UCLouvain

École polytechnique de Louvain

Scalable and extensible social networks

Author: **Adrien WIDART**
Supervisor: **Etienne RIVIERE**
Readers: **Kim MENS**, **Guillaume ROSINOSKY**, **Raziel CARVAJAL GOMEZ**
Academic year 2019–2020
Master [120] in Computer Science

Acknowledgements

I would like to thank my supervisor, Etienne Riviere, for his help and availability during this year-long work. One of his courses, "Cloud Computing", is behind the choice of this thesis and the field in which I like to exercise my knowledge.

This thesis allowed me to work on a consequent project and to be able to handle each step: documentation, reflection, design, development and testing. Not only did I have the opportunity to discover new software engineering techniques, but also to apply them.

I also want to thank the other members of my jury, namely Kim Mens, Guillaume Rosinosky and Raziel Carnajal Gomez, for the time spent reading this work and for their presence during my presentation.

Abstract

Leading social networks bring together millions of users, but they are horizontal: the functionalities are basic and do not serve a specific social community. Moreover, they are centralized: the data generated are in the same place, which can present a risk (leakage, exploitation...). Lastly, the architecture of these large applications is monolithic, i.e., built in a single block. It is, therefore, complex to modify, add, and even share social network services. Scaling such an architecture is not always done efficiently.

We propose the DEVI approach for the development of vertical, modular, and interoperable social networks. The new approach promotes the use of modern software engineering techniques: microservices, CQRS (Command Query Responsibility Segregation), event sourcing...

The social networks become vertical: they offer several features to help a specific social community. The use of microservices allows each feature to be developed as an isolated module. These modules can then be easily reused. This architecture also enables tailor-made scaling.

Decentralization allows several instances of a web application to exist and communicate together. Moreover, with the DEVI approach, these new platforms are part of the Fediverse: a federation of servers with different social networks that can communicate with each other. In order for each instance to communicate easily with each other, the DEVI approach uses the W3C's ActivityPub protocol.

The thesis will propose the implementation of a new social network. This proof of concept will be a social network for the French-speaking Scouts Federation of Belgium. The development will be done using tools such as Node.js, MongoDB for NoSQL databases, or Docker to run microservices within containers and facilitate scaling. The proof of concept will show that it is possible to create, thanks to the DEVI approach, a Decentralized, Extensible, Vertical, and Interoperable social network.

Contents

Chapter 1

Introduction

The Internet has become an essential part of everyday life. In recent years, social networks are more present than ever before and represent today a significant part of the Internet. Most users are familiar with massive platforms which count millions of members such as Facebook, Twitter or Instagram... However, these large social networks do not seek to offer services for a specific social community. They are called horizontal social networks. Indeed, they include basic functionalities (post, like, follow...) so that no Internet user is excluded. As a result, everyone may find an interest in registering on such a social network, but the latter will never provide a significant utility for each of its members.

Additionally, leading social networks may have some weaknesses. For example, these web applications are centralized: all members' data are stored in the same place. A platform that gathers an important number of accounts will then become an information monopoly. Besides, the code of these applications is not public, so users do not have any information on data management. Excesses and problems then become possible, such as the exploitation of private data or data leakage.

Finally, these platforms are often rigid applications based on a monolithic architecture. This structure type makes it complex to modify or even add new features. It also becomes difficult to share components of the architecture so that they can be reused in other projects. Furthermore, another consequence of developing in a single block is poor scaling. In case of heavy use, the entire application will have to be scaled, even if this traffic overload only concerns a specific functionality.

In this thesis, we would like to create a new approach to develop social networks. Each platform would be vertical, i.e., each one would answer to the needs of a smaller and specific social community. The question is then the following: "How should we engineer these vertical social networks?" Moreover, we would like to promote code reuse, to offer interoperability between all social networks, and even to give the possibility to scale each platform component independently.

