but it could also

mean, that only a text file is exported, like some services described in section 2.1 provide and these are mostly used to enable the user to read the conversation, but not to import the data.
The selection of the included data also differs. The backup normally does not provide the possibility to save only some conversations, instead all messages are saved, maybe with the option to exclude additional files. An export is mostly used to export specific conversations and thus the possibility to select the exported conversations is provided.
Another difference between an export and a backup is the location, where it is saved. A lot of services provide, or even enforce, the possibility to save the backup on their own servers or at other cloud storage providers, whereas an export is normally only saved at the device itself and is managed by the user.
I would like to discuss some advantages and disadvantages I found in Obrutsky's document [9, p. 3]. Lets have a look at the advantages first. One argument is that only the used storage must be paid and this is a good point, because sometimes it does not even cost anything to create a backup for example at Google Drive. Another point is the accessibility of the data. This is valid as well and can be seen when someone switches the device and installs, for example WhatsApp. Because the user is already logged into the Google account, WhatsApp can see that there is a backup in the Google Drive folder of that account and automatically provides the option to import the old data from the backup. Another valid point is the better protection in case of a disaster. If the backup is only saved on a mobile device, the chance of a disaster is fairly high and if the backup is saved at the local computer as well, the risk is a lot lower, but because the computer and the mobile device are often located at the same place, it is not really safe as well. When the backup is saved in the cloud it is normally fairly safe, because the cloud storage provider normally ensures that there are backups that are located at different locations and ensures the safety of the data. This also includes the problem of hardware failures mentioned in the paper. Another point is the storage limit. Many mobile devices do not have a lot of storage (also this is changing and most devices today have enough storage), but it is easier and often cheaper to enhance the storage at the cloud provider if needed.
But there are also disadvantages to the usage of cloud storage. At first there are additional costs that sometimes come with the cloud backup. Another point are bandwidth limitations. If the service makes backups often, it creates traffic, if it must be transmitted to the server as a backup. Depending on the type of backup (full/incremental) this may become more and more severe as time flows and the amount of data grows. This is very relevant for mobile devices where the bandwidth is normally limited, especially if media data with large files like images and videos is included. The most important disadvantage in my opinion is the security. If the data is saved inside of a cloud storage, the data is no longer in the hands of the user. It can be viewed by unauthorized people. The risk can be lowered, if the backup is encrypted, but encryption alone is not enough. Some messenger services like WhatsApp do create the backup, encrypt it and save it to some cloud storage. This could lead to the assumption, that the data is save, but because the user does not provide the password for the encryption, the encryption is done completely and automatically by the service. This enables the service to decrypt the data, because they have all the information to do that. This

is also a point of discussion, because the law enforcement in the USA can get everything needed to decrypt the backup from the service and can access the encrypted backup of the cloud provider, if both are located in the USA, and thus they can encrypt the backup. This is a big problem, because some of the biggest messenger services and cloud storage providers do operate from the USA and must provide the requested data. This does also include non US-citizens.
This is why it is important to give the user the opportunity to choose the preferred location of the save. Backups often do not let the user choose the location of the save. When the data is exported, this is normally done locally and the user has the full control, where it should be saved. It can remain on the device, could be moved to a computer, NAS or it could be saved in the cloud. The user also has the possibility to encrypt the data additionally to the existing encryption that may be provided by the service.
Like often in the area of security there is a choice between security and usability. It is easier, if the service handles the complete backup process and the user does not need to do anything. Additionally, the user sometimes does not even need to remember a password to recover the data. On the other hand there is the secure solution that requires the user to remember the password, or the data may be lost and the user may need to manage the backup or the exported data.
An export is normally not done automatically, but started by the user to get the data from the service. In this process it is no problem to add a request to the user, if he wants to set a password or not and to add the encryption. A backup is normally something that should be done often and thus automatically without user interaction, when possible in the background.
In general, the user should have the possibility to add an additional passphrase and to select the location, where the data should be saved. In my opinion it is a bad trend in the industry to remove more and more options from the user just to make the interface less loaded. I find it important that the user keeps the option to decide, even if it is in a hidden part of the settings that may be harder to find. People who are interested in their data security will look for those options. Regardless of the possible settings, the more secure solution should be enabled by default, if it is possible without bigger usability losses.

3 Concept

3.1 Export-feature integration into the existing application

3.1.1 Different options

The goal of the Bachelor Thesis is the integration of the export feature into the existing Android application. There would be a lot of possible options on how to enable the user to do the export. Some of the better options would use the existing menus. There are two menus that would make sense to use for an export feature. One of them is the menu at the top of the conversation where currently the audio and video call menu items are located. This menu is easily accessible for every conversation. The other option would be to add the menu to the existing settings. These are accessible when clicking at the conversation title. Then at the bottom there is a menu where it is already possible to mute, archive,... the conversation. The problem with this menu is that it is only available for group conversations and would need to be added to the 1-on-1 conversations. This is why I chose to add the export menu item to the menu at the top of the conversation.
Another possibility that may make sense to use in the future would be to make the export accessible via the user settings. In this case, a full settings page would be added, where not only the current conversation would be exportable, but a selection of multiple conversations could be added. The export structure and the complete code are already designed to enable multiple conversations to be exported at once. So it would make sense to move the export to the global settings and to not locate it in a single conversation.

3.1.2 Description of the used option

As already mentioned, I decided to add a menu item to the conversation that enables the user to export the currently selected conversation. The result of the menu additions for navigation to the export fragment can be seen in figure 3.1.

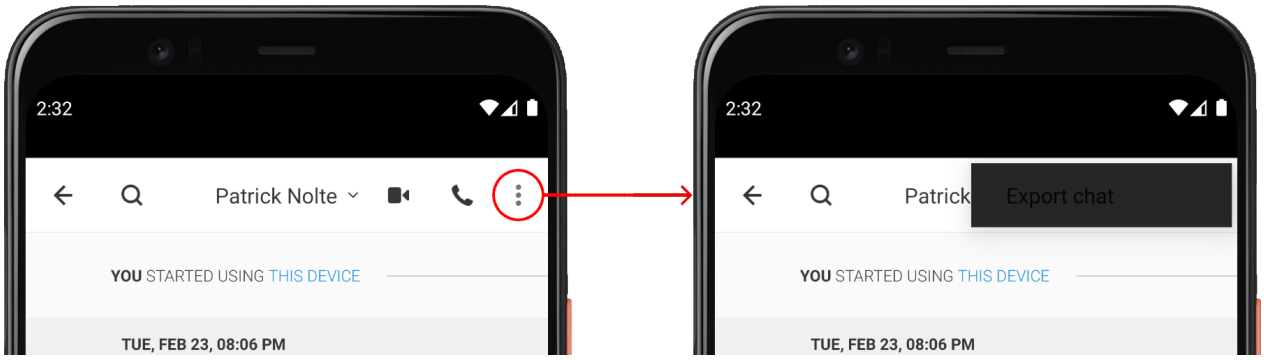
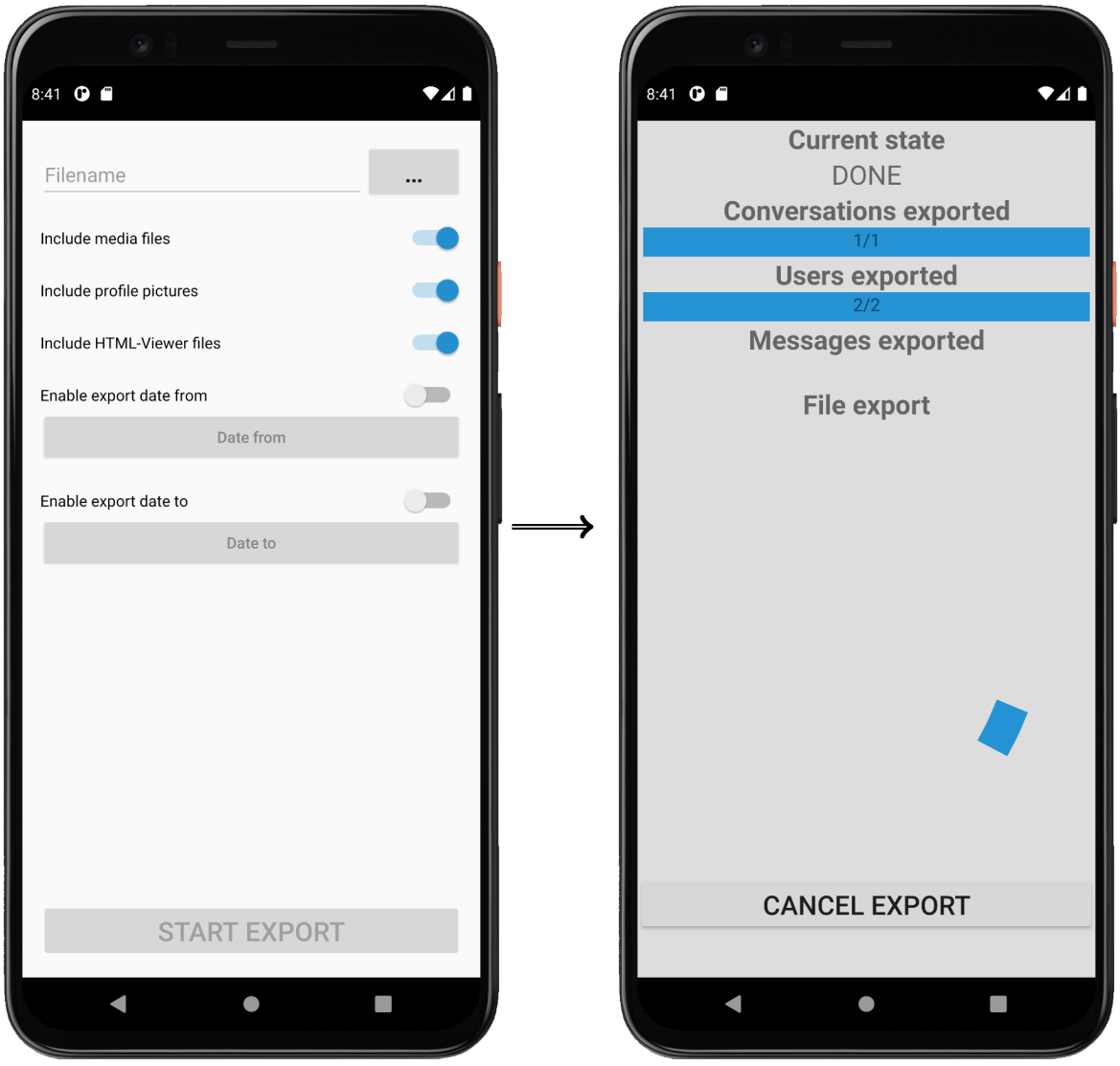


