Bachelor thesis
Computing Science

Radboud University

Extending Matrix's Functionality in a Secure and Sustainable Way

Author:
Luuk Maas
luuk.maas@ru.nl
s1010080

First supervisor/assessor:
Prof. dr. BPF (Bart) Jacobs
bart@cs.ru.nl

Second supervisor:
H.F. van Stekelenburg
harm.vanstekelenburg@ru.nl

August 18, 2023

Contents

Chapter 1

Introduction

For a long time now, instant messaging and social media have played a large role in our lives. Almost all of the major platforms (i.e. WhatsApp, Facebook, Instagram) provide centralized communication, storing your (personal) data in their massive data centres. In 2014, a company called Amdocs started developing Matrix [13]. Matrix realized a standard that aims at secure, decentralized and real-time communication. It has been published and ready for use in production since June 2019 [4]. To learn more about how the Matrix protocol works, we refer to the preliminaries in the next chapter.

1.1 Problem and Motivation

Up until the moment of writing this paper, the Matrix protocol has mainly been used to implement instant messaging applications, of which the most well-known example is Element [2]. What if you want something more than "just" instant messaging?

As of November 2021, the development of a new (Dutch) community network - named PubHubs - started [10]. PubHubs is an open-source project based on public values. It is open and transparent and protects data of the network's participants.

Obviously, a project like PubHubs needs more than the basic features that are offered by clients such as Element. It would therefore be essential to extend Matrix's (or its clients'[1]) functionality. Preferably in such a way that new features can be added without too much endeavor. Consequently, the research question that will be answered during this paper is: "*what is the best way to extend Matrix's functionality in a secure and sustainable way?*".

Sustainability in this case refers to the desire that when a new feature or bug-fix is needed, this can be done in a quick, efficient way using the method that we find in this research. We also focus on maintainability and

[1] A client is the program that allows a user to send messages through Matrix

the ease of deployment. Security primarily refers to maintaining end-to-end encryption, one of Matrix's key features.

1.2 Contributions

In this paper the following will be discussed:

- We examine the different possibilities to add functionality to Matrix (clients). These include altering the source code of a client like Element (chapter 3); Matrix Widgets (chapter 4) and Matrix bots (chapter 5). For each of these methods we analyze the advantages and disadvantages with respect to sustainability and security.
- A prototype in the form of a shared calendar has been developed using the method we deemed "best" from the previous point, which turned out to be Matrix bots.

Chapter 2

Preliminaries

2.1 What is Matrix? How does it work?

Matrix calls itself an "open standard for interoperable, decentralized, real-time communication over IP." [4] This sentence contains the key concepts of Matrix:

- It is an *open standard*: Matrix published the protocol in detail in their so-called Matrix Specification [5].
- *Interoperable* means that it is possible to communicate with other communication systems.
- Matrix is *decentralized*: there is no single point of failure, nor a single point of storage. Anyone can run a homeserver – which is basically a server that forwards messages.
- Matrix is designed to operate in *real-time*, so it is perfect for applications such as instant messaging.

In order to use Matrix, you need two things:

1. A **client**, which is used to connect to a homeserver. The user can choose this freely. Like mentioned before, a commonly used client is Element [2].
2. A **homeserver**, which forwards messages to other users. These other users can be using the same homeserver, but also different ones. The Matrix.org Foundation[1] has built Synapse, which can be used to run a homeserver [11]. It is currently the most widely installed implementation for homeservers, according to The Matrix Foundation[2]. If

[1] The non-profit organization that guards the Matrix standard on behalf of the Matrix community.

[2] `https://matrix.org/docs/guides/installing-synapse`

you do not want to run your own homeserver, you can use Matrix's homeserver.

2.1.1 User/device identification

Users send messages using a client (i.e. Element). In order to do so, every user needs a user account and a *device*. A user is identified by a username/-domain combination, like so: `@username:domain`. The domain part refers to the current homeserver that a user is using. See figure 2.1.