1.1 New protocol and software engineering techniques

It is possible for a social network to be decentralized. This means that several different servers can host the application. A server becomes an instance of the social network. Users can then choose to register on one of them, instead of finding them all on a single instance. It is also possible to register the social network within a federation. Several different social networks can then connect and communicate with each other. Such a federation already exists and is called Fediverse. Besides, all social networks and instances must be able to communicate. ActivityPub is an emerging protocol that enables this communication.

New software engineering techniques also exist, and each one can respond to a specific problem. For example, an alternative approach to architecture is to divide an application into a multitude of different components. These are called microservices. These components are smaller applications that can work in isolation and autonomously. Then, the different microservices need to be connected so that they can interact and create the entire application.

CQRS stands for Command Query Responsibility Segregation and describes a new pattern that can be used in the development of an application. It consists of completely separating the write and read parts of an application. Commands are used to write new data and are based on actions specific to the application domain. Queries are used to read data from the application. The read and write models are also different depending on the needs.

The event sourcing is also a pattern and offers a new approach in data storage. This technique proposes to store the sequence of all the events that change the application state. These high-level events are related to the application domain. This way, a store has the whole history of the platform.

1.2 Ambitions

The goal is to offer an alternative to the traditional development of social networks. More precisely, this thesis aims to create a new approach to the development of social networks. Therefore, the generated social networks will have new qualities. First of all, the simplest one: the development will be open so that any user can be aware of the internal operations. Then, they will be vertical: each application will target a specific social community and offer functionalities useful for that community. Social networks will also be decentralized, which will put an end to information monopolies and enable the creation of smaller community-specific platforms. These platforms will also have much better scalability and extensibility. Adding new functionalities will be an easy task, and the application will cope with

higher traffic loads more appropriately.

It is hence necessary to combine the new software engineering techniques previously listed. This combination will constitute the DEVI approach, i.e., a new approach to develop Decentralized, Extensible, Vertical, and Interoperable social networks.

1.3 Plan

The thesis is composed of two main parts. The first one presents the creation of the DEVI approach. To do so, the next chapter examines in more detail the problems addressed by this thesis, and the solutions that already exist. The chapter will help to set out precisely the objectives that the new development approach must achieve. A second chapter analyses the new approaches and techniques introduced by the section 1.2. This analysis will identify how they work and what they can bring to the application.

The second part focuses on the implementation of a social network based on the new approach. First, this thesis proposes to realize a proof of concept through a social network for the French-speaking Scouts Federation of Belgium. It will, therefore, be necessary to establish the different services appropriate for such a community, but also an architecture to arrange them correctly. Further explanations will identify the possible interactions with each of these modules, as well as the way they work. Then, the next chapter will list and explain all the tools used for the implementation.

Chapter 2

Problem

This chapter will study more deeply the problems that we will treat throughout this thesis. Indeed, the implementation of a significant part of current social networks is based on an approach that has become standard: centralized data, functionalities aimed at attracting a broad audience and a monolithic development. But is it possible to determine a new approach to develop alternative social networks? Can new software engineering techniques help to establish a new way of developing these platforms?

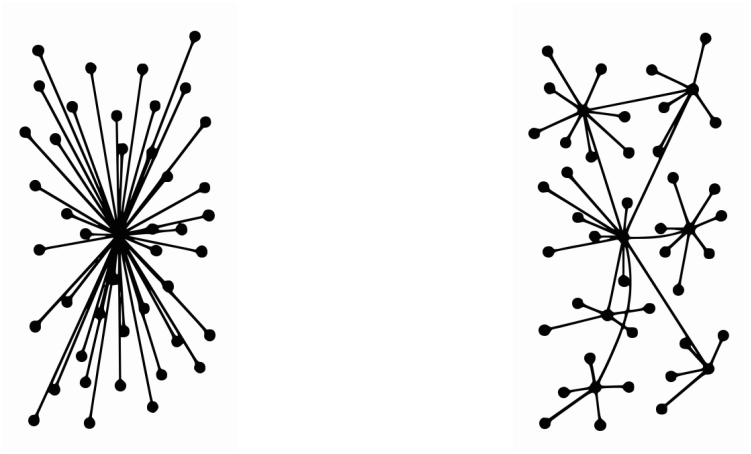
2.1 Towards a new approach to social networks development

As a first step, it is necessary to define the different characteristics of a new approach to the development of social networks.