Figure 3.1: Conversation menu additions

3.1.3 Export functionality

After clicking the **Export chat** menu item the app navigates to the export configuration fragment. The export configuration fragment enables the user to configure the export. Next to the selection of the location the export should be saved to, it is possible to toggle the addition of the media files. Another configuration option is the possibility to select a time frame that should be exported. Additionally, the *HTML-Viewer* can be added to the export file to create the possibility to view the exported chat with a nice- looking representation without other tools. The export configuration screen can be seen in figure 3.2 together with the screen for a running export.



a) Export configuration **b)** Running export

Figure 3.2: Export screens

3.2 Structure of the export

The export is done by creation of a ZIP file containing all data selected in the export configuration screen, because it is easier to move one file instead of complete folders with a lot of files inside. Additionally, it is possible to add the option for encryption of the ZIP file later to secure the export. ZIP is the first format that comes to mind when looking for the ability to compress multiple files into one and to make it accessible on most platforms.
The ZIP file always contains a data folder with an XML document named **chats.xml** and optionally may contain a **media** and **profileData** folder. The **chats.xml** file contains all selected conversations with the corresponding information and their messages converted into XML elements. Additionally, all users that are included in the conversation and their information are converted to XML elements and inserted as well. XML was choosen because it provides the possibility to save the data in a structured way that is also easily human readable and has the possibility to describe the structure clearly in other formats like XSD. JSON is more compressed, but it is also harder to read for humans and in XML it is easy to define new types that are used a lot for the export. This makes the XML structure better defined and thus more secure to use.

If media files are exported, they will be saved inside the **media** folder. This includes not only audio, video and images, but also all other documents or files in general that may have been sent in a message. The profile pictures of all exported users are saved inside the **profileData** folder.
If the *HTML-Viewer* was selected for export, the HTML file **HTML-Viewer.html** is copied into the root directory of the ZIP file. It is the entry point when the display of the chat in a more beautiful representation inside a browser is desired. The *HTML-Viewer* needs some additional files that are saved inside the **websiteData** directory. This includes the CSS styles as well as some JavaScript files used for processing the XML document. Overall the structure of the ZIP file looks like described in figure 3.3.

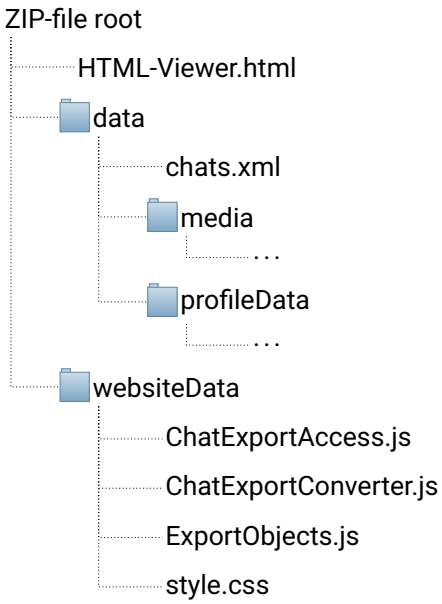


Figure 3.3: ZIP file structure

3.3 Module relationships

There is a number of files I needed to modify to implement the export feature. They needed just small changes to be able to show the button that opens my own fragments. These files were the **ConversationManagerFragment.scala**, **ConversationFragment.scala**, **WireApplication.scala** and the **strings.xml**. In the *ConversationFragment* the export button got added to the menu. The *ConversationManagerFragment* is responsible to open the export configuration, if the corresponding event was triggered in the export controller. The *WireApplication* had to be modified to enable the injection of the export controller. Lastly the **strings.xml** was edited to include all new texts needed by the changes.
Then we have some files I needed to create. At first there are some resources. One of the added files is the **conversation_header_menu_export.xml** that contains only the button displayed in the *ConversationFragment*. Then there are the **fragment_export.xml** and the **fragment_export_configuration.xml** containing the layout that is used by the **ExportFragment.scala** and the **ExportConfigurationFragment.scala**. They contain the configuration screen seen in figure 3.2a. To connect the design with the functionality there is the **ExportController.scala** that is used by the *ExportConfigurationFragment* to trigger the export. The *ExportController* then uses the **ExportConverter.scala** to initialize the export. The *ExportConverter* then starts to read all information about the selected conversations and the involved users together with all media files. To gather the information, it uses the *ExportController* to access the other injectable objects and their data. Everything the *ExportConverter* finds is then added to the defined ZIP file with the help of the **ExportZip.scala** file. The *ExportConverter* is also responsible to add the files included in the **export_html_viewer.zip** resource to the ZIP file.
The **export_chat_xml_schema.xsd** is only used during development to make sure, that the *ExportConverter* and the *HTML-Viewer* have the same understanding of the exported **chats.xml** document and to enable others to know the structure of the XML to process the document by themself.

With the *ExportConverter* using the objects provided by the *ExportController* the complete server interaction is handled by accessing the objects made accessible by the controller and request further objects there. This means that it was not necessary to write an own server communication, as the existing structures could be used. This also includes the files exchanged in the conversation where the same rules that are true for the conversation do also apply for the export. If the file is not already saved at the device, it will be downloaded from the server. If this is not done, it cannot be exported. To make sure to export all files, they should already be loaded on the device the export is done from.

3.4 HTML-Viewer

I created the *HTML-Viewer* to provide a tool to view the exported data. I decided for a web technology based solution to be platform independent. Nearly everyone has a browser already installed. Additionally, everything needed for the website is relatively small with roughly 100KB or rather 25KB when compressed in the ZIP.
The design of the website is inspired by "Wire for web", the web client of Wire, and thus looks nearly the same. I added some buttons I needed like the light-/darkmode switch and removed others that were no longer needed like the settings button.
The code used to create the design is completely new because the "Wire for web" website is developed to be used with their backend and is designed to receive messages and much more. This is not needed for the export feature. This is why I could not simply reuse their website. Additionally, I created the website without additional libraries to keep it small and created my own small backend that is only used to parse and process the **chats.xml**.

When the *HTML-Viewer* was included in the export, it can be looked at by opening the **HTML-Viewer.html** file. The first action that must be done is to click at the button that can be seen in figure 3.4 to select the **chats.xml**.

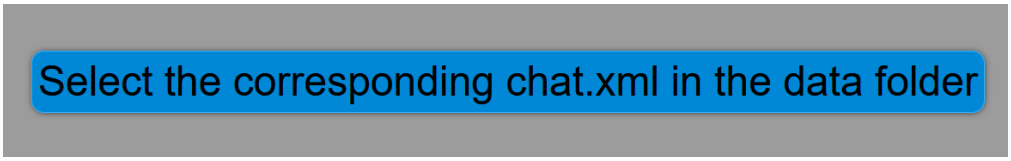


Figure 3.4: XML document selection

Then the XML document is loaded and the content is parsed. Now on the left side is a navigation, visible in figure 3.5, to select the conversation and to read the messages at the right side. Additionally, there is a button at the bottom left to switch to the user list to get a list of all contained users. The users can be selected to get more information about them as seen in 3.6. If a conversation is opened then there is an information icon visible at the top right. When clicking, more detailed information of the conversation is shown, including the members of the conversation and their roles visible in figure 3.7.

