TOZTI Final Report

'Projet Intégré' — ENS Lyon

More information at "https://tozti.github.io"

- Léonard Assouline
- Peio Borthelle
- Guillaume Cluzel
- Guillaume Duboc
- Julien Ducrest
- Lucas Escot
- Joël Felderhoff
- Félix Klingelhoffer
- Romain Liautaud
- Pierre Meyer
- Alex Noiret
- Pierre Oechsel
- Lucas Perotin
- Vincent Rebiscoul
- Emmanuel Rodriguez
- Daniel Szilagyi
- Lucas Venturini
- Damien Reimert (supervisor)

March 23, 2018

Contents

Chapter 1

Introduction

In this report, we describe TOZTI, the open source centralised platform for managing (student) association. Its aim is be a ready-made ("batteries included") solution, tailored specifically towards managing the internal organisation of an association, while still remaining extensible and easily customisable by advanced users.

The ENS has a rich and diverse associative life, and many of the student associations require efficient means of internal organisation and external communication. Currently, they use software ranging from a simple email client to a fully custom solution for managing members, meetings, and finances, in addition to social networks. The biggest players in the field of associative software are Facebook, mailing lists, and Adeline – developed by former ENS students who grew tired of manually managing their hundreds of members. Overall, the existing tools are well suited for public relations (e.g. informing members of upcoming events), but do not allow for efficient internal organisation (e.g. storing documents, meeting of the board, scheduling events, archiving the work that was done). Many student associations are therefore dragged down by the tools they use, which are often counterintuitive and do not integrate well with one another. Most notably, this leads to a lot of redundant *plumbing* work, and makes it hard for users to find the information they want to access, as it is scattered over multiple sources. Even more surprisingly, the ENS isn't the only school subject to such problems. Associations of other schools (Polytechnique, Centrale, ENSTA, EDHEC) are also in the same situations: they are using a patchwork of tools for their internal organisation.

There is a real need for a data centralisation which TOZTI, the centralised associative platform, will provide. The TOZTI project will **allow each association to build the unified organisation suite that best fits its needs** by choosing from a wide range of well-integrated modules. These modules, which we will develop over the course of the project, will each answer a specific organisational need: *communicating* in a hierarchical manner, reaching *consensus*, *planning* tasks and events, *archiving* files securely, *collaborating* on textual documents, or even *automating* recurring tasks.

We ultimately want to build a service that student associations can use straight away. To this end, we maintain close communication with their representatives, to ensure that what we are building suits their needs.

In an effort to be flexible, our software architecture will revolve around small, loosely-coupled but interoperable modules, which we will build using today's web standards and best practices.

Chapter 2

Specifications

In order to determine the exact requirements for our product, the *communication team* conducted interviews with several student associations at the ENS as well as several engineering schools that also have a rich associative life. They were asked what solutions they were currently using or have used over time – which we will discuss in the next chapter – and what features an ideal internal organisation tool might have. The answers they gave us can broadly be split into two main categories: user expectations and developer expectations. The further subdivision of these categories is described in the following sections.

2.1 User Features

Upon interviewing the (non technically inclined) users, we realised that their answers also fell into several distinct groups. We decided to implement some of the most requested features:

Unified overview of relevant information. The users should be able to see an overview of all events, discussions, etc. that are relevant for them in a single place. This requirement has a very deep meaning: TOZTI should be able to seamlessly display information from one association or another but it also must not become another all-integrated platform locking users inside it.

Intuitive and consistent organisation. The users should have two ways of accessing their files: an *intuitive* one, that presents the most recent documents or unread notifications, and a *structured* one, which enables the user to find a given piece of information in a consistent fashion. The first part will be implemented using a dashboard-like interface, and the second one will resemble a file browser, with abstractions similar to files and folders.

Calendar. A distributed calendar that displays the events from all associations of which the user is a member. This feature should be accessible to each and every student of the school, not only the ones that are part of the organisation group of an association. Optionally it should support event organisation, with a form system helping the association members assign themselves to a given time slot at an event (*permit* system).

File storage. A versioned and collaborative file storage service, that interfaces with the other components. The stored files should be able to be referenced from other places in the system. Any file type can be stored but we might provide rich editing and viewing for some of them (for example, multimedia, formatted text or PDF).

Discussion board. A standard bulletin board, with the additional feature of being able to reference other entities in the system such as events, files, etc. For example in an ongoing discussion about an event on the forum, one will be able to reference both the calendar event and the event poster image.The discussion feature should also allow users to switch topics in a fluid and non chaotic fashion, in opposition to most of the currently used instant messaging services.

You can find in appendix C the full specification we wrote.

2.2 Developer Features

Apart from the non-technical users, we also communicated with the developers and power-users that maintain internal organisation systems within individual associations. Given that most of these systems were developed ad-hoc, with time they often become increasingly hard to maintain and extend. Thus, the main request from the developers was to provide them with a convenient API with which they can interface. Upon presenting them with our planned user-facing features, we agreed on the following:

Ability to create new views and extend existing ones. The developers should be able to add completely new components (such as a document viewer component), or extend existing ones (like the dashboard).

REST HTTP API. The entire service state should be queryable through a well-defined REST HTTP API, such as JSON API. This would enable the developers to automate common actions and create new clients, other than the web-interface.

Implementing these features implies a fully modular architecture, where the core server knows close to nothing about the entities it handles, and treats them completely uniformly. This further allows us to have a consistent representation of all these entities, when returned by the API.

Chapter 3

Existing solutions

Based on the specification from the previous chapter, we evaluated several existing solutions that are currently being used for the internal organisation of the associations we interviewed. We considered several platforms: email, Slack, Facebook, Adeline, Doodle, Google Apps, and hosted Git services (GitHub and GitLab).

Apart from most of them being closed-source (which limits the degree of extensibility), all of these platforms have their own specific advantages and drawbacks. They are presented in the remainder of this chapter.

3.1 Email

Email is the probably the most commonly-used communication platform today. As such, it is not surprising that it is the preferred platform for many of the smaller associations. Its most prominent advantage is that virtually no setup is required before it can be used, as everyone is issued an insitutional email address (e.g. *@ens-lyon.fr*), which can be easily accessed via the school Webmail. Experience has shown that such a system is feasible as long the the monthly volume of sent messages is relatively low – however, most people are members of multiple associations while only being active in a select few, which means that they receive a lot of emails of no interest to them.

More specifically, this problem occurs because there is no standardised way of classifying emails (apart from the email subject). Thus, with each sent message, it becomes increasingly hard to have an overview of all the information that was exchanged between the members of a single association, and in a sense the data gets lost.

The other disadvantage of emails stem from the fact that all non-text data needs to be sent as a small embedded attachment, or as a link towards a third party service. Both of these approaches complicate information retrieval, as it is all but impossible to search the contents of the attached files or links. Additionally, the use of third-party services contributes to information fragmentation.

3.2 Slack

Slack is a proprietary cloud-based collaborative instant messaging platform, originally designed to replace the wide variety of general-purpose instant messaging services used by teams in a professional environment. As a messaging-first platform, it shares many of the same disadvantages as emails.

Still, it has several interesting features, most notably its integration with third-party services. Namely, those services can implement *bots*, so that Slack users can access those services by exchanging messages with a bot. Unfortunately, the chat-based form of this communication often limits the degree of interaction with the third-party service, sometimes so much that external links are still needed. Therefore, for similar reasons as email, Slack is also unsuitable for mid- to large-sized associations.

3.3 Facebook

Facebook is the second most used communication platform, and just like emails, almost everybody already uses it. On the other hand, because of its nature (determined by the fact that it is run by a for-profit corporation that focuses on selling private data), many people are also uncomfortable with using it for either professional or academic matters. Although Facebook's messaging component (Facebook Messenger) contains some advanced features (polls and event planning), it is still not a proper long-term high-volume archival solution. Because Facebook's UX is centered around a powerful search-box and not a strict taxonomy like trees or tags, old content might easily get lost. Additionally, because of Facebook's closed nature, interoperability with third-party services is difficult.

Recently, Facebook introduced *Facebook Workplace*, a platform that leverages Facebook's group feature, and adapts it to a more professional setting. However, the majority of philosophical and practical problems applies to Workplace as well – chief among them being that it is a closed platform, and as such, there is no way that it can have anything more than superficial integration with other solutions it claims to interoperate with (G Suite, Office Online, Dropbox, etc.).

3.4 Adeline

Adeline is a service created by (former) ENS students, marketed as a social network for managing a student's associative life. Our analysis concluded that Adeline takes a lot of concepts verbatim from Facebook: it is more a social network for regular associations members than an association management platform. Thus, most of the criticism aimed at Facebook applies to Adeline as well. One notable feature however is the ability of association staff members to create forms (speadsheets). In fact, implementing such a feature in TOZTI is one of our long-term plans.

3.5 Doodle

Doodle is an event-scheduling service that helps a group of people in finding a common time slot when they are all available. It is well integrated with personal calendars, which is a strong feature. But given that it is such a special-purpose service, it can only be used in conjunction with other document storage and communication services.

3.6 Google Apps

Google provides a wide range of services, aimed at private users and organisations. It provides tight integration between the services, and connects all of them with their search engine. However, since it is completely hosted by a third party (Google), it cannot be modified or extended in any way. Since Google Apps are not implemented with associative workflows in mind, this is very much a problem.

3.7 Framasoft Apps

Framasoft also provides a wide range of services (e.g. event-scheduling, poll, forms, etc.), with less integration between them or with other services as their Google counterparts, but with the distinct advantage of being a completely free software, free licence, and an open source based solution. However, these solutions can only be used in conjunction with other services, because of their special-purpose design.

3.8 Hosted Git services (GitHub, GitLab)

Hosted Git services provide a publicly (or privately) hosted Git server, augmented with features that enable more efficient software development. As such, they are ideal for storing versioned text files and their collaborative editing. They are usually beloved by – but also restricted to – tech-savvy users (one such group of users at ENS is the AliENS association). Exactly because of

their special purpose, it is hard to extend these platforms to support non-software-development workflows e.g. calendars and event scheduling.