Every time a user logs in on a new client, it registers as a new *device*. For every new device a set of keys are generated. These keys consist of long-term keys and one-time keys. More on key-management can be found in section 2.2.

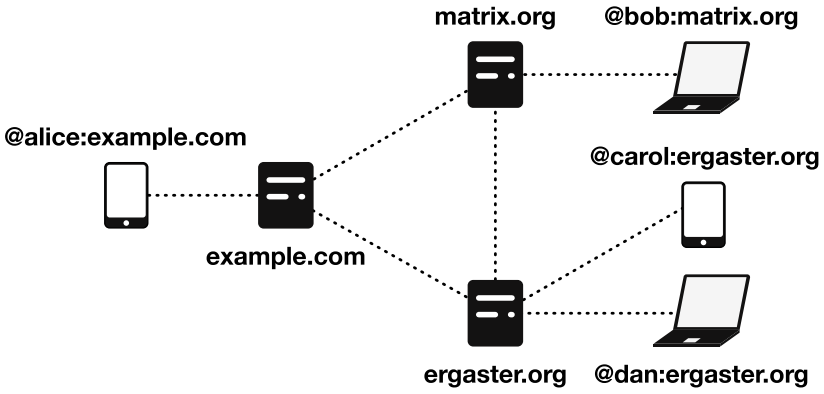

Figure 2.1: Schema of clients connected to federated homeservers, from the Matrix documentation[4]

2.1.2 Rooms

Everything in Matrix happens in so-called **rooms**. Rooms are *distributed*, meaning that they do not live on a single homeserver. A room is identified by a (unique) room identifier and a domain, like this: `!opaque_id:domain`. However, a room may optionally have one or more aliases to increase the findability of that room. A room alias looks like this: `#room_alias:domain`. Note that the 'domain' part (in both the room ID and room alias) just refers to the homeserver that the room was created from. Like mentioned earlier, a room does not live on a single homeserver. The domain part is simply there for namespacing (to avoid clashes of room identifiers between different homeservers) [5].

[4] `https://matrix.org/docs/matrix-concepts/elements-of-matrix/#homeserver`

2.1.3 How messages are exchanged

Assume Alice and Bob each have their own homeserver and established a room together. Now Alice wants to send a message to Bob. First, Alice's message gets sent to Alice's homeserver. Next, Alice's homeserver stores and forwards ("federates") this message to Bob's homeserver which stores the message, too. Lastly, Bob can now retrieve the message from his homeserver.

If Alice now logs on to a new device, she can retrieve the message she sent earlier from her homeserver (if the homeserver approves this request).

Before messages get federated from one homeserver to other homeservers, they are first hashed and signed (using the homeserver's private key). Upon receiving a message, a homeserver first verifies the hash and signature before processing it. The encryption of messages is discussed in section 2.2.2.

2.2 How does Matrix handle key-management?

As mentioned in the previous section, Matrix maintains both long-term and one-term keys for devices. Each time a user logs in to a client for the first time, that client is registered as a new "device". For this device, both long-term keys and one-term keys are generated:

Long-term device keys

- An Ed25519 fingerprint key pair. This key pair is used for signatures and key verification.
- A Curve25519 identity key pair. This key pair is used for shared secret key establishment.

One-term device keys

The one-term key pairs are generated using the aforementioned Curve25519 key pair, and they are used for shared secret key establishment as well.

All public keys are published to the homeserver using the client-server API. The private keys are stored within the client.

2.2.1 Using multiple clients

When a user registers with a new client (device), it will not be able to read prior sent messages immediately. In order to do that, the user (through the client) has to request the keys from its other device(s) through an `m.room_key_request` event. If the "old" client receives and accepts this request, the keys can be exported to the new client.