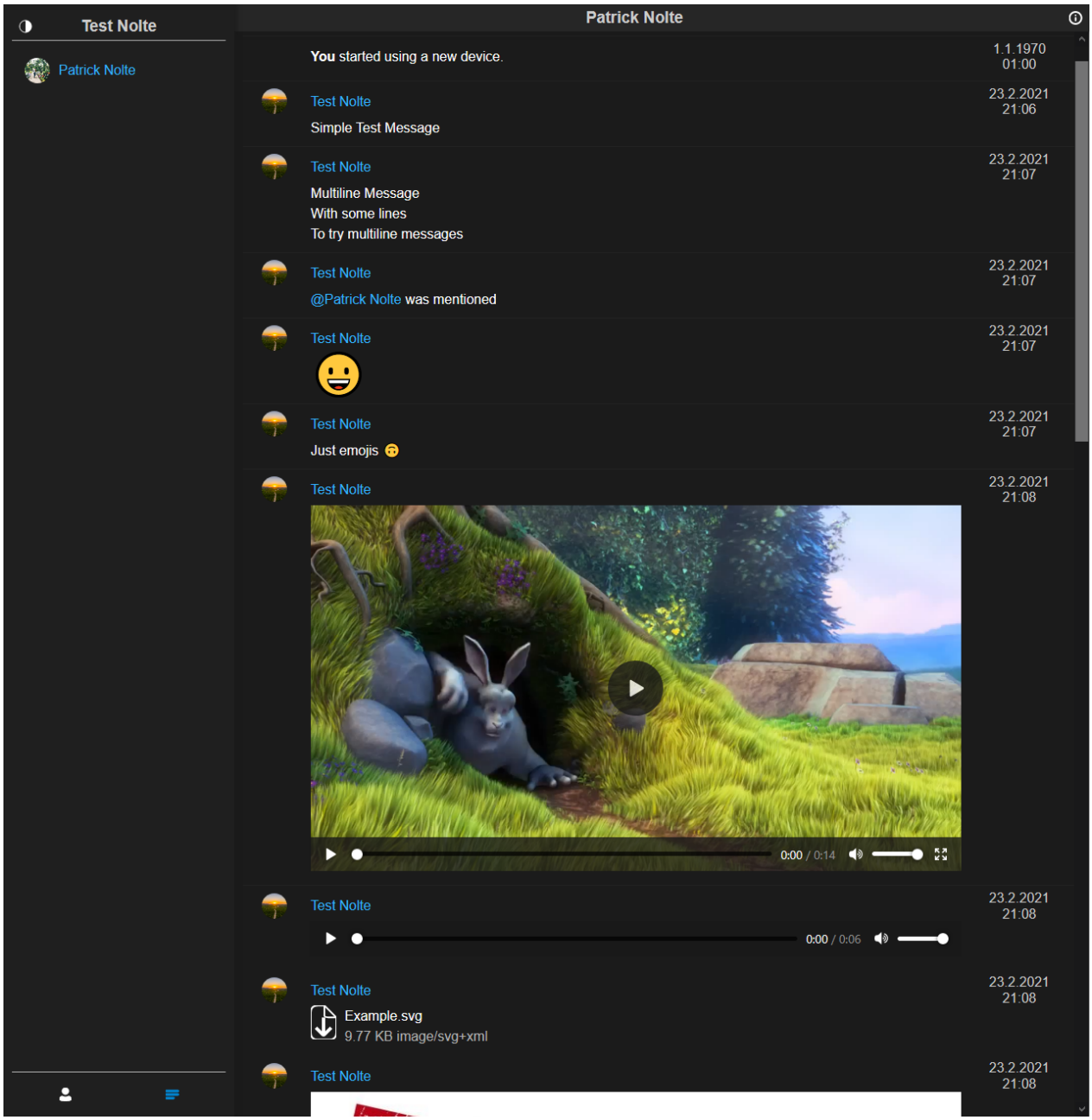


Figure 3.5: Website conversation

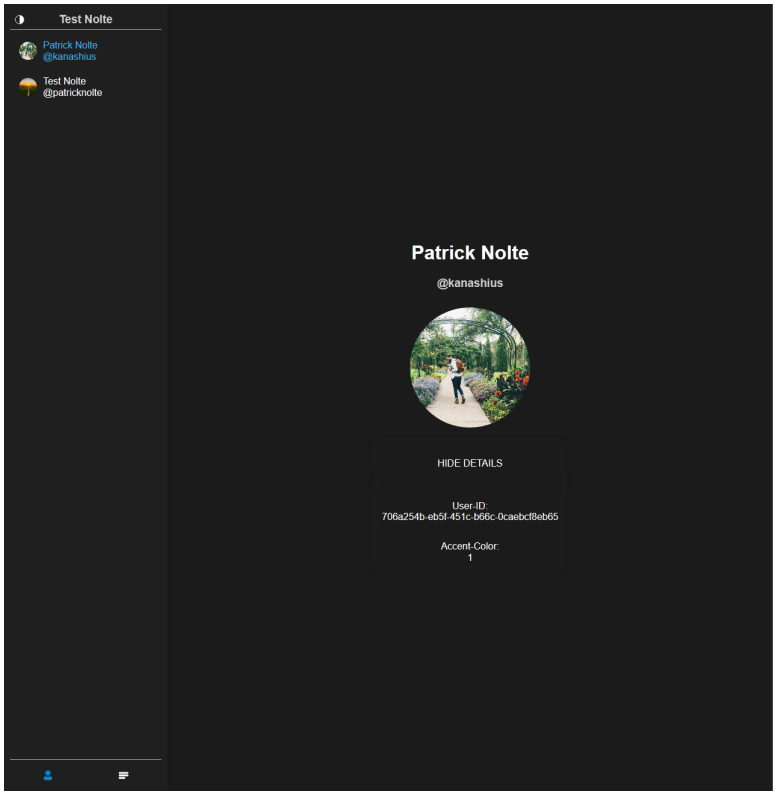


Figure 3.6: User information

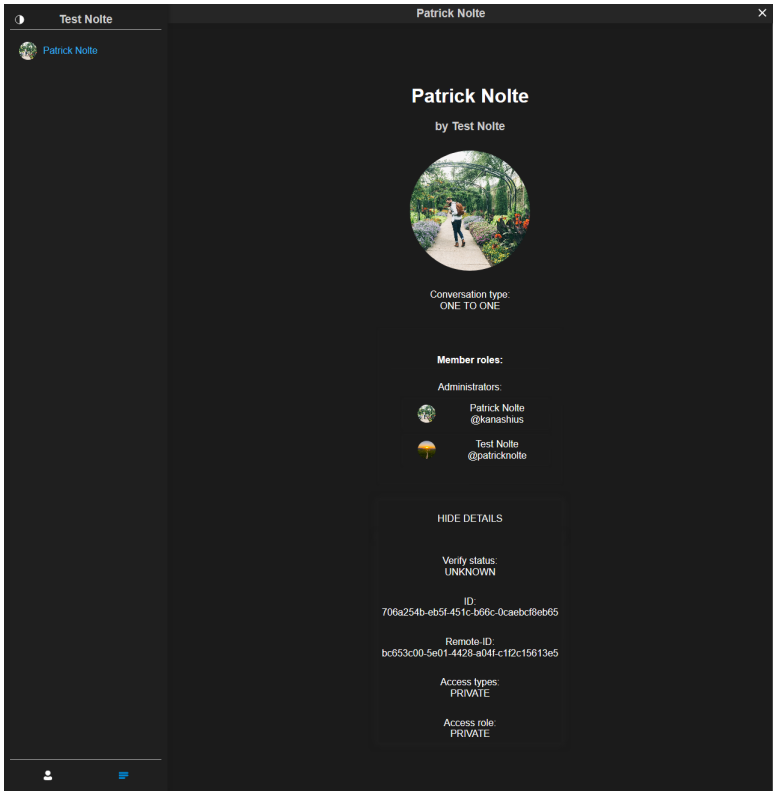


Figure 3.7: Website conversation information

4 Implementation details

4.1 Android-client changes

The first additions needed to be added into the *ConversationFragment* to add the menu item for navigation to the export configuration. To achieve this it needs the *ExportController* available by requesting it by injection in line 112. Then the export menu item is added to the existing menu

```
112  private lazy val exportController = inject[ExportController]
```

Figure 4.1: ConversationFragment - line 112 to 112

containing the video- and voice call items as can be seen in line 226 to 228. When the button is

```
226      toolbar.getMenu.clear()
227      id.foreach(toolbar.inflateMenu)
228      toolbar.inflateMenu(R.menu.conversation_header_menu_export)
```