Chapter 4

Our solution

4.1 Choice of technologies

We chose *Python* and *Javascript* supported by the *Vue.js* framework in order to develop TOZTI. Finally, the database backend is provided by *MongoDB*. We describe the reasons of this choices below:

4.1.1 The web as a platform

The web is not only about HTML and CSS anymore, it is now the overwhelmingly dominant platform for building user application. The reasons for that are multiple but here are some we could think of:

- HTML and CSS provide an easy to use declarative language for designing tailored graphical interfaces enabling fast prototyping.
- The *installation* step of the software is hidden from the user. A package manager or an installer is not a very complicated tool but it still adds friction (similar mechanisms are at play in compiled versus interpreted languages).
- It is not an abstraction layer on top of heterogeneous platforms but a fully featured and comprehensive platform.
- Almost every consumer device with a screen has a web browser.

For these reasons – as well as the simple fact that the prevalence of this platform brings modern and well supported tools – we have chosen to develop a *single-page app* in *Javascript*. We picked the *Vue.js* framework because it is lightweight, yet still powerful enough to express the most interesting design patterns. Additionally it is well documented and has a large community and an ecosystem built around it.

4.1.2 Python

The core API server language has few constraints besides having a good HTTP server library. We chose Python for the language as it is easy to both write code and understand somebody else's. The public we are targeting (ENS students at first, and afterwards students from other schools and universities) will almost certainly be at least a bit familiar with *Python*. One noteworthy detail is that we used the *aiohttp* library leveraging the new asynchronous IO features of Python, that are well-suited to highly concurrent tasks such as those performed by a HTTP server.

4.1.3 MongoDB

As described in section 2.1, our idea is to store the data as a rich network of resources that can reference each other in a variety of ways (relationships). Additionally, in order to allow third-party extensions to define new resource types, it is easier from a developer's point of view to work with a database that that does not impose a fixed schema (besides the necessary minimum of the object's

ID and type). Therefore, MongoDB seemed to be a good fit for us: it is a mature document-oriented schemaless database, with excellent Python support. However, it should be noted that only a small part of TOZTI depends directly on MongoDB: the storage abstraction layer. All other components, both server- and client-side communicate only with the storage abstraction layer through its API: either by using it directly (on the Python side), or calling it through the exposed HTTP methods (on the client side).

4.1.4 Adoption and installation

Most associations have only one person in charge of maintaining their infrastructure. We know that to convince this person of using TOZTI over an other solution, TOZTI must be:

- easy to deploy,
- easy to configure,
- (optionally) easy to adapt and tweak.

We wrote an exhaustive installation guide explaining how to install TOZTI. However, as we worked on TOZTI, the complexity of installation grew. Therefore we decided to provide a **Docker** container allowing a system administrator to create an instance of TOZTI easily.

To make sure the copy of TOZTI provided on our GitHub always remains functional, we implemented **continuous integration** on the repository. Every change on the code must pass through multiple tests making sure no regressions happened. **Travis** is used to automatically execute these tests on the GitHub repository, on each commit. As of now, over 100 tests are written, achieving 80-90% code coverage. The tests mainly focus on module loading, authentication and storage. We also wrote some tests using *Selenium* to test the UI of TOZTI end-to-end. The mechanism of branching was also used extensively while developing new features. Each new feature has to pass rigorous code reviewing before being accepted and merged with the public version of TOZTI.

4.1.5 Documentation

As suggested earlier, and in adequacy with our modular design, we want to make it easy for external developers to contribute to TOZTI. We wrote several documentation articles, all accessible on https://tozti.readthedocs.org. We also provide a sample extension, that demonstrates how to interface both with the server- and client-side of TOZTI.

Finally, we chose the **AGPL** licence for TOZTI. We set for ourselves the goal of providing a *libre* tool benefiting the common public, and thus a strong copyleft license was agreed upon. Because we believe that services disempower people while tools and protocols empower them, we took the adopted the *Affero* clause of AGPL, effectively ruling out inclusion of TOZTI in any closed-source service.

4.2 Architecture

TOZTI is implemented as a modern web application. As such, it is divided in two parts, the *backend* and *frontend*. The backend code is executed on the server, whereas the frontend code is executed on the client (i.e. a web browser). Additionally, *extensions* (also called *modules*) can extend the functionality of both parts, as mentioned in the specification.

An overview of the architecture can be seen on figure 4.1. Black arrows are function calls (JavaScript on the client-side and Python on the server-side), blue arrows are HTTP queries and the red arrows are WebSocket connections. We can see how the core (the large rectangle on the left) interacts with extensions (narrow rectangles on the right).

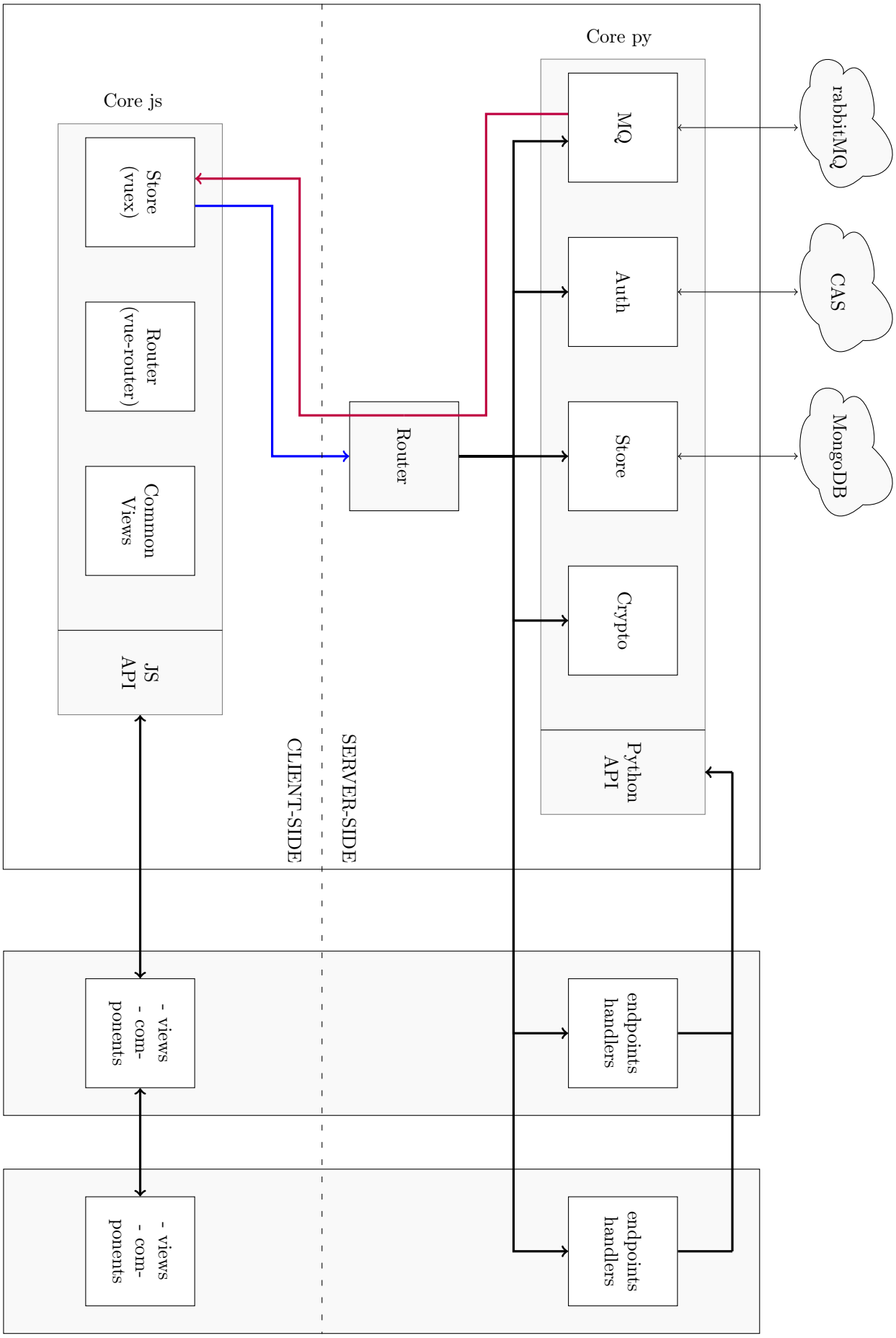

Figure 4.1: Architecture diagram

4.2.1 Backend

The backend consists of three communicating components: the HTTP API server, the MongoDB storage server, and the RabbitMQ message broker. The purpose of the API server is to act as a middleman between the frontend and the storage server. It validates the incoming data, formats the outgoing data, and takes care of the permissions for accessing data in the storage. The communication with the frontend is done through a HTTP API, that is heavily inspired by JSON API[5]. Extensions can introduce new functionality by hooking new HTTP endpoint handlers into the core HTTP server.

4.2.2 HTTP API

A JSON API inspired API has been chosen because it follows the well-known *HATEOS* architecture (Hypertext As The Engine Of Application State). This essentially means that the client need not to construct URLs *ex nihilo* to navigate the API: a single entry point is needed (in our case, `/api/auth/me`), after which URLs for all related actions will be provided inside the HTTP reply. This means that the state of the client *is* the document given as the reply, and possible state transitions are all included as hyperlinks inside this state. The benefit of HATEOAS—in addition to being the pedantic 3rd and highest level of RESTful API design—is that it makes the API more discoverable and clients more robust. Such a strong contract between the client and the servers enables us to keep the client completely unmodified, even in face of relatively large changes, such as server hostname change, URL change or type description change (see below).

Resource objects

JSON API is centered around *resource objects*. They are what we would call *entities* in databases or *hypermedia* on the web. A resource object is a simple abstract structure: it has an unique `id`, a `type`, a few meta-data (`meta`) and finally a `body`. Extension can declare different resource types, constraining what goes into the content.