2.2.2 End-to-end message encryption and authentication

Whenever two users want to communicate securely (end-to-end encryption) using the Matrix protocol, they first verify each other's public fingerprint key using SAS (Short Authentication String) verification. These fingerprint keys are used to sign the identity keys, which can be verified by both parties. Upon success, they establish authenticated encryption/decryption as follows:

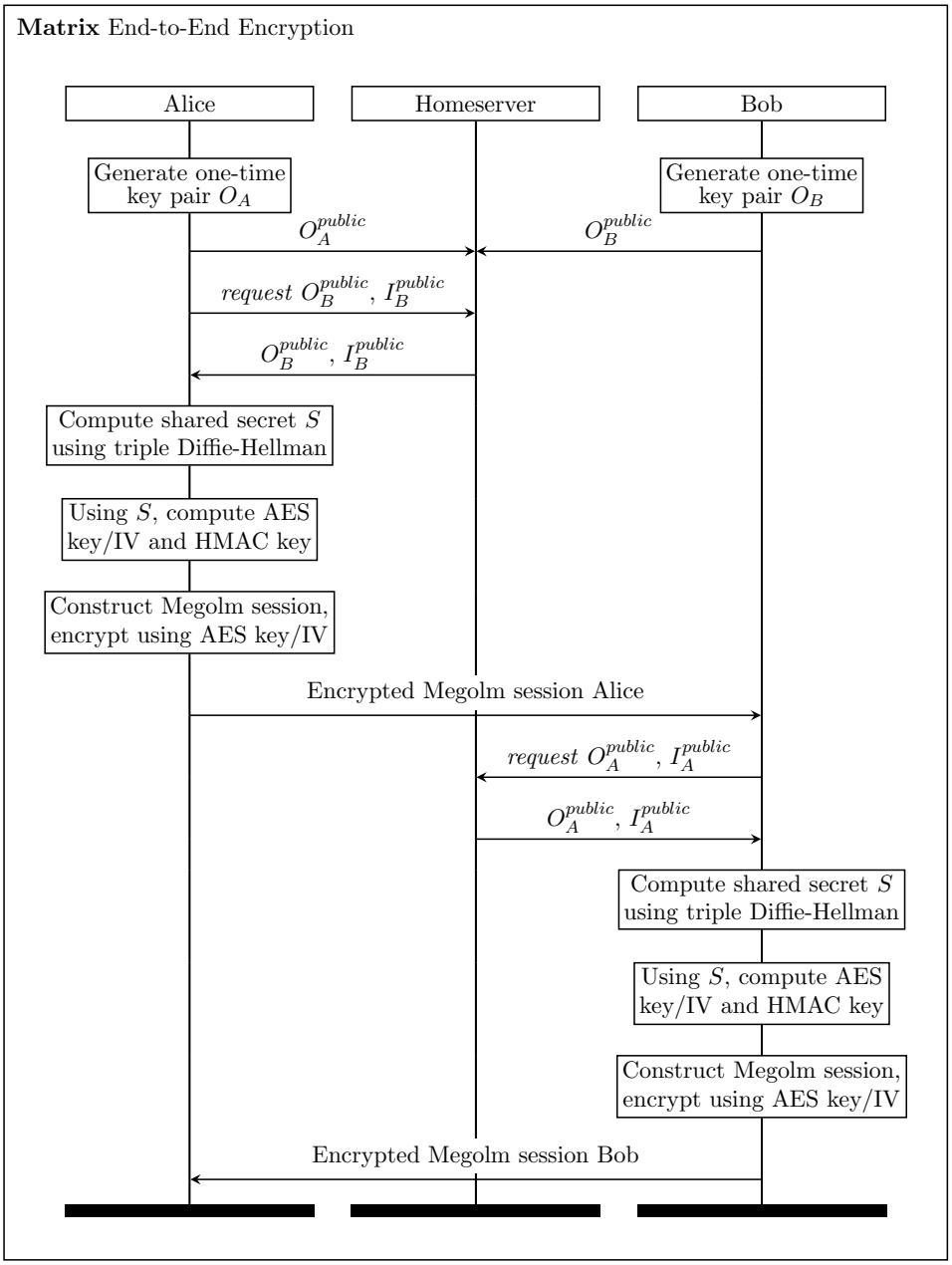

Figure 2.2: Matrix End-to-End Ecryption Protocol

Chapter 3

Altering Element's Source Code

The first attempt to extend Matrix's functionality was to alter the source code of the widely used client Element. Element, like Matrix, is an open-source project [1]. This makes it possible to alter the source code and optionally submit a pull-request to have the features added to the main version ('branch') of Element.

Element maintains three repositories for three different platforms: the web, iOS and Android. Given the fact that the web version has the most forks and is the easiest to setup and test with, we chose the web version as our playground.

3.1 TypeScript

Element for web is largely programmed in a programming language called TypeScript. It is built on top of JavaScript, but with stricter syntax [12]. It converts to native JavaScript, so it runs anywhere where JavaScript runs, like on the web. Although I had not worked with TypeScript before this research, it is fairly straightforward to understand and work with.

Even-though the programming language itself can be understood quite easily, the total Element (web) project is quite extensive and takes some time to fully comprehend. The project is broken down in three components: the Element project itself; the Matrix React SDK[1]; and the Matrix JS SDK[2]. Altering the source code would at the very least require mastering all three of these components and likely also to make changes to multiple components. For the rather short duration of this research it is infeasible to completely master the code enough, let alone extend it.

[1] `https://github.com/matrix-org/matrix-react-sdk`
[2] `https://github.com/matrix-org/matrix-js-sdk`

3.2 Sustainability

The fact that it appears to be infeasible to come up with a prototype for an extension within the duration of this research, points to another problem. As stated in the introduction and research question, we are looking for a sustainable solution. This means that we want the solution not to just fit our example (a shared calendar), but we rather want it to be a general method that allows for the development of a multitude of extensions/plugins.