Figure 4.2: ConversationFragment - line 226 to 228

there, the click event is needed. This is done as we can see in line 363 to 378. The signal **onShowExport** is called at the *ExportController* in line 372 and if the user is currently typing a message, the typing is interrupted and the keyboard hidden (line 373/374). When the **onShowExport** is triggered, the *ConversationManagerFragment* that is already listening at the **Signal** receives the event and opens the *ExportFragment* as seen in line 142 to 145. To access the *ExportController* it is also requested by injection (line 78). To enable the *ConversationFragment* and the *ConversationManagerFragment* to request the injected *ExportController*, the *WireApplication* needs to initialize and bind the *ExportController* as seen in line 316. The *ExportFragment* is a container for the *ExportConfigurationFragment*. That is why the *ExportConfigurationFragment* tag is provided as a parameter for the new *ExportFragment* instance. Of course the Android way to define strings in the **strings.xml** is followed and that is why there are some strings added, that will be needed at various places as seen in figure 4.7 and 4.8.

```
363   toolbar.setOnMenuItemClickListener(new Toolbar.OnMenuItemClickListener() {
364     override def onMenuItemClick(item: MenuItem): Boolean =
365       item.getItemId match {
366         case R.id.action_audio_call | R.id.action_video_call =>
367           callStartController.startCallInCurrentConv(withVideo = item.getItemId
                ↪  == R.id.action_video_call, forceOption = true)
368           cursorView.foreach(_.closeEditMessage(false))
369           true
370         case R.id.action_export_chat =>
371           //TODO implement
372           exportController.onShowExport ! None
373           cursorView.foreach(_.closeEditMessage(false))
374           keyboardController.hideKeyboardIfVisible()
375           true
376         case _ => false
377       }
378   })
```

Figure 4.3: ConversationFragment - line 363 to 378

```
142   subs += exportController.onShowExport.onUi { p =>
143     showFragment(ExportFragment.newInstance(Some(ExportConfigurationFragment.
          ↪ Tag)), ExportFragment.TAG)
144   }
```

Figure 4.4: ConversationManagerFragment - line 142 to 145

```
78  private lazy val exportController = inject[ExportController]
```

Figure 4.5: ConversationManagerFragment - line 78 to 78

```
316   bind [ExportController]    to new ExportController()
```

Figure 4.6: WireApplication - line 316 to 316

```
899   <string name="conversation_toolbar__export_chat">Export chat</string>
```

Figure 4.7: Strings - line 899 to 899

```
1257    <string name="start_export">Start export</string>
1258    <string name="export_file_not_set">The export file is not defined</string>
1259    <string name="export_done">Export done</string>
1260    <string name="include_media_files">Include media files</string>
1261    <string name="include_html_files">Include HTML-Viewer files</string>
1262    <string name="enable_export_date_from">Enable export date from</string>
1263    <string name="date_from">Date from</string>
1264    <string name="enable_export_date_to">Enable export date to</string>
1265    <string name="date_to">Date to</string>
1266    <string name="dots">...</string>
1267    <string name="filename">Filename</string>
```

Figure 4.8: Strings - line 1257 to 1267

4.2 Android-client additions

4.2.1 Layout and graphic interface

At first some layouts were needed to contain the basic structure of the fragments. One of them was the menu item added to the *ConversationFragment*. It is a simple text item visible in figure 4.9. Then

```
6   <item
7       android:id="@+id/action_export_chat"
8       android:orderInCategory="300"
9       android:title="@string/conversation_toolbar__export_chat"
10      app:showAsAction="never" />
```

Figure 4.9: Conversation header menu for export - line 6 to 10

we have the *fragment export layout* being a simple container for the *fragment export configuration layout* containing a *FrameLayout* within a *LinearLayout*, similar to the *fragment participant layout*. At the moment it would not be needed, but maybe the export will be expanded and may contain more fragments then only the configuration. The *fragment export configuration layout* is designed to look like shown in figure 3.2a. It contains a *RelativeLayout* with a *Scrollview* containing the configuration options and the export button at the bottom. Additionally, there is a hidden *LinearLayout* inside the *RelativeLayout* containing the indicator for the running export that is made visible if the export is running. The configuration options consist of a vertically orientated *LinearLayout* with some basic controls like buttons and switches as well as the *TextField* and *Button* for selecting the export file inside a horizontal *LinearLayout*. The **start export** button is placed at the bottom of the *Scrollview* by wrapping it inside another *LinearLayout* with the **android:gravity** set to **center|bottom**. This can be seen in figure 4.10.

```
104         <LinearLayout
105             android:layout_width="match_parent"
106             android:layout_height="0dp"
107             android:layout_weight="1"
108             android:gravity="center|bottom"
109             android:orientation="vertical">
110             <Button
111                 android:id="@+id/b__export_start"
112                 android:layout_width="match_parent"
113                 android:layout_height="wrap_content"
114                 android:gravity="center"
115                 android:text="@string/start_export"
116                 android:textSize="@dimen/wire__text_size__medium" />
117         </LinearLayout>
```

Figure 4.10: Export configuration fragment layout - line 104 to 117

The described layouts are then loaded in the corresponding *ExportFragment* and *ExportConfigurationFragment*. The *ExportFragment* loads the *fragment export layout* and replaces the export container with the child fragment that should be loaded. At the moment there is only the *ExportConfigurationFragment* that can be loaded into the container. This implementation was created with the example of the *ParticipantFragment* in mind, where multiple fragments could be displayed, to keep the option to add other fragments later with just some small changes. The resulting code can be seen in figure 4.11.

```
21  override def onViewCreated(view: View, @Nullable savedInstanceState: Bundle)
        ↪ : Unit = {
22    verbose(l"onViewCreated.")
23    withChildFragmentOpt(R.id.fl__export__container) {
24      case Some(_) => //no action to take, view was already set
25      case _ =>
26        (getStringArg(PageToOpenArg) match {
27          case Some(ExportConfigurationFragment.Tag) =>
28            Future.successful((new ExportConfigurationFragment,
                ↪ ExportConfigurationFragment.Tag))
29          case _ =>
30            Future.successful((new ExportConfigurationFragment,
                ↪ ExportConfigurationFragment.Tag))
31        }).map {
32          case (f, tag) =>
33            getChildFragmentManager.beginTransaction
34              .replace(R.id.fl__export__container, f, tag)
35              .addToBackStack(tag)
36              .commit
37        }
38    }
39  }
```

Figure 4.11: ExportFragment - line 21 to 39

The *ExportConfigurationFragment* initializes the controls with the values currently set in the *ExportController* and adds the listeners for all controls. One example can be seen at line 42 to 54 where the **start export** button gets the listener for the **onClick** event added. When the button is clicked, the indicator showing a currently running export is activated. Then the export is started. At the end of the export, the provided callback is called, that runs on the current UI thread and hides the indicator again.
Another functionality is the file selection if the **...** button is clicked. To create the file, in Android 11, it is required to use the **ACTION_CREATE_DOCUMENT Intent** provided by Android to enable the user to select the file. This ensures that the application only gets access to the files, the user wants to grant access to. If the result of the intent is received, the **exportFile** field in the *ExportController* is updated accordingly and the **start export** button can be clicked now. Another interesting feature is the call of the *DatePickerDialog* and in case of a successful selection the call of the *TimePickerDialog* to select the dates of the export date range that can be enabled. In the default

```
42    exportButton.setOnClickListener(new View.OnClickListener {
43      override def onClick(v: View): Unit = {
44        exportLoadingIndicator.setVisibility(View.VISIBLE)
45        exportController.‘export‘(()=>{
46          getContext.asInstanceOf[Activity].runOnUiThread(new Runnable {
47            override def run(): Unit = {
48              exportLoadingIndicator.setVisibility(View.GONE)
49              Toast.makeText(getContext,WireApplication.APP_INSTANCE.
                    ↪ getApplicationContext.getString(R.string.export_done),Toast.
                    ↪ LENGTH_LONG)
50            }
51          })
52        })
53      }
54    })
```