There are several kinds of resource objects:

- Structures: the type declaration specifies the fields names and value types. The body is a JSON object mapping fields to their values. Supported operations are `get(field)` and `set(field, value)`.
- Arrays: the type declaration specifies the type of the items. Supported operations are `get(index)`, `append(value)`, `insert(index, value)` and `pop(index)`.
- Dictionaries: the type declaration specifies the type of values (keys must be strings). Supported operations are `get(key)`, `set(key, value)` and `pop(key)`.

Type Description Objects

We will introduce types since they are treated abstractly in the original JSON API specification. Type are simply namespaced strings so that each extension can create its own types, in addition to the ones provided in the `core/` namespace. New types can be declared together with a type description object in an extension.

Type description objects ressemble JSON schema but we added the new `relationship` type. A relationship is a link between the current resource and either one or many target resources. This enables us to create a directed tagged graph between resources and build complex data-structures like a file system or an identity management system.

For example, the `core/user` type might specify a *to-many* relationship called `groups`, whose targets are restricted to the `core/group` type. A partial description of the `core/user` type can be seen below:

```
{
    "body": {
        "name": {
            "type": "string"
        },
        "handle" : {
```

```
            "type": "string"
        },
        "groups": {
            "type": "relationship",
            "arity": "to-many",
            "targets": "core/group",
        }
    }
}
```

This creates a directed graph, but regularly we want to have bidirectional links where both directions are kept in sync. For example, we would like `core/group` to specify a relationship named `members` mapping to the list of `core/user`s that have access to the group. To handle that, we add a new kind of relationship description: *auto* relationships. These relationships cannot be modified by the user, they are specified by a target type and a forward-relationship name—here `core/user` and `groups`. This relationship is not kept in the internal database, but upon query of a given object we fill it we the result of the query "all resources of matching type such that the current resource is contained in the matching relationship".

Authentication

To perform a password login, we use state-of-the-art Argon2 password hashing from the libsodium[4] library. Note that our authentification scheme if currently trivial (*eg* post your password to the `/login` endpoint). This scheme is the most common and recommended, on top of SSL/TLS, but it has some pitfals: it does not provide mutual authentication and permits easy man-in-the-middle attacks, either by an active network attack or by a compromised server. Thus we will probably switch to SCRAM[1] We might also implement additional login schema like CAS (to integrate with the ENS de Lyon single sign-on), GPG authentication (for power-users that have a key), TOTP[3] or U2F[1] (for power-users that have a hardware security token).

The login endpoint answers with a macaroon[2] cookie. A macaroon is composed of public key signed data, in order to ensure the authenticity. In our current implementation, it contains the handle and the uuid of the current user, in order to access the database quickly. This macaroon can be seen as a capability authorizing its bearer to act as the user. The macaroons also provide advanced functionalities such as delegations and caveats, which will prove useful if we ever switch to a multi-server architecture.

4.2.3 Frontend

The frontend follows the "single-page application" approach, which is the de facto standard for modern web application development. Essentially, it means that the application interacts with the user by rewriting the current page rather than loading new pages from the server, which improves the user experience by making the whole application feel more responsive.

In this approach, the client retrieves most of the application logic on the first request to the website. This includes the routing layer (which tells the browser which page to render depending on the URL) and the rendering layer (which describes the HTML code to produce for a given type of resource). This way, subsequent page changes do not trigger a full page refresh: instead, the client uses the HTTP API to retrieve JSON objects for the resources that need to be displayed, and renders them locally.

Specifically, our implementation uses well-established tools such as *Vue*[8], *Vuex*[9] and *Vue Router*[7] to help with the routing, state management and rendering. Like most recent web frameworks, Vue allows for declarative rendering of small, reusable components, which simplifies the development of single-page applications. This way, even developers with limited web development skills can write relatively robust and modular code. Working with small components also allows us to easily **create new views and extend existing ones**, as required by chapter 2.

[1] SCRAM was originaly specified for TCP sockets but the experimental [6] ports it to HTTP.

4.3 User Experience

In order to streamline the development process, we created mockups for the most complex components, so that their design is already known at implementation time. This process is known to save (developer) time, as it allows them to focus on actual code, as opposed to design and usability concerns. The mockups can be found in appendix A.

4.3.1 Taxonomy

As specified, we want a consistent organisation of the resources (a taxonomy). Usually, this is done with a resource hierarchy, that is a tree. However, a common requirement is to share some resource with multiple people. On a file system this is usually done with symbolic links. Our approach for the taxonomy is closer to hard links than symbolic links: we allow resources to show up at multiple places in the tree. This taxonomy could also be described as being the crossover of a tag system and a tree: the set of tags form a tree and every resource can be tagged one or more times.

4.3.2 Common UI

There are two UI elements that we want to be present in every view of TOZTI. Firstly, there is a thin horizontal header at the top containing the profile, settings and notification menu. Secondly, there is a vertical strip on the left containing quick-links to some nodes of the taxonomy tree (folders, groups, ...). These quick-links will contain the root directory of every group the user is a member of but will also be customisable by the users. See the top left drawing of figure A.2 for these common parts.

4.3.3 Core Views

Besides the user profile, group profile and settings views, there are two more views that are provided by the core.

The first one is the *directory* view, that is the consistent view of directory (also called tag) of the taxonomy. As such it can be compared to the main window of a file browser or to *category list* view in bulletin-boards. The content of the view is straightforward: the main pane contains a list of items, each describing an element of the directory. Each list item contains the name its name as well as some additional information based on it's type (it can be a file, a thread, an event or a subdirectory). On the right, we have a full-height column showing the file preview, meta-data and interactions like *share-to* or *edit* when the user selects an item. For a mockup, see figure A.1.

The second one is the dashboard and landing page. This view is much more versatile and can be extended at will with new widgets. It shows things like *upcoming events*, *recently modified resources* etc.

4.3.4 Multimedia

The *multimedia* file extension mostly provides a right sidebar for viewing the meta-data of a file, as well as the context of this file in TOZTI. This includes: group membership, user rights, viewers, etc. It should also serve as a basic visualization tool for text or PDF files...

4.3.5 Calendar

The *calendar* extension is centered around *events*. Events are presented on a calendar (grid where columns and rows corresponds to a specific day and time). Events can be shared between groups of persons or restricted to a sub-group of the current group. This notion of sharing and restricting is integrated directly thanks to the *taxonomy*. Furthermore, events can be defined to be recurring or one-off. As such, the calendar view is made of two big components:

- A search bar with some controls to search through specific events, see the events planned later or earlier in time, and add an event.
- A calendar displaying the events of the week, month and day.

When a user chooses an event inside the calendar, an overlay page will allow him to view a more precise description of this event. He will be able, if he has the permissions to, to edit and delete this event. See figure A.2 for the mockup, as well as figures B.10 and **??** for the actual implementation.

The final view is composed of three frames that correspond to the daily, weekly or monthly representation of an agenda. The design is simple but clean, and integrates into the rest of the TOZTI layout. A right sidebar is mainly used for event creation, modification or simply to view the details of an event.

4.3.6 Discussion

As detailed in C, the main resource type of the *discussion* extension is the *thread*. As such, it provides a classical bulletin-board view of a given thread: a vertical sequence of messages followed by a message redaction zone. On the right there is a full-height column displaying the list of resources that have been mentioned in the thread. For example the organisational thread of some upcoming event might mention resources like the public event, the staff task-attribution form or advertisement PDFs and images. The right column also features a "scrubber", which allows the user to easily scroll to a specific message. This is especially useful for long threads, which would otherwise be hard to navigate. See figures A.3 and A.4 for mockups.

4.4 Security

4.4.1 Our problem

Goals & Motivations

When we surveyed prospective users of TOZTI (i.e. student associations), one particular use-case came up which required us to set up cryptographic security: some associations have to collect very sensitive data such as testimonies of sexual harassment. For this reason among others, we require that all data server-side be encrypted and be only accessible to those with the necessary clearance level.

In particular, one important requirement for our storage system is that the server's owner cannot have access to unencrypted data. This is both for obvious privacy reasons, as well as to keep our hands free for if we ever want to distribute the data over multiple servers. Indeed, one long term project for the storage is to be compatible with multiple servers (an object of a certain type could for instance be stored on a server even though its type is only defined on another server). In that scenario, we cannot possibly trust an increasing number of server owners, even if we are ready to trust just one.

For this same reason, we require that decryption happen client-side as much as possible. One way around this would be to use (fully) homomorphic encryption[2] (FHE) or Predicate Encryption[3] (PE). Indeed that would allow some operations on encrypted data to be done server-side without allowing the owner of the server to gain access to any information.

Choices & Compromises

While we cannot guarantee full anonymity for users storing their testimonies (an attacker listening in might be able to see that a user sent data for instance) as not authenticating users would leave us wide open to Denial of Service (DOS) attacks, we must guarantee the integrity and confidentiality of these testimonies.

Considering TOZTI's intended use-case, (F)HE seems too complicated and is more of a research topic than a practical solution: there are neither widely trusted constructions or efficient implementations of these constructions. We therefore decided to forgo it entirely.

The cryptographic protocol we present here might be vulnerable to side channel attacks (depending on the libraries we use), but then again, considering the most general use-case (i.e everything but sensitive testimonies), it does not seem to matter at first (TOZTI is not meant, *as of yet*, to be used to store data so sensitive as to catch the eye of so resourceful an attacker). Still, before we launch TOZTI we will have to do a proper security audit, analysing possible side-channels and implementation issues.

[2]Homomorphic encryption allows the server to compute over encrypted data (without having to decrypt).

[3]Roughly the secret's owner can issue keys that have the ability to evaluate some predicate from the ciphertext.