From that point of view, altering the source code does not seem like a great idea, since every little change to the code (may that be a bug-fix or a new feature) requires a pull request that needs to be reviewed and deployed. Another option would be to maintain a fork of the project, which is not a sustainable option either. We will therefore move away from this idea and explore other options.

Chapter 4

Matrix Widgets

Matrix widgets is another option that appears to be promising for extending Matrix' functionality [6]. In essence, a Matrix widget is nothing more than an iframe embedded in a Matrix room. Widgets can be room-based and user-based. Room-based widgets are attached to a single room and available for all users in that room. User-based widgets are linked to a single user, and available to that user in all rooms. An example of a user-based widget is a sticker pack.

Examples of room-based widgets are a Spotify widget or a Google Calendar widget. Below is an example from [6] on how to create a room-based widget. Note the options `type: "m.widget"` and `url: "..."` which are the key options for creating a widget.

```
{
    content: {
        creatorUserId: "@rxl881:matrix.org",
        data: {
            title: "Bridges Dashboard",
            ...
        },
        id: "grafana_@rxl881:matrix.org_1514573757015",
        name: "Grafana",
        type: "m.grafana",
        url: "https://matrix.org/grafana/whatever",
        waitForIframeLoad: true
    },
    room_id: "!foo:bar",
    sender: "@rxl881:matrix.org",
    state_key: "grafana_@rxl881:matrix.org_1514573757015",
    type: "m.widget"
}
```

In principle Widgets seem promising. However, there is a major flaw that prevents Widgets from being a realistic option for extending Matrix (client) functionalities right now. The problem is that the Matrix Widget API is currently nothing but a proposal. There is a live version of the API, but it is not part of the Matrix specification. That implies that as of this moment, a widget - or more specifically, the iframe - cannot communicate (securely) with the Matrix room or the users that are in that room. Although the Matrix foundation does not give an explicit reason as to why Widgets are not part of the specification, the long list of security considerations (see 4.2) might be part of that reason.