Figure 4.12: ExportConfigurationFragment - line 42 to 54

case, the start and end of the date range are disabled so that the entire conversation is exported. The start and end date can be enabled separately to limit the start and/or end of the export. When one of the date buttons is clicked, the **showDateTimePicker** method seen in figure 4.13 is called with the corresponding date value currently selected. At the end of a successful selection, the callback is called and the selected date is delivered. The callback in figure 4.14 is responsible to update the correct date at the *ExportController* and the date shown at the button. The *ExportConfigurationFragment* holds own fields for the selected date so that the fields of the *ExportController* are only updated, if the corresponding start/end date switch is enabled.
Finally there is the *ExportController* as an interface between the graphical interface and the back end. It provides the fields that are set by the graphical interface and read by the *ExportConverter* that can be seen in figure 4.15. Additionally, it provides the objects needed by the *ExportConverter* to access the data of the app. This includes the messages and the objects they are defined with, as well as the media files that are called assets. Lastly it has an export function that requests the current conversation, creates the *ExportConverter* and starts the export of that conversation.

```
128  def showDateTimePicker(time: Option[RemoteInstant], callback: RemoteInstant
         ↪ => Unit): Unit = {
129    val currentDate = Calendar.getInstance
130    val date = Calendar.getInstance
131    if(time.nonEmpty)
132      date.setTimeInMillis(time.get.toEpochMilli)
133    new DatePickerDialog(getContext, new DatePickerDialog.OnDateSetListener() {
134      override def onDateSet(view: DatePicker, year: Int, monthOfYear: Int,
           ↪ dayOfMonth: Int): Unit = {
135        date.set(year, monthOfYear, dayOfMonth)
136        new TimePickerDialog(getContext, new TimePickerDialog.OnTimeSetListener
             ↪ () {
137          def onTimeSet(view: TimePicker, hourOfDay: Int, minute: Int): Unit = {
138            date.set(Calendar.HOUR_OF_DAY, hourOfDay)
139            date.set(Calendar.MINUTE, minute)
140            callback(RemoteInstant.ofEpochMilli(date.getTimeInMillis))
141          }
142        }, currentDate.get(Calendar.HOUR_OF_DAY), currentDate.get(Calendar.
             ↪ MINUTE), false).show()
143      }
144    }, currentDate.get(Calendar.YEAR), currentDate.get(Calendar.MONTH),
         ↪ currentDate.get(Calendar.DATE)).show()
145  }
```

Figure 4.13: ExportConfigurationFragment - line 128 to 145

```
100    dateToInput.setOnClickListener(new OnClickListener {
101      override def onClick(v: View): Unit = {
102        showDateTimePicker(dateTo,r=>{
103          dateTo=Some(r)
104          val date = new Date()
105          date.setTime(r.toEpochMilli)
106          val format=DateFormat.getDateTimeInstance(DateFormat.SHORT,DateFormat.
               ↪ SHORT)
107          dateToInput.setText(format.format(date))
108          if(exportLimitToSwitch.isChecked)
109            exportController.timeTo=dateTo
110        })
111      }
112    })
```

Figure 4.14: ExportConfigurationFragment - line 100 to 112

```
23  var exportFile: Option[Uri] = None
24  var timeFrom: Option[RemoteInstant] = None
25  var timeTo: Option[RemoteInstant] = None
26  var exportFiles = true
27  var includeHtml = true
```

Figure 4.15: ExportController - line 23 to 27

4.2.2 Export functionality

When the export is started by the *ExportController* the *ExportConverter* starts to convert the information into the XML document. At first the *ExportConverter* creates the *ExportZip* where all files are added as well as the *Document* where all XML information is saved. Then all conversation ids (currently just one, because there is no conversation selection) provided as a parameter are iterated and the corresponding XML elements are created and added to the document. When iterating the conversation, all conversation members are added to a list. This list is written to the XML document when the conversations are processed. At the end the XML document is saved to the ZIP file as well as the *HTML-Viewer* files if they were selected for export. The media files are saved to the ZIP file while processing the conversations and the profile pictures are added when the users are processed. The conversion of the users can be seen in figure 4.16 as an example to get an overview how the elements are created and added to the XML.
Something similar is done with the conversations as well. At first all available information of the conversation, including the member list of users with their roles (line 138 to 145), are added to the XML document.
Then all messages in the provided time period are retrieved from the *MessagesStorage* accessible with the *ZMessaging* object that is saved as a field in the *ExportController* and then sorted by date and processed to create message elements that are then added to the XML document. The message elements contain basic information, like the user sending the message and the message type. The message type is needed to distinguish different messages and display them correctly later. Depending of the message type, different other elements are added to the message element. This includes simple text content as well as assets like images. If an asset is to be saved, first the *AssetServiceImpl*, that is accessible by the *ZMessaging* object, is used to check if the asset is available and saved to the ZIP if possible (line 520 to 550). On success the filename is saved into the corresponding XML element, otherwise the asset id is saved. Because sometimes the file extension is missing, it is retrieved by the *AssetServiceImpl* as well.
The *ExportZip* is a helper class to manage the ZIP file and add all files to the export ZIP. It is synchronized to add the possibility to make the export multithreaded at a later point in time. Currently the export is done in succession to avoid conflicts especially when elements are added to the XML *Document* object.

```
77    userList.toList.distinct.map(u=>exportController.usersController.user(u).future).map(f=>Await.ready(f,Duration.Inf)).map(f=>{
78      f.map(ud=>{
79        val user=addElement(users, "user")
80        if(selfId.nonEmpty && selfId.get.equals(ud.id)) user.setAttribute("isSelf","true")
81        addElement(user,"userid",ud.id.str)
82        ud.handle.foreach(h=>addElement(user,"username",h.string))
83        ud.teamId.foreach(tid=>addElement(user,"teamid",tid.str))
84        addElement(user,"name",ud.name.str)
85        ud.email.foreach(em=>addElement(user,"email",em.str))
86        ud.phone.foreach(p=>addElement(user,"phone",p.str))
87        ud.trackingId.foreach(tid=>addElement(user,"trackingid",tid.str))
88        ud.picture.foreach(p=>{
89          (p match {
90            case p: PictureUploaded => saveAssetIdAndGetFilename(p.id, profilePath).orElse(Some(p.id.str))
91            case p: PictureNotUploaded => saveAssetIdAndGetFilename(AssetId.apply(p.id.str), profilePath).orElse(Some(p.id.str))
92            case _ => None
93          }).foreach(path=>{
94            val pic=addElement(user,"picture", path)
95            addAttribute(pic,"uploaded",p.isInstanceOf[PictureUploaded].toString)
96          })
97        })
98        addElement(user,"accent_color",ud.accent.toString)
99        if(ud.fields.nonEmpty) addElement(user,"userfields",ud.fields.toString)
100       if(ud.permissions._1!=0 || ud.permissions._2!=0) addElement(user,"permission",ud.permissions._1+" "+ud.permissions._2)
101     })
102   }).foreach(f=>Await.ready(f,Duration.Inf))
```

Figure 4.16: ExportConverter - line 77 to 102

```
138   val userroles=addElement(conversation,"userroles")
139   val userAddFut=exportController.convController.convMembers(convId).future.map(m=>m.foreach({case (uid,cr)=>
140     val userrole=addElement(userroles, "userrole")
141     addAttribute(userrole,"userid",uid.str)
142     addAttribute(userrole,"role",cr.label)
143     userList+=uid
144   }))
145   Await.ready(userAddFut, Duration.Inf)
```

Figure 4.17: ExportConverter - line 138 to 145

```
520  private def saveAssetIdAndGetFilename(assetId: AssetId, folder: String,
         ↪ filename: Option[String] = None, convId: Option[ConvId] = None):
         ↪ Option[String] = {
521    var path: Option[String]=None
522    val lockObject = new AtomicBoolean(false)
523    lockObject.synchronized {
524      val acc=(ai: AssetInput)=>{
525        try{
526        ai.toInputStream.foreach(is => {
527          path=Some(folder + filename.getOrElse(assetId.str+
               ↪ assetIdGetFileExtension(assetId).map(a=>"."+a).getOrElse("")))
528          zip.writeFile(path.get,is)
529          verbose(l"SUCCESS LOADING FILE : ${showString(path.get)}")
530        })
531        }catch{
532          case e: Throwable => verbose(l"EXPORT - ERROR: ${showString(e.toString
               ↪ )}")
533        }finally{
534          lockObject.synchronized {
535            lockObject.notifyAll()
536          }
537        }
538      }
539      zmsg.assetService.loadContentById(assetId, None).map(acc).onFailure({case
           ↪  e=>
540        zmsg.assetService.loadPublicContentById(assetId,convId).map(acc).
             ↪ onFailure({case e2=>
541          verbose(l"FAILURE LOADING FILE : ${showString(e.toString)} AND ${
               ↪ showString(e2.toString)}")
542          lockObject.synchronized {
543            lockObject.notifyAll()
544          }
545        })
546      })
547      lockObject.wait()
548    }
549    path
550  }
```