4.4.2 Handling Permissions

First solution: ACL

The most commonplace way to handle permissions is to use an Access Control List (ACL) based system. The idea is that each ressource will store a list of authorised users. When user Alice wants to access the ressource, her id is checked against the list and she is denied access if her name is not on the list. In particular, anyone may request ask that their id be checked against the list: one need only know the name of the ressource to attempt an operation of which it is the target. On the plus side, this means that we do not set a boundary on accessible objects, however this does leave the system open to a worm, virus, backdoor, or stack buffer overflow. The list itself should be stored in a safe location, in this case it must be stored on the server (which may be problematic with a distributed system). An analogy for this system would be the following: the server is a bank, the ressource is a safe at the bank, and users are the bank's customers. Anyone can walk up to the counter and request to have access, which is enforced by the bank. And the list's integrity is only as good as the bank's owner's, which must be trusted for all transactions (i.e. all requests on the ressource must be approved by the server).

Second solution: C-List

An other system for handling permissions is the Capability List (C-List). In a certain sense, it is the converse of ACL as each users as the "capability" to access certain ressources. So each user knows which ressources he or she can access, and not the reverse. Moreover one must have the necessary capability in order to event attempt to access the corresponding ressource. An analogy for this is the real-life key and lock system. Each ressource is locked behind a locked door, and each user is issued a key for each of the ressources he or she is allowed to access. The key cannot be forged (i.e. an attacker knowing only the shape of the lock cannot forge a key which will open it), however a user may copy the key and give access to another. While this is unclear in the analogy, a user cannot even attempt a request without the necessary anology, thus preventing the aforementioned attacks on ACLs. More precisely, "A capability is a token, ticket, or key that gives the possessor permission to access an entity or object in a computer system" [4] and is classicly implemented in a structure containing an identifier and an access right. An added benefit of capabilities is that they are not based soleley on users: the possessors may be processes and procedures too. This allows for more flexibility and finer granularity.

Our choice for TOZTI

For the permission system, we chose to use a Capability List (C-List) based system. Here are the specific points which ultimately explain how the latter is preferable over the former for TOZTI (it might be worth recalling the bank and lock analogies in order to better understand the pros and cons we discuss here):

- *Minimum server interference:* The server need not verify the requests with C-Lists which fits in perfectly with our general philosophy of not relying on a trusted third party unless absolutely necessary;
- *Transparency:* While ACLs can better guarantee transparency over permissions (authorised users are explicitely listed), the fact that users can freely transfer capabilities to their friends is not of great concern as users can in any case copy and share any ressource they want;
- *Simplicity:* Capabilities work in a very intuitive way and both the users and the administrators are already familiar with the lock-and-key system, which means they are less likely to make security blunders;

However, in the associative world, mandates and responsabilities tend to be relatively short and therefore the authorisations to access given ressources change regularly. Unfortunately, with capability lists, it is tedious to revoke access as this means changing the lock and reforging all keys. Still, we feel that this system's intuitive nature means it is easier for both users and administrators to handle. This alone outways the aforementioned drawbacks.

[4] concept introduced by Dennis and Van Horn in 1966

4.4.3 Encryption Protocols

A general scheme

Notation 1 (Shorthand & Notations)**.**
- *pub_A, $priv_A$ are Alice's public and private keys (for an asymmetric encryption algorithm).*
- *Given a message M and a key χ, define $e_\chi(M)$ the encryption of M using χ (this is a general notation, common to all encryption methods).*

Creating secret data *Case:* User Alice wants to store a message M on the server.

- Alice generates (client-side) a key K in order to encrypt M (client-side).
- Alice sends $e_K(M)$ to the server.
- Alice encrypts (client-side) K with pub_A.
- Alice sends $e_{pub_A}(K)$ to her keychain on the server.

Notice that since the server does not have access to $priv_A$, it cannot efficiently compute K and hence M.

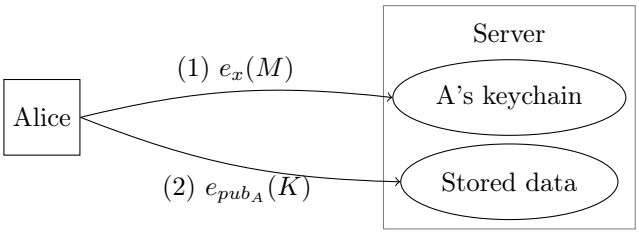

Figure 4.2: Creating secret data

Retrieving secret data *Case:* User Alice wants to retrieve the message M which is stored on the server.

- Alice sends the request to the server.
- The server checks if she is allowed access (via a tag attached to the secret message).
- If she is, the server sends Alice $e_K(M)$ and $e_{pub_A}(K)$ (the latter retrieved from Alice's keychain).
- Alice decrypts (client-side) $e_{pub_A}(K)$ using $priv_A$ in order to obtain K .
- Alice decrypts (client-side) $e_K(M)$ using K in order to obtain M .

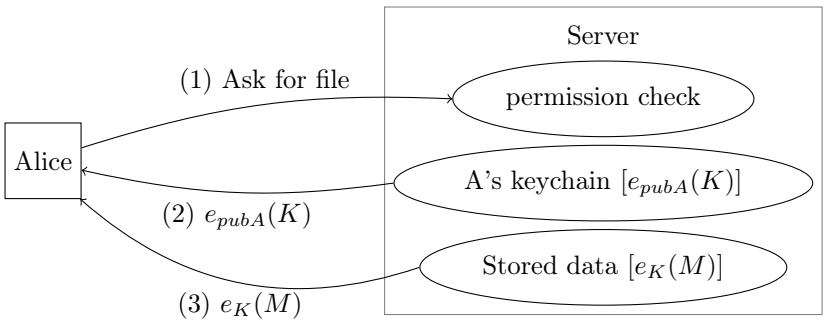

Figure 4.3: Retrieving secret data

Sharing secret data *Case:* User Alice wants to share with user Bob the message M which is stored as $e_{pub_A}(M)$ on the server.

- Alice retrieves pub_B from B's keychain on the server.
- Alice retrieves $e_{pub_A}(K)$ from her keychain on the server.
- Alice decrypts (client-side) $e_{pub_A}(K)$ to get K and encrypts it (client-side) using pub_B .
- Alice sends $e_{pub_B}(K)$ to Bob's keychain.
- Bob now has access to K, hence M .

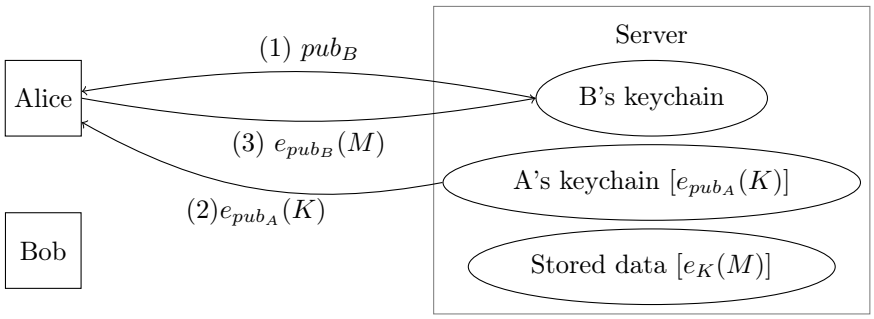

Figure 4.4: Sharing secret data

Group keys This is a generalisation of sharing secret data amongst multiple users. Each group has a symmetric key (which all members share). In order to save memory space on the server, the group can also share a keychain, so as not to copy the group's keys in every one of the group's members keychains.

This approach has two major deficiencies however:

- *Confidentiality:* The group's confidentiality is only as strong as its weakest link, and it is unreasonable to assume that all members of a large group have good security practices.
- *Computation time:* Whenever a member leaves a group, all of the group's keys have to be generated anew. This is computationally long, especially for larger groups of users. Once the system is implemented, and depending on the efficiency of our key generation algorithms, we might have to think of a better way to share a secret amongst a group.

Still, this is the best compromise we have between security and ergonomy.

Implementation recommendations

Caveat Please note that in the above paragraphs, whenever we mentionned a attack by worms, viruses, potential backdoors, stack buffer overflow, denial-of-service, or side-channels, it should be understood that all of these attacks are purely implementation problems. We only mentioned them as they can be used against the system without the proper precautions, but we trust the libraries we use to properly guard themselves against those attacks. We will use widely tested library in order to ensure that.

Possible encryption schemes Our high-level protocol relies on us having some means of performing both asymmetric and symmetric encryption. Here are the algorithms which we plan on using:

- for asymmetric encryption: either Curve25519 (which offers 128 bits of security on elliptic curves, and is good with broken random number generators), or RSA (at least with 2048 bit keys, since under 1024 bits is broken).
- for symmetric encryption: Advanced Encryption Standard (AES) or ChaCha20-Poly1305 (modern and has authentication).

Libraries For the moment, the library we plan on using is the JavaScript version of `libsodium`.

4.5 Our work

4.5.1 Division into workpackages

In order to be more productive, organise ourselves and split the work, several workpackages were created inside of TOZTI. Below we provide a list of the workpackages, their team members, and the work they have done to contribute to TOZTI.

Core (Romain (leader), Joël, Léonard, Peio, Daniel, Pierre O.) Designed the architecture of TOZTI, and implemented the core parts of the server and the client.

Meta (Daniel(leader), Peio, Romain, Lucas E., Pierre O., Guillaume D., Alex) In charge of the technical infrastructure around the project. The main tasks were code review, documenting the inner parts of TOZTI and writing tests. Provided a Docker image and a basic configuration to build extensions. Organized the Git, Javascript and TOZTI extension-writing workshops.

UX/UI (Lucas E.(leader), Felix, Romain, Daniel, Lucas P., Alex, Guillaume C., Vincent, Emmanuel) Created mockups for the user interface of TOZTI. Made sure that the user experience is consistent across all parts.

Demo (Alex(leader), Pierre O., Pierre M., Romain) Delivered the pitch during the public presentation of TOZTI.

Communication (Alex(leader), Peio, Lucas E., Pierre O.) Communicated with associations of ENS Lyon (and of other schools) to understand their needs. Investigated existing solutions in order to understand their advantages and disadvantages.

Storage (Peio(leader), Julien, Daniel, Lucas E., Vincent, Joël, Pierre M.) In charge of designing the storage part of TOZTI. This also includes investigating and researching about ways to allow encryption and permission control.