For static applications - like weather widgets, displaying static dashboards or showing a YouTube video - it is not an issue that there is no API between the application and the Matrix room. However, for our example of a shared calendar, we would need a connection between the application (calendar) and the Matrix room.

Given the potential of these Matrix Widgets I do expect that in the future this API will be included in the Matrix specification, after the security considerations have been addressed. Despite the fact that it is impossible right now to develop a prototype using this method, I will elaborate further on this topic for future reference (in case it becomes relevant).

4.1 Sustainability

On the assumption that the API would work, widgets would be a very suitable option for our example: a shared calendar. The calendar itself - including the data(base) - could live on the server that hosts the widget and the users in the room could communicate with it by making (shared) appointments, for instance.

The convenient part is that almost no knowledge of Matrix or its clients is required to implement a plugin like this. It mainly boils down to developing a web application which communicates through an API with the room that the widget is in. This also has the upside that the widget can easily be changed and maintained, if needed. Since we do not alter any source code of Matrix or Element, updates like bug fixes or added features are visible immediately.

4.2 Security

The widget API documentation [6] describes some security issues. For instance, a room administrator could add a widget to a room and retrieve all members' IP addresses by monitoring requests made to the widget. This is not desired for obvious privacy reasons.

Potentially even more worrying is the following note from the security considerations:

> *"The presence of a* ***$matrix_user_id*** *does NOT mean the entity making the request has control of that account, there is no verification step. [...] Widgets may incorrectly assume this and hence do awful things like store important info per-account and think that's secure in any way."*

In other words: there is no such thing as authentication between the Matrix room and the widget. For the shared calendar that would imply that a user can create, update or delete appointments from a different user. This is undoubtedly a security issue that should be fixed by changing the implementation of the widgets and add some form of authentication.

Another issue is the fact that interaction between the Matrix client and the widget - and vice versa - seems to be unencrypted. This is apparent from the following example request (from the documentation), which allows a widget to post a message in a room:

```
{
    api: "fromWidget",
    action: "send_event",
    widgetId: $WIDGET_ID,
    event: {
        "msgtype": "m.text",
        "body": "Hi"
    }
}
```

It makes sense that there is no end-to-end encryption, since the widget is not a "user" of the room. It therefore has no key-pair, and even if it could generate a key-pair it cannot publish its public key to a home-server, nor retrieve other public keys.

4.3 Conclusion

Although widgets have a lot of benefits, there are simply too many drawbacks for now to consider it as a realistic option for extending Matrix (client) functionality. For now, we will move on to a different method, but it does seem worthwhile to revisit this method in the future when the API has been implemented properly and the security flaws have been dealt with.

Chapter 5

Matrix Bots

Besides altering Element's source code or implementing widgets, there is also the possibility of adding so-called *bots* to a Matrix room. Technically, a bot is nothing more than a user in a Matrix room that returns certain messages based on input messages - which you could see as a "command" - with optional arguments. As a simple example, a bot could function as a dice, that upon calling it returns a random number between 1 and 6. Another example is a weather bot, that returns the weather from an external API based on a given location.

A bot can be built from scratch, but there also exists a very convenient bot manager called Maubot [9]. Maubot runs as a separate application which can be setup on any server. For convenience I would suggest running it on the same server as the Matrix homeserver (where the room was created), which is what I have done during the development of the shared calendar prototype. Maubot comes with some official and third party plugins [9], but it is designed to allow developing your own, custom plugins (written in Python), which is exactly what we are looking for.

5.1 Maubot Manager Setup

As mentioned, the Maubot manager can be run on any server, since it is basically a normal Matrix user. The Maubot manager communicates to the homeserver using the Client-Server API [5], which allows for convenient security features (see 5.5). This also means that Maubot manager is not dependent on the homeserver implementation: it works with any homeserver.