Figure 4.18: ExportConverter - line 520 to 550

4.2.3 Export format definition

To ensure that the *ExportConverter* uses the same format for the XML document as the *HTML-Viewer* uses for parsing, an **export_chat_xml_schema.xsd** was created. In this file the correct structure of the **chats.xml** document is defined. This structure is strongly based on the objects used in the Wire app and contains basically any information that may be relevant for reconstructing and displaying the conversation. An example for such a definition is the conversation element that can be seen in figure 4.19. It is based on the *ConversationData* object and contains some information that is always available like the **id** of the message, the **remoteid**, the id of the creator named **creator**, the conversation type (group/1-on-1/...) **convType** and the **verified** field. Additionally, there may be optional data like the **name**, **teamid**, **access**, **accessrole** and the **link**. When all values available at the object are included, some additional data may be needed. In this case this includes a list of users participating in the conversation that is saved in the **userroles** list. The list contains all user ids with their role (member/admin/...) in the conversation. Lastly a conversation without messages is rather meaningless thus the messages of the conversation are received and added, too.

```
14    <xs:complexType name="conversationType">
15       <xs:all>
16          <!-- required values -->
17          <xs:element name="id" type="uuid"/>
18          <xs:element name="remoteid" type="uuid"/>
19          <xs:element name="creator" type="uuid"/>
20          <xs:element name="convType" type="xs:string"/> <!-- maybe type
               ↪ conversation_type_type -->
21          <xs:element name="verified" type="xs:string"/> <!-- maybe type
               ↪ verified_type -->
22
23          <!-- Sequences -->
24          <xs:element name="userroles" type="userrolesListType"/>
25          <xs:element name="messages" type="messageListType"/>
26
27          <!-- optional values -->
28          <xs:element name="name" type="xs:string" minOccurs="0"/>
29          <xs:element name="teamid" type="uuid" minOccurs="0"/>
30          <xs:element name="access" type="accessListType" minOccurs="0"/>
31          <xs:element name="accessrole" type="xs:string" minOccurs="0"/> <!--
               ↪ maybe type accessrole_type -->
32          <xs:element name="link" type="xs:anyURI" minOccurs="0"/>
33       </xs:all>
34    </xs:complexType>
```

Figure 4.19: Export chat XML schema (XSD) - line 14 to 34

4.2.4 Comparison of the export format and other existing solutions

I already mentioned some types of export that are used in section 2.1. They do not have an official documentation easily available, so I could not check it without exporting the data myself. Some of them only export the text messages. This is mostly done with simple text documents containing all messages line by line with a date, the sender and the text message, similar to the example "<19.03.2021 22:14:00> Patrick Nolte: This is an example message." without any other structure. In WhatsApp for example were all media file paths just a path inside of the text data. This is no definite way to export messages because any path typed inside a text message may be parsed as a file. Another type of export was the approach of Telegram and the Facebook HTML export. There the data is converted into HTML and this HTML file can be opened in a browser to view the exported conversation. This is one approach I also thought of, but decided against, because it may be good for viewing the exported data, but it makes further processing much more difficult. Importing into other applications is much easier when using a defined structure.

Comparison to the Matrix protocol

To see if others may implement an export in a similar way, I would like to compare my export structure to the Matrix protocol, that was already mentioned when the messenger Elements was discussed in section 2.1.1. Taking a closer look at the specification of the Matrix Client-Server API [6] it can be compared to my approach. One difference is that Matrix has a lot of events that can be sent whereas my export only contains messages. The Matrix protocol has one event that is used to sent a message.
The first difference when looking at the event is the use of JSON. In my solution the size is not as critical as in an application that sends the data over the internet, especially if mobile data is involved, and thus I gave preference to the better structure of the XML files instead of the smaller size.
The structure of the message data is relatively similar. If we take the example of a message containing an image with my exported XML in figure 4.20a and the Matrix JSON data in figure 4.20b this can be clearly seen. Firstly, we see that both have a **creator**/**sender** of the message, a unique id for the message (**id**/**event_id**) and a time when the message was sent (**time**/**origin_server_ts**). My export has the **state** as additional field as well as the **local_time** and the **asset_id**. The **state** is used to show if the message was delivered. This is not included into the Matrix message because they have another event that is sent to achieve this. The **local_time** is used in Wire to be able to see, when the user sent the message locally, but Matrix uses only the time of the server. The **asset_id** is only added for completeness into my structure, because this way it can be identified at the Wire servers for downloading and that could be used for importing to other devices. The **msg_type** in my export structure is equal to the **msgtype** in the JSON data. In the JSON it is located at the content, because the **type** is already used to differentiate other events in the Matrix protocol.

Thus **type** is one of the additional fields of the JSON data, next to the **room_id**, that is used to identify the room (that is similar to a Wire conversation) the message comes from as well as the **unsigned** used to save additional data of the event.
The last element both structures share is the **protos**/**content** element. It contains the content of the message(s). Because Wire supports sending multiple images at the same time, this is a list in my structure, whereas Matrix splits them into multiple messages. Both have a filename saved in the **name**/**body** elements, as well as a **height**/**h**, **width**/**w**, **mime_type**/**mimetype** and **size**/**size**. The location of the image itself is saved as **filepath** in my export, because it describes the path inside the ZIP file, whereas it is an **URL** in the Matrix protocol because it is made available at the given URL at the server.
The same is true for the other message types. They are all very similar because they just hold the data for the messages that is needed.

```xml
<message>
  <id>ab685d47-a112-4271-bf5c-9172
      ↪ eaf452e3</id>
  <msg_type>IMAGE_ASSET</msg_type>
  <userid>4bfb16b5-72a2-4f0c-a9e3-
      ↪ b544045d2880</userid>
  <state>DELIVERED</state>
  <time>2021-02-23T20:10:07.420Z</
      ↪ time>
  <local_time>2021-02-23T20
      ↪ :10:08.223Z</local_time>
  <asset_id>3-5-5497d43a-5a58-40af
      ↪ -8036-b56d6845a883</
      ↪ asset_id>
  <protos>
    <proto>
      <asset>
        <mime_type>image/gif</
            ↪ mime_type>
        <name>5ym8IM6F90wd0mIeyd.gif<
            ↪ /name>
        <size>769376</size>
        <metadata type="IMAGE">
          <image>
          <width>480</width>
          <height>270</height>
          <tag/>
          </image>
        </metadata>
        <filepath>data/media/5
            ↪ ym8IM6F90wd0mIeyd.gif</
            ↪ filepath>
      </asset>
    </proto>
  </protos>
</message>
```

a) Exported message of type image - XML

```json
{
  "content": {
    "body": "filename.jpg",
    "info": {
      "h": 398,
      "mimetype": "image/jpeg",
      "size": 31037,
      "w": 394
    },
    "msgtype": "m.image",
    "url": "mxc://example.org/
        ↪ JWEIFJgwEIhweiWJE"
  },
  "event_id": "$143273582443PhrSn:
      ↪ example.org",
  "origin_server_ts":
      ↪ 1432735824653,
  "room_id": "!jEsUZKDJdhlrceRyVU:
      ↪ example.org",
  "sender": "@example:example.org",
  "type": "m.room.message",
  "unsigned": {
    "age": 1234
  }
}
```

b) Matrix message of type image - JSON [6]

4.2.5 Common base between Matrix and Wire

Most of the fields that would be good to have in an export are available with Matrix as well as Wire. If we look at the rooms (Matrix) or conversations (Wire), then they both have most of the fields. Matrix has some special fields like the **topic**, **pinned messages** and **tags** for the rooms, whereas Wire supports teams in the commercial area and thus has the **teamid**, the **accessrole** (first role after joining) and an invite **link**, if available.
The next part would be the user information. There, both support a **member_id**, an **avatar**, a **display name**, the current state/role (**membership**) in the conversation and the possibility to recognize the user creating the export (**isSelf**). Matrix has an additional field showing the user that provided the invite to the conversation, whereas Wire has other additional fields. Some of them like the more general ones, like the **username**, the **email** and the **phone**number, but some are more specific for Wire like the **teamid**, the **trackingid**, the **userfields** and the **permissions** for the user.
The most important fields for messages are supported by Matrix and Wire. These are the **type** of the message, an unique **id**, the **sender** and the **send_time** of the message as well as the **content**. The **state** is better defined in Matrix, because it shows for each user, if the message was read and when. Wire only has a general field for the complete message to show if the message was sent, delivered, read, ... Wire also provides additional fields to support quotes, mentions, expiring messages and more. Edited messages are supported by both again, but Wire writes the information to the message itself, whereas Matrix has a change event, that must be processed to add the information to the export.
When it comes to the **content** of the messages, both are pretty similar again, with just some small changes to the structure. Wire for example, has some additional fields regarding **mentions**, **rich_media** data and **open_graph** data. There are some content types both support like **text**, **image**s and **location**s. **Video**, **audio** and **files** are also supported by both, but Matrix has an own content type for them, whereas Wire combines them into the **asset** type. This also applies to the **emote** type, that is supported by Wire in the **text** type. Matrix has two types that Wire does not have, the first one is the **notice** type, that can be used by room administrators to sent server messages, the other one is the **sticker** type. Wire on the other hand supports **calling**, **knock**ing, **clear**ing the conversation, **clientaction**s, message **delet**ion, **edit**ing, and some other Wire specific content types.
To sum it up: The export of Matrix and Wire could be put together into one schema, because they do not have any conflicts in their information. But to support both it would make sense to remove some fields of each messenger, that are not really important to create an export and have a lot of the fields as optional fields that may add information, but are not required. Because the current XML schema I created to support Wire already has most of the fields included, it would be relatively easy to add the few fields it is missing to support the Matrix protocol. Some of the fields currently exported could also be removed, because they are not really necessary to provide a complete export, but would only be useful to import them back into Wire.