Calendar (Pierre O.(leader), Guillaume C., Guillaume D., Lucas E., Daniel, Lucas V.) Designed a multi-view (day, week and month) that allows one to add, delete and modify events easily.

Discussion (Felix(leader), Romain, Alex, Felix, Lucas P.) Specified, designed and implemented the discussion board.

Multimedia (Léonard(leader), Peio, Guillaume D., Emmanuel) Designed the basis UI elements for file handling on the user part such as download, upload, consulting file information, etc.

4.5.2 Difficulties

At the beginning of this project, as soon as we decided that we are going to strive for a modular architecture, we realized that there is no point in (and no possibility of) developing TOZTI modules before the core and storage modules were designed, implemented, and ready for usage. Unfortunately, since for logistical reasons not everybody could participate in the design and development of TOZTI's core, some team members felt stalled while waiting for the core codebase to mature. We tried to compensate this by organising workshops in Git and Javascript, where interested team members were able to get acquainted with the technologies we use.

Luckily, this period ended after several weeks, when the communications team compiled the requests received from different associations both inside and outside of the ENS. Having those requests as their reference, the members of the module workpackages have started to design the UX mockups – first on paper, and later as HTML/CSS. Finally the "warm-up" phase has ended roughly two and a half months after the start of the project, when the preliminary work on the core layer was finished both on the server- and the client-side.

Apart from the slow start, we encountered several other problems that usually plague inexperienced teams such as ours. Chief among them was a lack of efficient internal communication. Although we did maintain a Slack chatroom, that turned out to be too frequent for some team

members, so they turned off notifications altogether. This lead to several misunderstandings regarding the division of labor, which could have been prevented had everyone been following the latest developments.

Additionally, we failed to cultivate a culture of strong respect for the internally and externally imposed deadlines, which forced us to organise several "hackathon" sessions, to make up for the lost time. Ultimately however, this helped cultivate the team spirit.

Chapter 5

Conclusion and next steps

Despite the difficulties described in the previous section, we managed to have a successful public demo of our project on March 2nd. There, we presented a fully working first version of TOZTI, complete with the calendar, discussion and multimedia modules, as well as some of the more subtle features, such as taxonomy. The presented design was minimalistic, but otherwise functional, and above all, consistent, thanks to the Vue components that were developed as part of the core work.

Now that we have all but released the first version, the next phase of the work is a big code cleanup and refactor. This is made necessary by the fact that several features were rushed into TOZTI mainline before the demo, with implementations that are suboptimal, but good enough for the time being. Additionally, we can use the experience we gained during these months to reimplement the parts of TOZTI that proved to be difficult to work with, or for which the requirements have changed since the initial design. Maybe the most prominent component that will need to be rewritten is the storage part, since it is quite inefficient in its current form – exactly because of the functionality that was added outside the original specification, in response to the requests from the client-side part of the team.

Once all major components are cleaned up, the TOZTI project will be ready to be deployed at the ENS, and we will be able to accept external contributions. After some amount of testing (and back-and-forth with the users) at the ENS of Lyon, we will feel confident to promote TOZTI to external users. As many of us participate in student associations, we are looking forward to the day when TOZTI becomes the industry standard for managing all the associations we belong to.

Appendix A

Mockups

Figure A.1: A mockup of the directory view.

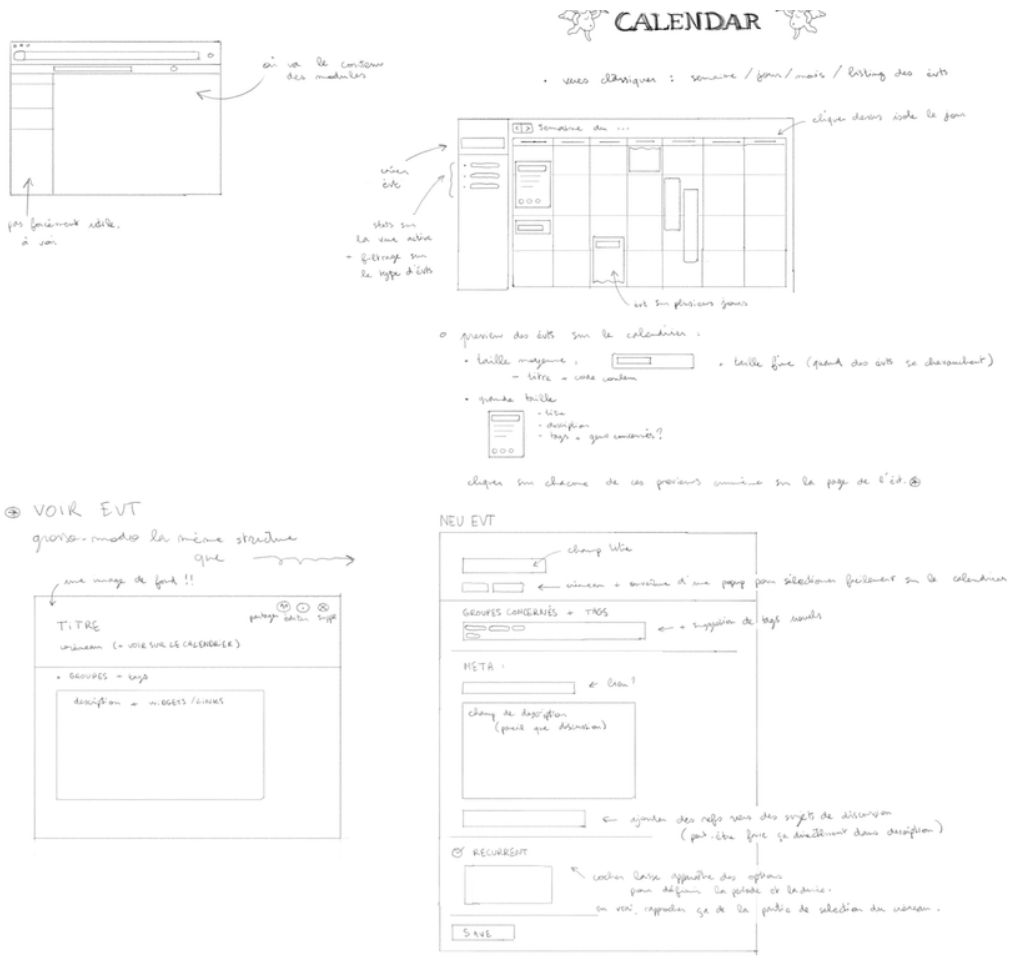

Figure A.2: Mockup of the calendar

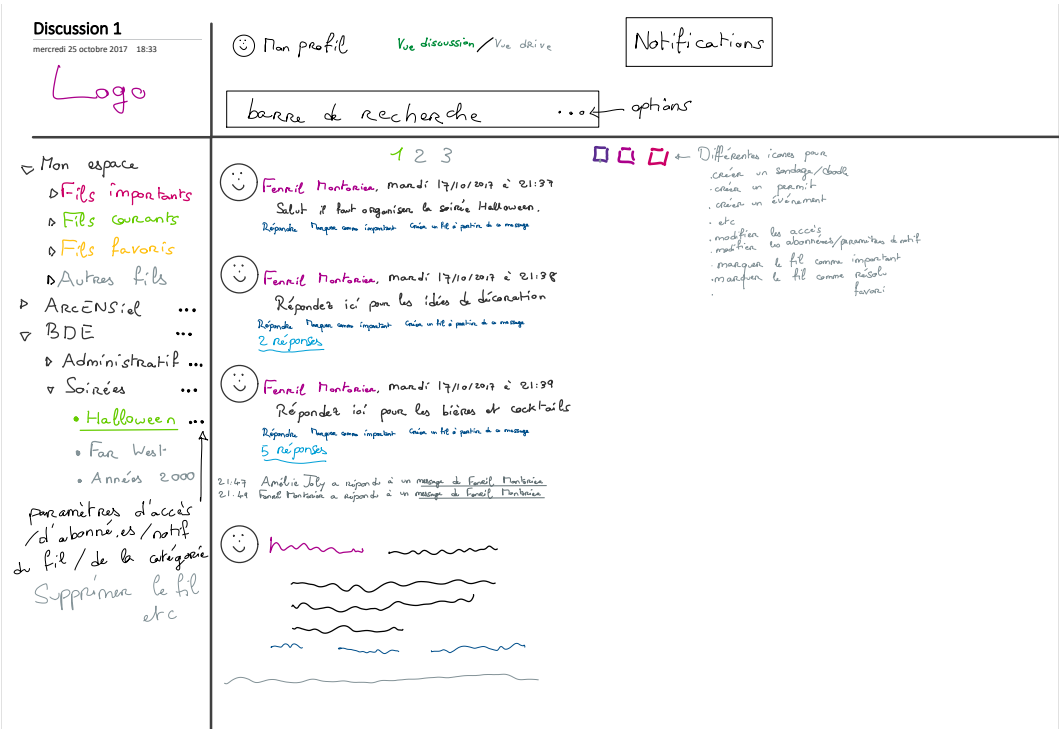

Figure A.3: A mockup of the forum

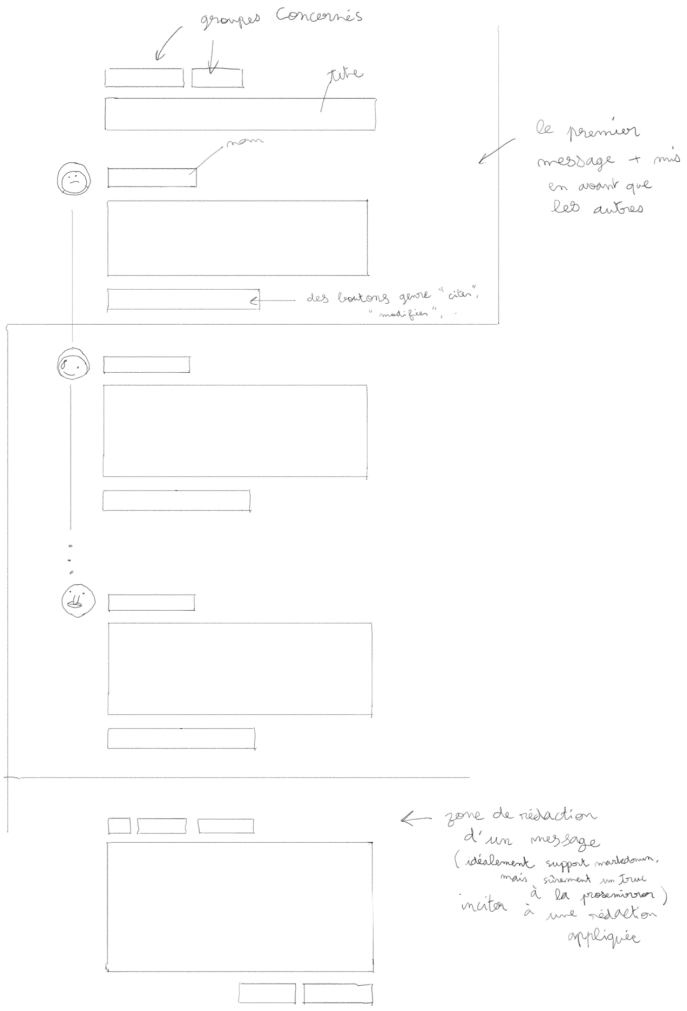

Figure A.4: Another mockup of the forum

Appendix B

Screenshots

Figure B.1: Informative website "tozti.github.io"