2.1.1 Decentralized

The first point of this new approach is the decentralized nature of the social network. In this structure type, the social network is composed of several servers, and the users are distributed among them. Such a server is called an instance of the social network. In addition, these instances can communicate with each other. As a result, the global operation of a social network would depend on multiple servers. All the connections, all the exchanges, all data generated would not necessarily accumulate in the same node of the network. Moreover, in a decentralized social network, users would even be able to choose their instance, i.e., where their data is processed and stored.

Decentralization also refers to the ability of different social networks to communicate. Indeed, it is possible to create a federation of several independent but



(a) Centralized (b) Decentralized

Figure 2.1: Network structure[1]

connected platforms. As a result, users of two different social networks are still able to communicate with each other.

Figure 2.1 shows the difference between the two network organization. In Figure 2.1a, each of the network users interacts with the same node. This causes not only a dependency on the node (if it fails, the whole network is down), but also a concern for information monopoly (a single node records all data). Figure 2.1b illustrates the decentralized network organization. Several users are connected to the same node in the network, creating a functional and autonomous group. However, these groups are also interconnected and create a more extensive network. It is then possible for a member of a group to exchange with a member of another group. Besides, data can then be stored at different nodes in the network, so none of them can become a monopoly of information.

2.1.2 Vertical

We can easily observe that the social networks we are familiar with offer some general-purpose services. They are horizontal social networks. Of course, they often have one specific feature (share an opinion in a few characters, promote photo sharing, share short video captures, share events, etc.). However, this characteristic remains very general so that the size of the potentially interested public is not restricted. Therefore, we can consider these social networks to be useful, but in a very general way: they do not meet specific needs that would significantly increase the interest of a smaller proportion of users.

[1]From `https://flic.kr/p/7Gx3wv`, by Paul Baran under licence CC BY-NC 2.0

On the opposite, some other existing social networks are vertical. The idea is no longer to offer a general functionality to a broad audience, but rather a suite of different services to a particular social group. Each social network then becomes much more specialized in one domain and more useful to its users. For example, we might want to develop a social network for musicians in Belgium. This one would link them, but in addition to that, it would offer services such as recording studio rental, a tool for putting the tracks online, a schedule to determine rehearsal dates, etc. Another example would be a social network for teachers, where they could share teaching tools, organize events between several schools, etc.

2.1.3 Modular and extensible

Thanks to the previous subsection, we know that a social network can offer particular services. Let us now consider a third example: a social network for football clubs. Again, it could provide several services: photo-sharing tool, online scoring, etc. Developers might also want a tool to plan training sessions. This last service is very similar, in the previous point, to the service allowing musicians to schedule their rehearsals.

Therefore, modularity becomes an essential point in the new approach to social networks. In this new approach, the social network is no longer developed according to a monolithic architecture, i.e., an architecture consisting of a single block where the functionalities are interconnected and interdependent. Indeed, the modularity of the structure means that each service is developed and operates in a completely independent way. This modules' isolation then provides the advantage of not having to reinvent the wheel when developing a new social network (in the example above, the social network of football clubs can reuse the module allowing musicians to organize their rehearsals). Besides, this new structure indirectly makes the social network much more extensible. Indeed, if at some point the desire is to add new functionality, then developers implement the appropriate new module (if it does not already exist) and connect it to the general structure of the social network. They don't have to make changes to the other services.

2.1.4 Interoperability

According to the previous points, developers might want to import a module into an existing social network. Users would have to communicate with this new module. So this raises the issue of interoperability: if when importing a new module, this one uses a different communication protocol than the social network, it creates a problem. The developers' team will then have to find a solution: review the protocols used within the system, adapt a protocol specific to this new module, redo the protocol of this module... In any case, this will lead to an additional workload.