=>	:	Subfield of the previous parent field
Uppercase type	:	The type is specified in another table
x	:	The field is required
(x)	:	The field is optional
$[x]$	:	Used for some file and URL fields if only one of them is required
//comment//	:	Defines a comment with additional information
(asset)	:	Parentheses in the type specify it further
M	:	Short for Matrix
W	:	Short for Wire

Figure 4.21: Legend for field tables

Name	Type	Matrix	Wire
room_id	id	x	x
remote id	id		x
name	string	(x)	(x)
creator id	USER/id	x	x
topic	string	(x)	
is_direct/conv_type	string	(x)	x
verified	boolean		x
avatar	AVATAR	(x)	//maybe user picture//
pinned_events	list of ids	(x)	
tags	list of tags ordered	(x)	
=> tag	string	x	
=> order	double (0 to 1)	(x)	
members	list of USERS/their ids	(x)	(x)
messages	list of MESSAGES/their ids	(x)	(x)
teamid	id		(x)
access	string	(x)	(x)
accessrole	string		(x)
link	string		(x)

Table 4.1: Fields of conversations/rooms

Name	Type	Matrix	Wire
member_id	id	x	x
avatar	AVATAR	(x)	(x)
display_name	string	(x)	x
accent_color	int		x
membership	one of invite/join/knock/leave/ban	(x)	x //user role//
invited_by	string //display_name//	(x)	
username	string		(x)
teamid	id		(x)
email	string		(x)
phone	string		(x)
trackingid	id		(x)
userfields	string		(x)
permission	int int		(x)
isSelf	boolean	(x)	(x)

Table 4.2: Fields of users

Name	Type	Matrix	Wire
info	(see INFO for image)	x	
url	string	x	
filepath	string		x

Table 4.3: Fields of avatars

Name	Type	Matrix	Wire
type	string	x	x
id	string	x	x
sender	id	x	x
send_time	timestamp	x	x
unsigned	UNSIGNED_DATA	(x)	
state	sent/delivered/...	(x) //list//	x //single entry//
=> userid	id	(x)	
=> timestamp	ts	(x)	
local_time	timestamp		(x)
error			(x)
=> userid	id		x
=> errorcode	int		x
is_first_message	boolean		(x)
members	id list //mentions//		(x)
recipient	id		(x)
edit_time	timestamp		(x)
ephemeral			(x)
=> length	long		x
=> timeunit	time_unit		x
expiry_time	date		(x)
duration			(x)
=> length	long		x
=> timeunit	time_unit		x
asset_id	id		(x)
quote	id		(x)
=> validity	boolean		(x)
content	list of CONTENT	x	(x)

Table 4.4: Fields of messages

Name	Type	Matrix	Wire
age	int	x	
redacted_because	event	(x)	
transaction_id	string	(x)	

Table 4.5: Fields of unsigned data

Name	Type	Matrix	Wire
msgtype	string	x	x
body/content	string	x	x
mentions	list of ids		(x)
asset_id	id		(x)
width	int		(x)
height	int		(x)
rich_media	RICHMEDIA		(x)
open_graph	OPENGRAPH		(x)

Table 4.6: Fields of content for all types

Name	Type	Matrix	Wire
kind	string		x
provider	string		x
title	string		x
linkurl	string		x
artist			(x)
=> name	string		x
=> avatar	filename/id		(x)
duration	long		(x)
artwork	string (filename/id)		(x)
expires	timestamp		(x)
tracks	list of rich_media		(x)
streamable	boolean		(x)
streamurl	uri		(x)
previewurl	uri		(x)

Table 4.7: Fields of richmedia

Name	Type	Matrix	Wire
title	string		x
description	string		x
tpe	string		x
permanenturl	uri		(x)
image	uri		(x)

Table 4.8: Fields of opengraph

Name	Type	text		emote		notice		image		file		audio		location	
		M	W	M	W	M	W	M	W	M	W	M	W	M	W
format	string	(x)		(x)		(x)									
formatted_body	string	(x)	x	(x)		(x)									
info/metadata	INFO/ METADATA							(x)	x	(x)		(x)		(x)	
url	string							[x]		[x]		[x]			
file	file (encrypted)/ path							[x]		[x]		[x]			
filename	string									(x)					
expiremillis	long		(x)						(x)						(x)
geo_uri	string													x	
quote	id		(x)												
mentions	list (start, length, id)		(x)												
linkpreviews	list of LINKPREVIEW		(x)												
latitude	float														x
longitude	float														x
zoom	int														x
name	string														x

Table 4.9: Fields of content by type (part 1)

Name	Type	calling		cleared		clientaction		deleted		edited		external		hidden	
		M	W	M	W	M	W	M	W	M	W	M	W	M	W
data	string/id		x						x						x
timestamp	long				x										
clientActionType	string						x								
replacingMessageId	id										x				
text/composite	CONTENT (text/composite)										[x]				
encryption	int												x		
otrKey	string												x		
sha256	string												x		

Table 4.10: Fields of content by type (part 2)

Name	Type	video		sticker		asset		knock		lastread	
		M	W	M	W	M	W	M	W	M	W
info/metadata	INFO/ METADATA	(x)		x			x				
url	string	[x]		x							
file	file (encrypted)/ path	[x]					(x)				
filename	string						x				
expiremillis	long						(x)				
timestamp	long										x
hotknock	boolean								x		
expiremillis	timestamp								x		

Table 4.11: Fields of content by type (part 3)

Name	Type	reaction		confirmation		availability		composite		buttonaction		datatransfer	
		M	W	M	W	M	W	M	W	M	W	M	W
emoji	string		x										
messageid	id		x										
type	string				x		x						
messageids	list of ids				x								
text	CONTENT (text)								(x)				
button	(id and text)								(x)				
buttonid	string										x		
refmessageid	string										x		
data	string												x

Table 4.12: Fields of content by type (part 4)

Name	Type	image		file		audio		location		video		sticker		asset	
		M	W	M	W	M	W	M	W	M	W	M	W	M	W
h	int	(x)	x							(x)		(x)			(x)
w	int	(x)	x							(x)		(x)			(x)
mimetype	string	(x)	x	(x)		(x)				(x)		(x)			x
size	int	(x)	x	(x)		(x)				(x)		(x)			x
duration	int					(x)				(x)					(x)
thumbnail_url	string	[x]		[x]				[x]		[x]		[x]			
thumbnail_file	file (encrypted)	[x]		[x]				[x]		[x]		[x]			
thumbnail_info	THUMBNAIL_INFO	(x)		(x)				(x)		(x)		(x)			
tag	string		x												(x)
normalized_loudness	int list														(x)
type	string														x
original_width	int		x												
original_height	int		x												
mac	string		x												
macKey	string		x												
otrKey	string		x												
sha256	string		x												

Table 4.13: Fields of info/metadata

Name	Type	Matrix	Wire
w	int	(x)	
h	int	(x)	
mimetype	string	(x)	
size	int	(x)	

Table 4.14: Fields of thumbnail info

Name	Type	Matrix	Wire
permanenturl	string		x
url	string		x
title	string		x
summary	string		x
urloffset	int		x
image	CONTENT (asset)		(x)
tweet			(x)
=> author	string		x
=> username	string		x