In principle anyone could setup a Maubot instance, both a random user or the room administrator. On the other hand, the room administrator can simply restrict this by (not) allowing a user - that is a bot - to join their room.

5.2 Installing and Using a Plugin

After setting up the Maubot manager (for instance using their Docker image [7]), installing and using a plugin is quite straightforward:

1. The first step is to create a `Plugin` by uploading a `.mbp` file to the server using the Maubot manager. As mentioned earlier, you could download an existing plugin and upload it, or develop and build your own.

2. Next, a `Client` has to be created in the manager. This `Client` is an object that is linked to a user from a certain homeserver.

3. Lastly, we create a so-called `Instance`. An `Instance` assigns a `Plugin` to a `Client`.

Once the corresponding user has been added to a room, you can use the bot by calling the command, preceded by an exclamation mark. For instance, the earlier mentioned dice bot can be used by sending `!roll`.

5.3 Developing a Prototype

Creating a custom bot is very straightforward, and a simple bot can be developed in a matter of minutes. Depending on the complexity of the bot, this may obviously take longer. A Maubot bot consists of two components, a `maubot.yaml` file and some Python modules, which you point to in the `yaml` file. The `yaml` file also indicates whether or not your plugin requires a database. In our case we need a database to store the calendar items (appointments).

For my testing environment I setup my own homeserver using Synapse [11]. I then downloaded Element for MacOS and set it up with my homeserver. Lastly, I also installed the Maubot manager using the Docker image.

Step 1 of developing the `calendarbot` - as we will call the plugin from now on - is creating `maubot.yaml`. This looks like the following:

```
1 maubot: 0.1.0
2 id: pubhubs.calendarbot
3 version: 1.0.0
4 modules:
5   - calendarbot
6 main_class: CalendarBot
7 database: true
```

The above configuration tells Maubot to find the `calendarbot` Python module and look for the `CalendarBot` class inside of it. It also tells Maubot that our plugin requires a database.

Now let us take a look at the `CalendarBot` class[1] in appendix A. For the (very) basic prototype that I have developed, I created two commands:

1. `!create_appointment TITLE DATETIME [PARTICIPANT]`
   This command can be called by any user and will create a database entry with the specified details. The last argument (`PARTICIPANT`) is optional and may be used to "share" an appointment with another user. It also automatically stores the `user_id` of the user who created the appointment and the `room_id` of the room the appointment was created in.

2. `!get_appointments`
   As the name suggests, this command retrieves all appointments from the user who calls it. It only returns the appointments that were created in the room that this command is being called from. This way, all appointments can be stored in the same database table, while keeping the appointment separated between rooms.

5.4 Sustainability

Now that we have implemented a (simple) prototype for a shared calendar, let us review the sustainability of Matrix bots. As we have seen, Matrix bots are relatively simple to implement once the Maubot manager is setup. Besides very rapid development, the plugins are also very easy to maintain. After a change, the new version of the plugin can simply be uploaded to the Maubot manager. The only thing that is required for the changes to take effect after that, is a reload of the `Instance`.

5.5 Security

From a technical perspective, bots do not add anything to Matrix or the client. They are simply users that listen for certain messages and respond accordingly. Therefore, the security of bots are the same as when two users exchange "normal" messages. Bots are compatible with end-to-end encryption, as long as the Maubot manager runs in an environment that has some dependencies installed [8]. These mainly include Python bindings for the Olm C library.

[1] Note: this class depends on the `Database` class from `calendarbot.database`. It is too extensive to include here, but can be found on my Github repository that I made for this thesis [3].

Because of the end-to-end encryption between users/clients, the communication between a user and the bot (which is technically also a user) is authenticated and confidential (see section 2.2.2). The calendar entries are stored in plain text however, like all regular messages in regular clients, like Element. To retrieve calendar entries, the sender ID and room ID are checked to retrieve only the correct entries.