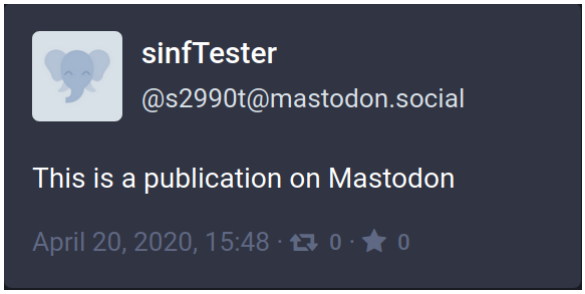


Figure 2.2: Publication on Mastodon

Moreover, if several modules use different protocols, it makes the architecture and the internal functioning of the platform more complex. In the new approach to social network development, the interoperability of a module will, therefore, be relevant.

The same will be valid for communication within a decentralized social network. We discussed the case of a social network composed of several instances. If, when creating a new instance, developers decide to create a new protocol, this makes interoperability between this instance and all the others more complex. As a result, it will also create some additional work since the developers will have to adapt all communications. Therefore, the notion of interoperability will also become essential to avoid this increase in complexity and adaptation difficulties.

2.2 Current situation

New approaches to the development of social networks already exist. Each one takes up one or more characteristics proposed in the previous section. In addition to these existing solutions, studies continue to search for and submit new solutions. Last year, a thesis was already looking at a new approach to develop social networks. This work proposed a possible foundation for scalable and decentralized social networks.

2.2.1 Some examples

Alternatives already exist on the internet. Some try to compete with leading social networks, and others try to offer more specific services.

Mastodon

The social network Mastodon[2] is a Twitter-like platform. Figure 2.2 shows a publication on the platform. The similarities with Twitter[3]are immediately noticeable. If it is a horizontal social network, it is still an alternative version based on a new approach. Indeed, Mastodon is a decentralized social network, i.e., several instances exist and are interconnected.

In the Mastodon case, a new user can choose the server that suits him[4], and he will still have the possibility to communicate with the users of the other instances. To go further, it is also possible for a user to host his instance. Thus, Mastodon proposes an alternative way to use social networks, avoiding the creation of an information monopoly. Besides, Mastodon is a free (and advertising-free) and open-source social network that continues to work: in May 2020, there were around 37,000 active users on the instance directly created by Mastodon[5].

Behance

Behance[6]is a social network owned by Adobe Systems. Although centralized, this social network is also an example since it stands out for its vertical nature. Indeed, Behance focuses mainly on visual artists (photographers, designers, etc.). They can then register to share content that can be commented on, or "appreciated". It is, of course, possible for a member to follow the artists he or she likes. The social network also offers various services to its users. For example, a member can create an art project. It specifies the guidelines, the programs to be used, the goals... Afterward, other members can then join the project in order to give advice and help. As shown in figure 2.3, Behance also offers a service dedicated to job offers: members can submit paid work, add conditions, discuss with a potential artist, etc.

HumHub

This last example is not a social network but rather a development tool for social networks. HumHub[7]is a free and open-source tool that aims to simplify the developers' work. The platform offers a base with the possibility to extend it. Indeed, during the creation of the social network, the creators set the different options, i.e., the modules needed, and HumHub will provide an implemented solution. This tool thus directly fits the "extensible" approach of social network development. However, the result obtained is based on a monolithic architecture

[2]`https://mastodon.social/`
[3]`https://twitter.com/`
[4]The official website provides a list of possible instances: `https://joinmastodon.org/`
[5]Information can be directly found on the site of the social network
[6]`https://www.behance.net/`

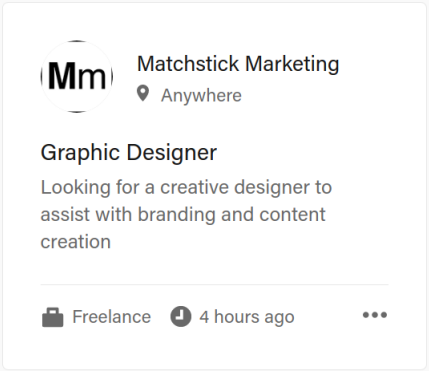


Figure 2.3: Preview of a job offer on the social network Behance

written in PHP. Therefore, due to the lack of modularity, it will not necessarily be easy to make changes (add/remove features, update an existing one, etc.) later on. Also, HumHub does not provide any decentralized solution. So, this development tool lacks support for interoperability with other social networks.