Table 4.15: Fields of link previews

4.3 HTML-Viewer

The *HTML-Viewer* consists of one HTML document. This documents loads the needed **styles.css** from the **websiteData** folder as well as the javascript files **ChatExportAccess.js**, **ChatExportConverter.js** and **ExportObjects.js** used for processing the **chats.xml** document.
The **HTML-Viewer.html** file provides the basic structure of the webpage. We have one container at the left where the user list and the conversation list are located. Then we have the container at the right, where the selected user information, conversation information or the conversation itself can be shown. Except for the structure and some smaller constant elements like the dark/light mode button at the top left and the buttons in the left container at the bottom, everything is loaded with JavaScript later. Because of security reasons, JavaScript is not allowed to read data from any local files at the current location, but only data from URLs. Thats why the **chats.xml** document must be opened by the user. The media files on the other hand can be included with HTML src attributes with local paths, because the browser loads files specified in there. This is because there is no way to access the loaded data with JavaScript and the website provider is thus not able to process or sent the files anywhere and this vulnerability is avoided.
When the **chats.xml** was selected, the *ChatExportAccess* starts processing the file. The XML is read accordingly to the export chat XML schema and the correct corresponding object defined in the *ExportObjects* file is created. This process can be seen at the example of the user object (figure 4.23) in figure 4.22. I chose to create the classes to represent the XML data to keep a structured and fast way to access the exported data and to use these classes at other places like in the *ChatExportConverter*. When the objects are created, they can be accessed at the *ChatExportAccess*. For example in the **HTML-Viewer.html** the users are iterated, converted to HTML and then added to the user list as shown in figure 4.24. The conversion of the user object to HTML is done by the *ChatExportConverter*. This class is responsible for converting objects to HTML depending on the position they should be placed at. For example, there is the **createUserItem** function, visible in figure 4.25, used to create the user item in the user list on the left side. This is also the reason, why I parsed the XML elements into objects. The data is used at multiple places and this way a structured and clear access is possible.

```
22    loadUsers(){
23       let userselem=ChatExportAccess.getByXPath("/chatexport/users").iterateNext();
24       if(userselem!==null){
25          for (let userelem of userselem.childNodes) {
26             if(userelem.nodeType===1){
27                /** @type {User} */
28                let user=new User(
29                   this.getTextContent(userelem,"userid"),
30                   this.getTextContent(userelem,"name"),
31                   parseInt(this.getTextContent(userelem,"accent_color")));
32                this.doWithTextContentIfExists(userelem,"username",(s)=>user.username=s)
                      ↪ ;
33                this.doWithTextContentIfExists(userelem,"teamid",(s)=>user.teamId=s);
34                this.doWithTextContentIfExists(userelem,"email",(s)=>user.email=s);
35                this.doWithTextContentIfExists(userelem,"phone",(s)=>user.phone=s);
36                this.doWithTextContentIfExists(userelem,"trackingid",(s)=>user.
                      ↪ trackingId=s);
37                this.doWithTextContentIfExists(userelem,"userfields",(s)=>user.
                      ↪ userFields=s);
38                this.doWithTextContentIfExists(userelem,"permission",(s)=>user.
                      ↪ permission=s);
39                this.doWithElemIfExists(userelem,"picture",(p)=>user.picture=this.
                      ↪ parseProfilePicture(p));
40                this.doWithAttributeIfExists(userelem,"isSelf", (a)=>{if(a==="true"){
                      ↪ user.isSelf=true}});
41                this.users.set(user.userId,user);
42             }
43          }
44       }
45    }
```

Figure 4.22: ChatExportAccess - line 22 to 45

```
1  class User {
2     /** @type {String} */ _userId;
3     /** @type {String} */ _name;
4     /** @type {Number} */ _accent_color;
5     /** @type {Boolean} */ _isSelf;
6     /** @type {String} */ _username;
7     /** @type {String} */ _teamId;
8     /** @type {String} */ _email;
9     /** @type {String} */ _phone;
10    /** @type {String} */ _trackingId;
11    /** @type {ProfilePicture} */ _picture;
12    /** @type {String} */ _userFields;
13    /** @type {String} */ _permission;
14    /**
15     * @param {String} userId
16     * @param {String} name
17     * @param {Number} accent_color
18     */
19    constructor(userId, name, accent_color) {
20       this._userId = userId;
21       this._name = name;
22       this._accent_color = accent_color;
23       this._isSelf=false;
24       this._username=null;
25       this._teamId=null;
26       this._email=null;
27       this._phone=null;
28       this._trackingId=null;
29       this._picture=null;
30       this._userFields=null;
31       this._permission=null;
32    }
33    /** @param {String} value */ set username(value) { this._username = "@"+value; }
34    /** @param {String} value */ set teamId(value) { this._teamId = value; }
35    /** @param {String} value */ set email(value) { this._email = value; }
36    /** @param {String} value */ set phone(value) { this._phone = value; }
37    /** @param {String} value */ set trackingId(value) { this._trackingId = value; }
38    /** @param {ProfilePicture} value */ set picture(value) { this._picture = value; }
39    /** @param {String} value */ set userFields(value) { this._userFields = value; }
40    /** @param {String} value */ set permission(value) { this._permission = value; }
41    /** @param {Boolean} value */ set isSelf(value) { this._isSelf = value; }
42    /** @returns {Boolean} */ get isSelf() { return this._isSelf;}
43    /** @returns {String} */ get userId() { return this._userId; }
44    /** @returns {String} */ get name() { return this._name; }
45    /** @returns {Number} */ get accent_color() { return this._accent_color; }
46    /** @returns {String} */ get username() { return this._username; }
47    /** @returns {String} */ get teamId() { return this._teamId; }
48    /** @returns {String} */ get email() { return this._email; }
49    /** @returns {String} */ get phone() { return this._phone; }
50    /** @returns {String} */ get trackingId() { return this._trackingId; }
51    /** @returns {ProfilePicture} */ get picture() { return this._picture; }
52    /** @returns {String} */ get userFields() { return this._userFields; }
53    /** @returns {String} */ get permission() { return this._permission; }
```

Figure 4.23: ExportObjects - line 1 to 53

```
39                for(let user of ChatExportAccess.cea.getUsers()){
40                   if(user.isSelf){
41                      document.getElementById("left_header_username").innerText=user.
                            ↪ name;
42                      document.body.setAttribute('data-accent', ""+user.accent_color);
43                   }
44                   contactList.innerHTML+=chatExportConverter.createUserItem(user);
45                }
```

Figure 4.24: HTML-Viewer - line 39 to 45

```
17    /**
18     * @param {User} user
19     * @returns {string} the html code
20     */
21    createUserItem(user){
22       return ('\
23       <div class="left_item" data-id="'+user.userId+'" onclick="showUserDetails(\''+
             ↪ user.userId+'\', this);">'+
24          this.getProfilePicture(user)+
25          '<div class="user_item_info">\
26          <div class="item_name">'+user.name+'</div>'+
27             (user.username==null?"":
28             '<div class="user_item_username">'+user.username+'</div>')
29          +'</div>\
30       </div>');
31    }
```

Figure 4.25: ChatExportConverter - line 17 to 31

5 Conclusion and future work

In this Thesis the benefits of an export were discussed and clearly shown with the development of an implementation for the messenger service Wire. Firstly, we took a look at the current state in multiple messenger services. Then we took a closer look at their implementation and the general problems that may come with an export and compared it with backups. Thirdly, I presented the concept of my implementation and the general structure of the implementation with the reasons for my design decisions. Lastly I provided a deeper look into the details of the implementation.

The implementation is fully functional, but it has the potential to be further improved and enhanced. This includes the possibility to export multiple conversations at once and the possibility to encrypt the exported ZIP file.

JOHANNES KEPLER
UNIVERSITY LINZ

Bibliography

[1] Yoran Brondsema. *The guide to extracting statistics from your Signal conversations*. Sept. 12, 2020. URL: https://www.yoranbrondsema.com/post/the-guide-to-extracting-statistics-from-your-signal-conversations/ (visited on 03/02/2021).

[2] Lukas Forst. *Wire backup decryption*. Aug. 20, 2020. URL: https://github.com/LukasForst/wire-backup-export (visited on 03/02/2021).

[3] *How to download your Facebook Messenger Chat history*. URL: https://www.zapptales.com/en/download-facebook-messenger-chat-history-how-to/ (visited on 03/16/2021).

[4] *How to save your chat history*. URL: https://faq.whatsapp.com/android/chats/how-to-save-your-chat-history/?lang=en (visited on 03/02/2021).

[5] Allen Lee. *Decrypt & Read Chats from WhatsApp Backup File on Android*. Oct. 1, 2020. URL: https://www.backuptrans.com/tutorial/decrypt-read-chats-from-whatsapp-backup-file-on-android.html (visited on 03/02/2021).

[6] *Matrix Client-Server API - Instant Messaging*. URL: https://matrix.org/docs/spec/client_server/r0.6.1#id42 (visited on 03/19/2021).

[7] Andreas Mausch. *WhatsApp Viewer*. URL: https://github.com/andreas-mausch/whatsapp-viewer/releases/tag/v1.13 (visited on 03/16/2021).

[8] Patrick Nolte. *GitHub Fork von wire-android*. Feb. 26, 2021. URL: https://github.com/kanashius/wire-android/commit/b55e9b18de4d88cc5c2a0d4d33bdf294ca14b8c6 (visited on 03/04/2021).

[9] Santiago Obrutsky. "Cloud Storage: Advantages, Disadvantages and Enterprise Solutions for Business". In: July 2016, p. 3.

[10] Alexander Rudyk. *Matrix Recorder*. URL: https://gitlab.com/argit/matrix-recorder/ (visited on 03/16/2021).

[11] *Signal Support - Backup and Restore Messages*. URL: https://support.signal.org/hc/en-us/articles/360007059752-Backup-and-Restore-Messages#android_restore (visited on 03/02/2021).

[12] *Skype Help - How do I export my Skype files and chat history?* URL: https://support.skype.com/en/faq/FA34894/how-do-i-export-my-skype-files-and-chat-history (visited on 03/16/2021).

[13] Alex Smith. *signal-back*. URL: https://github.com/xeals/signal-back (visited on 03/16/2021).

[14] *Tenorshare - 4 Ways to Export Chat History LINE on iPhone and Android*. URL: https://www.tenorshare.com/iphone-tips/export-chat-history-line.html (visited on 03/16/2021).

[15] wondershare. *Wondershare MobileTrans - WhatsApp Transfer*. URL: https://mobiletrans.wondershare.com/whatsapp-transfer-backup-restore.html (visited on 03/16/2021).