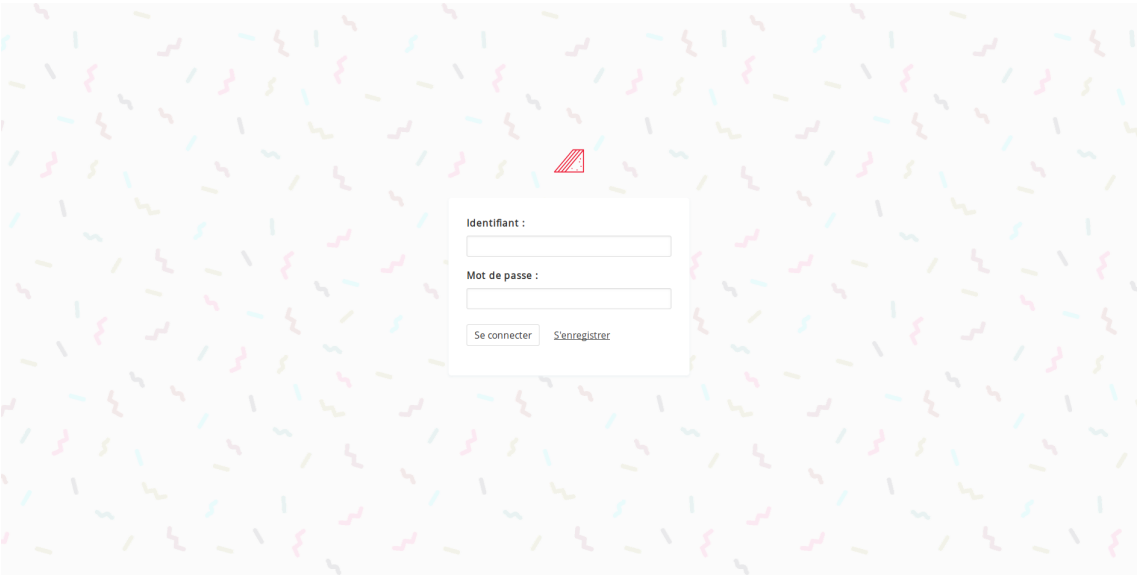

Figure B.2: Login screen

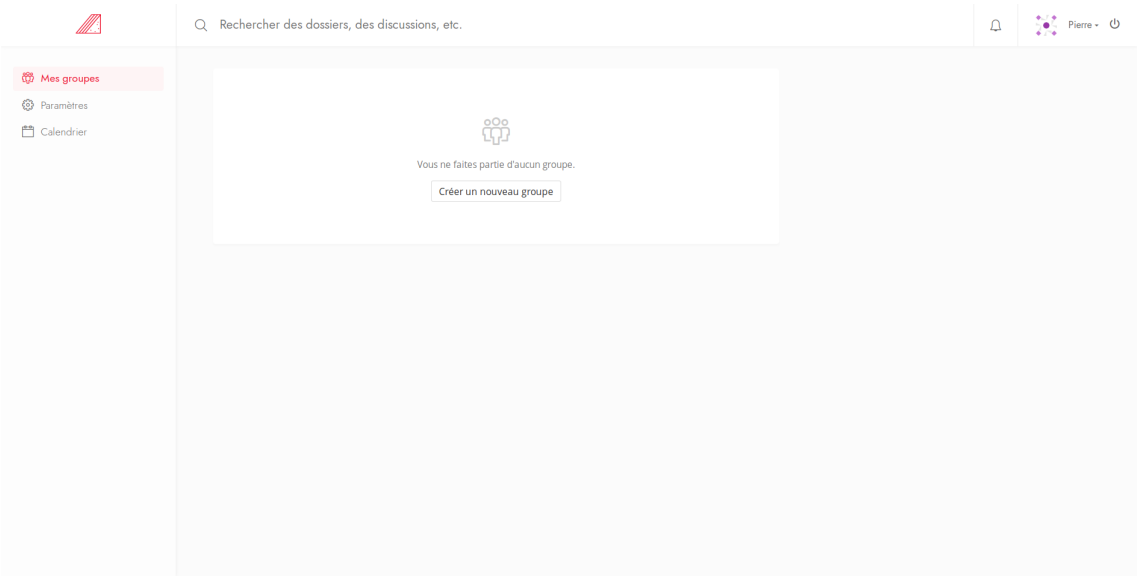

Figure B.3: Group view

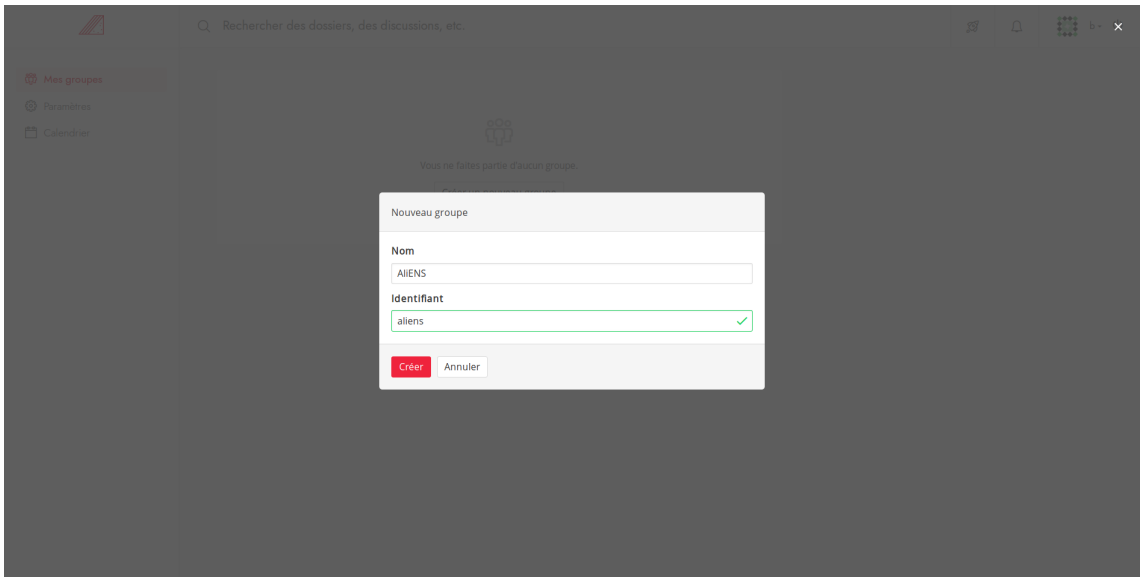

Figure B.4: Adding a new group

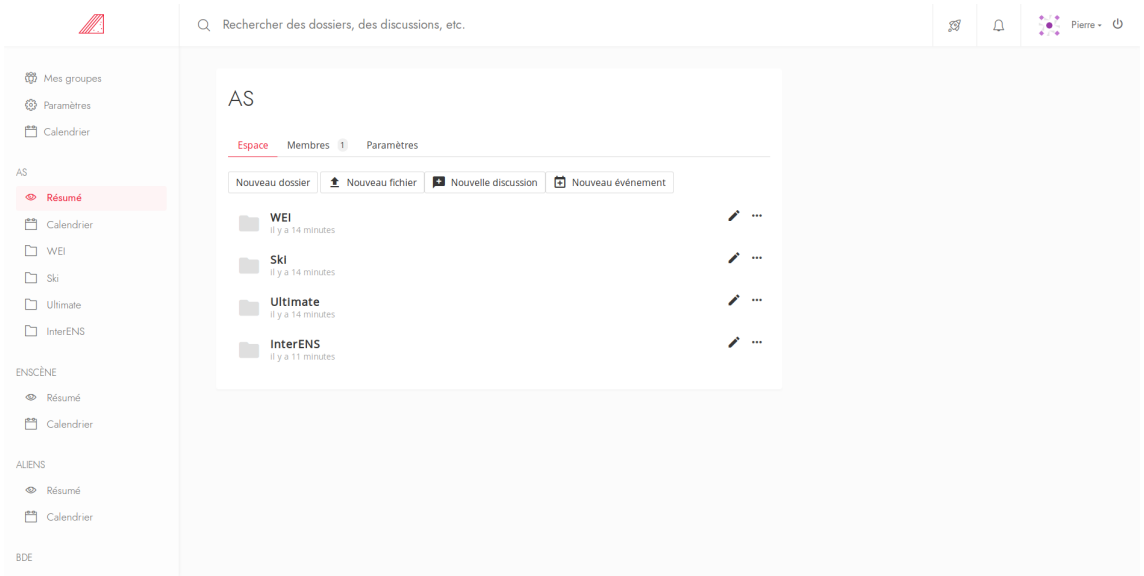

Figure B.5: Folder view

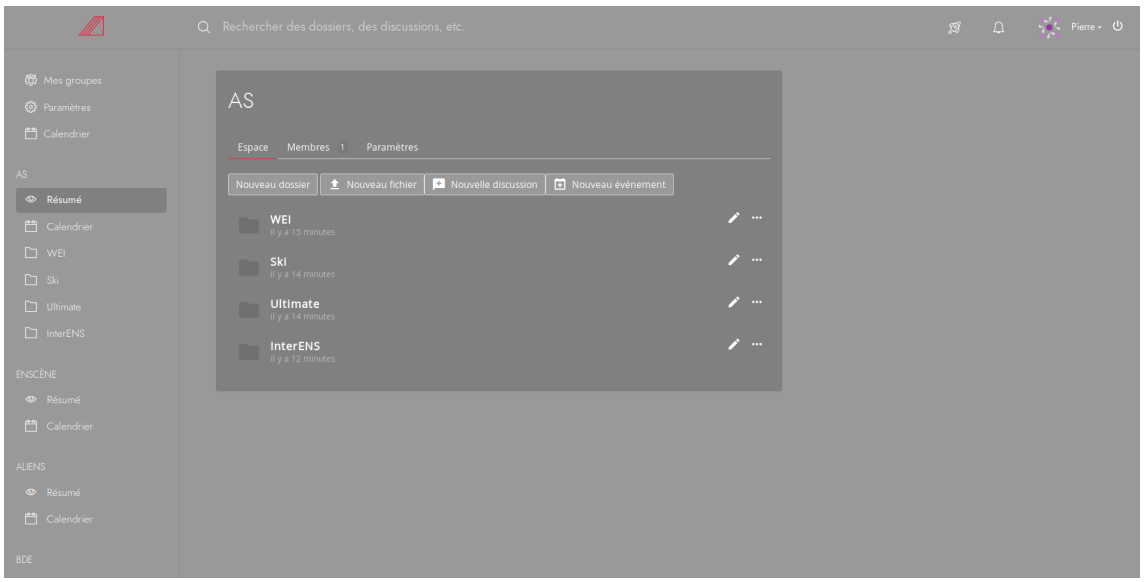

Figure B.6: Night mode

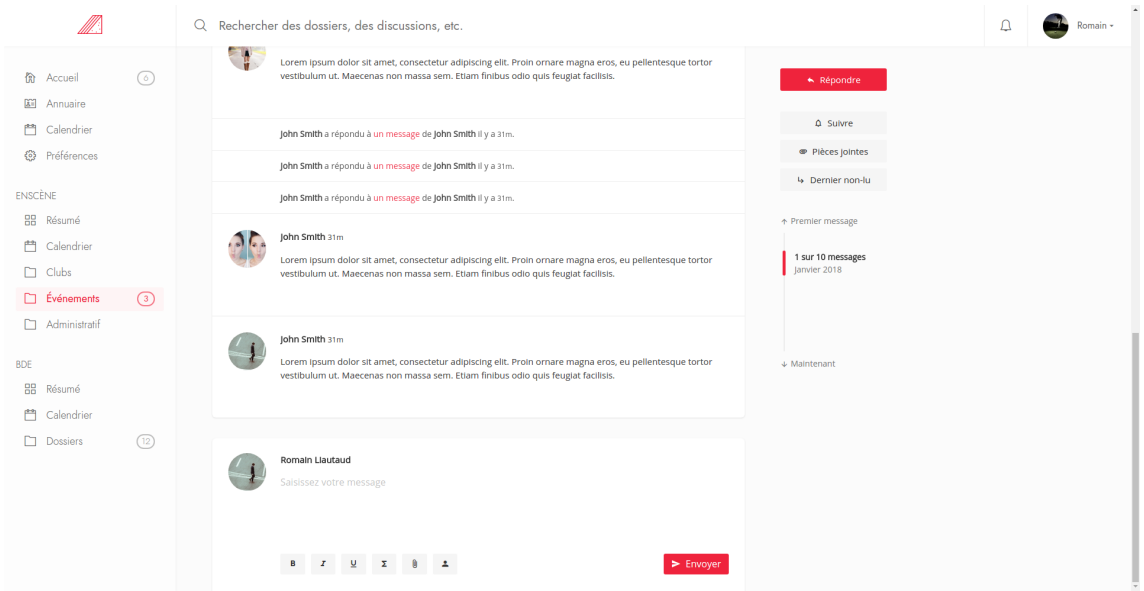

Figure B.7: Discussion

Figure B.8: Calendar day view

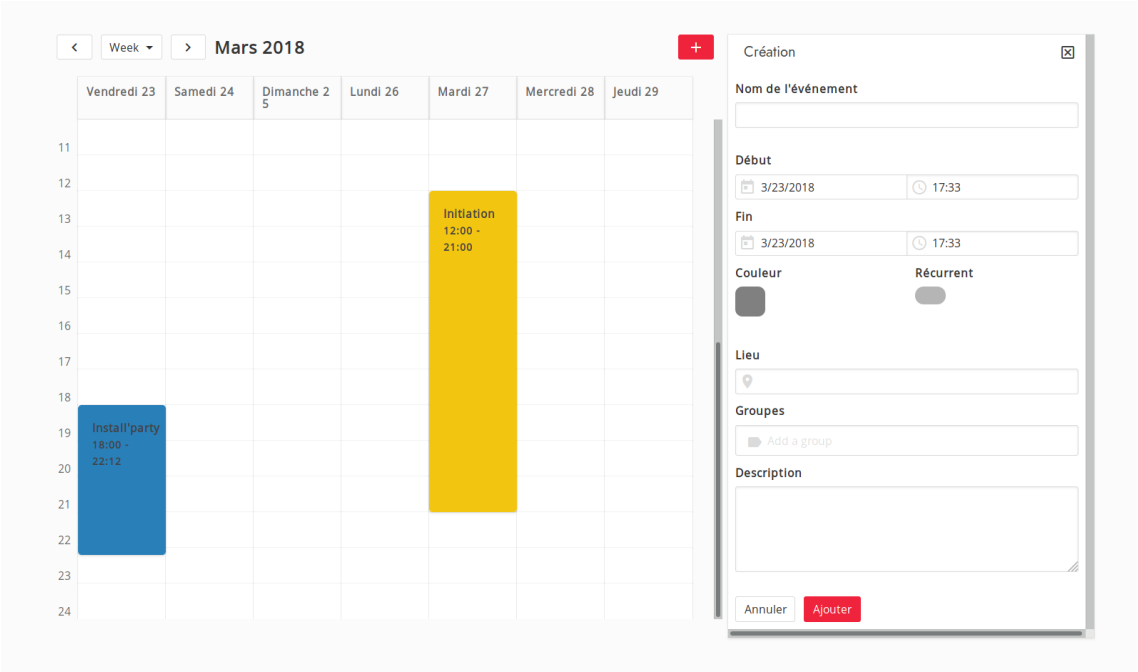

Figure B.9: Calendar week view

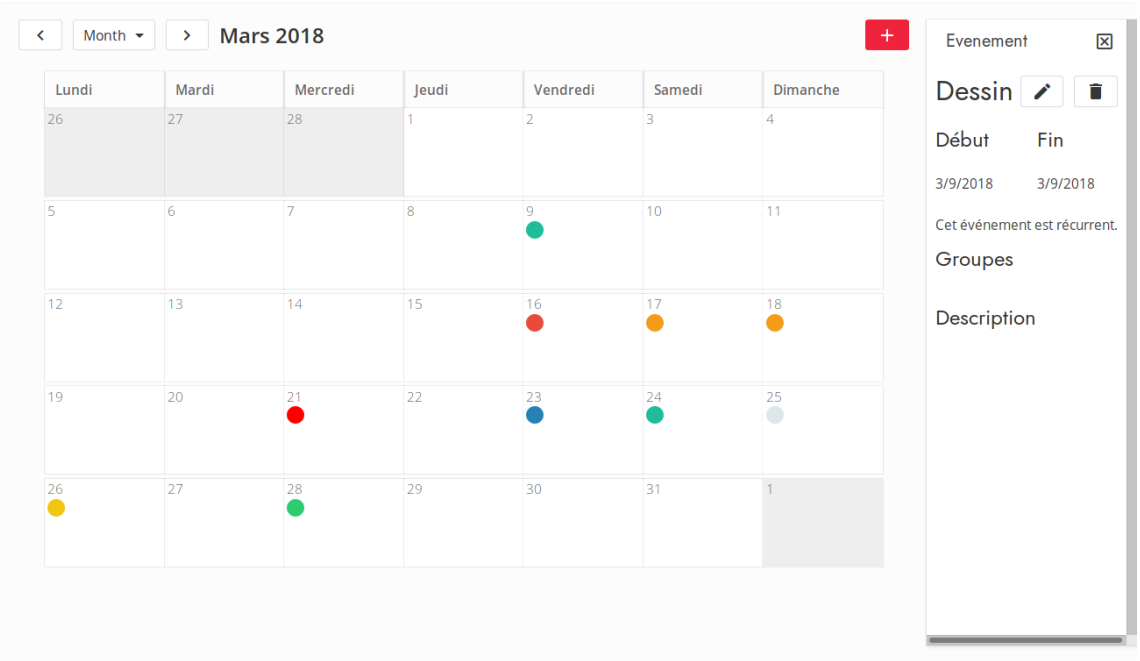

Figure B.10: Calendar month view

Appendix C

Specifications

In this appendix we present the specifications we gathered during our inquiry. Divided into several sections, we present them more or less in their raw form. That is the reason why they are presented in first person. Since the lists were compiled by members of different workpackages, both French and English are used.

Fonctionnalités générales

En tant qu'utilisateur de la plateforme, je veux pouvoir :

M'inscrire et me présenter. Je peux renseigner des informations me concernant, comme mon adresse e-mail ou mon numéro de téléphone.
Je veux pouvoir choisir les utilisateurs ou groupes d'utilisateurs ayant accès à chaque information.
Je veux avoir accès à l'historique des consultations de mes informations.