5.5.1 Other Security Considerations

Since multiple bots (`Plugin`s) can live on the same Maubot manager instance, it is important to realise that they both use the same database. This means that caution is required when installing multiple `Plugin`s: checking the source code for malicious database queries is probably a good idea. I would even suggest to not install multiple `Plugin`s that use a database on the same Maubot manager, to prevent malicious database queries.

5.6 Conclusion

Overall, bots have a lot of advantages, especially from a security point of view since they have the same security features as a normal Matrix user. Moreover, the relative little development time needed to implement a plugin using the above method is a big plus. The simple prototype we developed proves this point.

Chapter 6

Conclusions

During the course of this thesis, we have looked at three different methods for extending the functionality of Matrix and its clients. The initial attempt - altering the source code of Element - quickly turned out to be infeasible within the time-frame of this research. Moreover, altering source code is such a complicated job, that it is absolutely unsuitable for implementing different, new plugins.

Next, we explored Matrix widgets. Although it definitely has some promising features - such as good maintainability and quick development - it still lacks a good and secure API that makes it possible for a widget to communicate with the room. For the time being, it is not quite suitable for developing sustainable plugins. However, it does seem worthwhile revisiting this method in the future.

Lastly, we investigated Matrix bots. The simplicity of bots made it to be a surprisingly good option for extending Matrix client functionality. Due to the fact that bots are technically regular Matrix users, they benefit from the same security features. For instance, the communication from the user to the bot (another user) is end-to-end encrypted. Additionally, we have seen that bots are incredibly easy to implement and maintain.

When considering both security and sustainability, we conclude that developing bots is the most advisable method to add functionality to Matrix.

6.1 Future work

As you may have noticed already, this research did not focus on user-friendliness at all. At the beginning of this research we decided to move that out of the scope. Bots are undoubtedly not a great choice from a usability point of view. That is why I would suggest future research to focus on how to improve the usability of bots. For example by making the "commands" selectable from a list in the client.

Bibliography

[1] Element github repository. https://github.com/vector-im.

[2] Element official website. https://element.io.

[3] Luuk Maas GitHub. https://github.com/luukmaas/pubhubs-plugins.

[4] Matrix official website. https://matrix.org.

[5] Matrix specification. https://matrix.org/docs/spec.

[6] Matrix widget api v2. https://docs.google.com/document/d/1uPF7XWY_dXTKVKV7jZQ2KmsI19wn9-kFRgQ1tFQP7wQ.

[7] Maubot documentation. https://docs.mau.fi/maubot.

[8] Maubot encryption documentation. https://docs.mau.fi/maubot/usage/encryption.html.

[9] Maubot github repository/website. https://github.com/maubot/maubot.

[10] Pubhubs official website. https://pubhubs.net/en.

[11] Synapse github repository/website. https://github.com/matrix-org/synapse.

[12] Typescript official website. https://www.typescriptlang.org.

[13] Teemu Makkonen. Implementing encryption for qt-based matrix client, 2019. https://www.theseus.fi/bitstream/handle/10024/264169/makkonen_teemu.pdf.

Appendix A

CalendarBot Code

Below you can find the CalendarBot Python class.

```
class CalendarBot(Plugin):
  db: Database

  async def start(self) -> None:
    await super().start()
    self.db = Database(self.database)

  @command.new("create_appointment")
  @command.argument("title", required=True)
  @command.argument("datetime", required=True)
  @command.argument("participant", required=False)
  async def create_appointment(self,
                               event: MessageEvent,
                               title: str,
                               datetime: str,
                               participant: str):
    self.db.create_appointment(event.sender,
                               event.room_id,
                               title,
                               datetime,
                               participant)
    await event.reply("Succesfully saved appointment '" + title + "'")

  @command.new("get_appointments")
  async def get_appointments(self, event: MessageEvent):
    appointments = self.db.get_appointments(event.sender, event.room_id)
    await event.reply("Appointments: \n" + appointments)
```