Recevoir des notifications. Ces notifications m'informent des activités récentes sur la plateforme qui sont susceptibles de m'intéresser. Je peux les consulter en un coup d'oeil sur le site, les recevoir sur mon téléphone (ou mon ordinateur) en push, ou encore les relayer sur Facebook (voir ici). Une fois lues, je ne veux plus qu'elles soient en évidence.
Je veux choisir quel type de notifications je souhaite recevoir.

Être notifié lorsque l'on me mentionne.

Être notifié lorsque l'on répond à un de mes messages dans une discussion.

Être notifié lorsqu'il se passe quelque-chose dans une catégorie ou une entité à laquelle je me suis abonné.

Rejoindre des groupes d'utilisateurs. Ces groupes correspondent généralement à des bureaux d'associations ou de clubs.

Créer ou rejoindre des espaces de travail. Chaque espace de travail est accessible à un ensemble de groupes et d'utilisateurs donnés.
Souvent, une association ou un club aura un unique espace de travail, mais on peut envisager des cas dans lesquels plusieurs associations souhaiteraient un espace de travail commun pour l'organisation d'un événement (typiquement le BDE, l'AS et Enscène pour le WEI) ; ou des cas dans lesquels un groupe d'utilisateurs souhaiterait superviser un ensemble d'espaces de travail (typiquement le bureau restreint du BDE pour les espaces de travail de chacun de ses clubs).

Module Discussion

Au sein d'un espace de travail, je veux pouvoir :

Démarrer une discussion. Une discussion est une suite de messages, et doit comporter un titre et un premier message.

Consulter les messages d'une discussion. Ces messages sont triés par ordre chronologique et paginés selon un nombre de messages par page configurable.
Ce nombre de message prend seulement en compte les messages "à la racine".

Aller directement au dernier message lu. Lorsque j'ouvre une discussion que j'ai déjà consultée par le passé, je veux pouvoir aller directement au dernier message que j'ai lu.

Ajouter un message à une discussion. Pour écrire ce message, je veux disposer d'un éditeur WYSIWYG (similaire à celui du webmail) qui gère les mentions, l'intégration de liens externes et de liens internes.
Pour mentionner un utilisateur, je veux taper `@` puis commencer à saisir son nom, et pouvoir choisir parmi les utilisateurs de l'espace de travail.
Pour l'intégration de liens externes, je veux pouvoir saisir mon lien et qu'il soit automatiquement reconnu et remplacé par un aperçu de la page vers laquelle il pointe.
Pour l'intégration de liens internes, je veux pouvoir saisir un lien vers une URL de Tozti, et qu'il soit automatiquement reconnu et remplacé par une référence à (et un aperçu de) l'entité qu'il désigne. Je veux aussi disposer d'un champ qui me permette de rechercher rapidement une entité et d'insérer un lien interne vers celle-ci.

Mettre un message en évidence. Ces messages sont affichés d'une façon qui les fait se distinguer, au premier coup d'oeil, des autres.
En haut de chaque discussion, je veux pouvoir déplier une liste de tous ses messages en évidence.

Répondre à un message. Je ne peux répondre qu'à un message "à la racine", i.e. je ne peux pas répondre à une réponse.
Toutes les réponses à un message sont affichées sous celui-ci, avec une légère indentation. Ces réponses, à l'exception de celles mises en évidence, sont cachées par défaut, et sont dépliables sur demande.
Pour maintenir une cohérence chronologique, un lien vers la réponse est affiché "à la racine", à l'endroit où le message aurait été placé s'il n'avait pas été une réponse.

Migrer un message et ses réponses vers une nouvelle discussion. Lorsque le nombre de réponses à un message excède une limite prédéfinie, on me suggère de migrer le message et ses réponses vers une nouvelle discussion.
Cette discussion contient une copie du message et de toutes ses réponses à la racine. Elle est créée au même niveau dans l'arborescence que la discussion du message original, et je veux pouvoir en choisir le titre.
Le message original demeure dans sa discussion, mais toutes ses réponses y sont supprimées. À la place du bouton pour déplier les réponses, j'ai une indication clicable que "ce message et ses réponses ont été migrés vers une autre discussion

Citer un message. Je peux sélectionner tout ou une partie d'un message, et le citer dans un de mes messages.
Je peux cliquer sur une citation, et être amené au message original.

Modifier les messages que j'ai écrits. Comme pour les réponses, pour maintenir la cohérence chronologique, un texte vers le lien du message modifié s'affiche à la racine du fil.

Savoir si un message a été modifié et consulter l'historique des modifications d'un message.

Module Calendrier

As users of the Calendar module, we would like to:

Add an event. We want to be able to create events.

Edit an event. We want to be able to edit an event after its creation. The changed made must be visible to all persons who can see it.

Add a recurring event. We want to be able to add event that appears periodically.

Instantiate a particular event of a recurrent event. Sometime a specific event part of a recurrent event must be changed. We can convert this instance to its own event and apply the modifications to it.

Give tags to an event We want to be able to give tag(s) to an event. The tags are used to classify and categorize the events, and can help to delimit the scope of synchronisation. Tags can be recursive. Each events should have at least one tag (it can be simple, like "bde").
Some tags (like "soirees", "AG") are preexistent and are used to make classifications of typical events easier.
Example:

- An event can have the tag "bde.wei.administration" and will only be pertinent to the member of the bde who are part of the organisation of the wei and who are doing administrative stuff. An event "bde.wei" is there for all members organizing the wei in the bde.
- An event could have for tags "bde.wei" and "as.wei", meaning it impacts both the members of the bde and of the as parts of the wei
  A Tag is mainly an information used to classify events. But it can also be used to specify the visibility of an event. To a tag we can specify which users are able to see the event, and which one are able to edit it.

Associate groups of people to an event. We can associate certain people to an event. This can be done either automatically, by the tags (a tags represents a group of persons), or either manually by specifying some persons when creating / editing an event.

Filter events. We wants to be able to filter events according to different criterions. These criterions can be:

- Their tags
- Their date
- The users involved in them

Specify if we want to be notified for an event. The user should be able to specify if he wants to get a notification when the deadline for an event is near (and at which frequency). He should also be able to decide if he wants to get a notification when an event is modified (for example the locations changes).

Words about synchronisation

- Synchronisation with external calendars services.
  We want to be able to synchronise some events with gcalendar, framagenda, ...
- Send an event other a media.
  We want to be able to send (or initialise) an event on an other media (Mails, Facebook, ...)

What is an event?

Entry	**Description**
Title	Title of the event
Date	Date (and time) on which the event occurs
Duration	Duration of the event. Can be changed with date of end
Description	Description of the event. Can be some formatted text and includes "links" to other sources
Location	location of this event
Period	period of time between two events if it is reccurent
Tag(s)	Tag(s) associated to this events
Persons	List of persons associated with the event

Example of usercase:
Shared calendar for La Festive availability:
To create a share calendar showing the availability of the room "la festive", a user could:

- create a tag where:

  - Everyone (i.e. some associations) can see the events
  - Only him can add one

- create an event associated to this tag to indicate that la festive is not available during this time

Module Multimédia

Send a file (PDF, etc.) We want to be able to send files without using another interface (Drive, Dropbox, etc.)

Sharing We should be able to select a group or persons that will have access to this file, and will be notified of it being available. The notion of groups, and therefore of a generalized contacts feature is important if we want to avoid having people enter entire groups of names by hand. Otherwise it is the same a WeTransfer (that allows to share a file to 20 persons simultaneously)

Change the version of a file If I modify a file that I previously shared, I should be able to replace it without re-sending it to everyone. They will be notified (or not) that there's a new version.

Give tags to a file In order to find it easily and explain its content a bit (most files have absurd names)

Delete a file Should be subject to caution. Some association files are a bit sensitive (STATUTS, etc.). Rights to do so should probably be defined beforehand.

Accessing all the files I have access to Without it being too constraining, like having to load a whole new page.

Filtering the files I have access to. Differents ways of doing this:

- chronologically
- by trending order - would be useful to work on the file that is being used most in your groups right now
- by types - for example to find all the PDFs

Searching among these files. Searching by name, tags, etc.

Structure the files into directories Most associations, like Enscène, use a Dropbox for their important common files, but are not very pleased with it.

Ask a file from someone Ask a file (e.g. a poster for a party) from a group of people able to produce it (e.g. the Respo Graphisme). This will notify them every n day that they need to make it, give them a deadline, and a "ghost" file should appear in the right folder to remind everyone that such a file will be created.

Bibliography

[1] FIDO Alliance. *Universal 2nd Factor 1.2*. Tech. rep. FIDO Alliance, 2017. URL: `https://fidoalliance.org/download/` (visited on 01/14/2018).

[2] Arnar Birgisson et al. "Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud". In: *Network and Distributed System Security Symposium*. 2014.

[3] M. Pei D. M'Raihi S. Machani and J. Rydell. *TOTP: Time-Based One-Time Password Algorithm*. Tech. rep. Internet Engineering Task Force, 2011. URL: `https://tools.ietf.org/html/rfc6238` (visited on 01/14/2018).

[4] F. Denis. *libsodium 1.0*. 2017. URL: `https://libsodium.org/` (visited on 01/14/2018).

[5] *JSON API Specification 1.0*. 2015. URL: `http://jsonapi.org/format/` (visited on 01/14/2018).

[6] A. Melnikov. *Salted Challenge Response HTTP Authentication Mechanism*. Tech. rep. Internet Engineering Task Force, 2016. URL: `https://tools.ietf.org/html/rfc7804` (visited on 03/23/2018).

[7] *Vue Router website*. URL: `https://router.vuejs.org/` (visited on 01/14/2018).

[8] *VueJS website*. URL: `https://vuejs.org/` (visited on 01/14/2018).

[9] *Vuex website*. URL: `https://vuex.vuejs.org/` (visited on 01/14/2018).