2.2.2 A first part of the solution

While examples are already implemented and deployed on the internet, research also continues to suggest different ways to develop social networks. Last year, a thesis[8]proposed the foundations of an extensible, decentralized and open social network.

The result of this work is a platform with the general functionalities of a social network: it is possible to post content, to share it, to like it, to follow a person, or to block it. However, the interest of the thesis concerns rather the structure of the project and the techniques used to develop it. Indeed, the architecture of the code is very modular and extensible. This allows us to take over a part of the project, but also and especially to reuse the project to add different modules. This social network also respects the conditions and protocols to form an instance within a more extensive set of social networks called the Fediverse[9]. It is then possible to create several instances that can communicate with each other but also with instances of other social networks. For example, one of the social networks within the Fediverse is Mastodon. As a result, members of the social network created can communicate with members of Mastodon. This last point justifies the decentralized nature of this new approach.

[7]`https://www.humhub.com/`

[8]Written by Gary Lovisetto under the supervision of Pr. Etienne Riviere and available in open-source: `https://github.com/glovise15/MasterThesis`

[9]`https://fediverse.party/`

Characteristic	Description
Decentralized	Several instances can be interconnected
Extensible	The architecture of the social network makes it easy to add services. To do so, the services are developed in a modular way, i.e., they are isolated from each other
Vertical	The social network concerns a specific domain and offers services related to it
Interoperability	Each module follows the same communication protocol

Table 2.1: All the characteristics of the DEVI approach

2.3 Objectives

All previous examples are based on a new approach to develop social networks. However, although each one answers to a specific identified problem, none of them presents all the characteristics at the same time. Mastodon is a decentralized social network, but it is not vertical. On the other hand, Behance presents the opposite situation: it is vertical but not decentralized. Although HumHub offers a solution to the extensible aspect of a social network, the generated platforms are developed according to a monolithic architecture and are not interoperable. These few examples illustrate a more general situation that deserves further study so that a social network can have all the characteristics.

The starting point for this thesis is based on the foundation created last year and presented in subsection 2.2.2. However, this foundation offers a horizontal solution. The current thesis will study the possibility of adding modules that provide functionalities for a specific social community. If these modules are correctly attached to the foundation, the result will be a vertical social network. Of course, the development of these modules and the result obtained must respect all the previously announced characteristics.

The objective of this thesis is, therefore, to establish a new approach to develop social networks that respect all the characteristics listed in section 2.1 and that Table 2.1 summarizes. We will call this new approach "DEVI", for Decentralized, Extensible, Vertical and Interoperable social network.

Chapter 3

Foundations for innovative social networks

The previous chapter identified not only the problem, but also the foundation of the solution, and the different characteristics we would like to respect. It is now interesting to look at existing protocols and techniques. We will discuss some distributed systems techniques such as the Fediverse for the decentralization issue, and the use of ActivityPub, a communication protocol, for the creation of interoperable APIs. We also need to explore new software engineering techniques. For example, a microservices architecture makes it possible to isolate modules from each other. We will also see how the modularity of the architecture can improve the scalability of the entire application.

Therefore, all the elements of this chapter will provide the solution. Indeed, the combination of distributed systems techniques and these new software engineering techniques makes it possible to create a social network based on the DEVI approach.

3.1 Interoperability

Since we want to be able to create decentralized social networks, the different instances of a social network must be interoperable. This term means that each of them must have the possibility to connect and communicate with all the others. Also, if we create services in the form of autonomous modules, they should ideally all use the same communication protocol. This way, there is no need for additional workloads to enable the use of new modules. Finally, we want several different social networks to be able to communicate with each other. So, in all three cases, this is a communication problem that needs to be solved.

This question, how to establish an adequate communication standard, directly addresses the interoperability characteristic defined in our objectives. It turns out

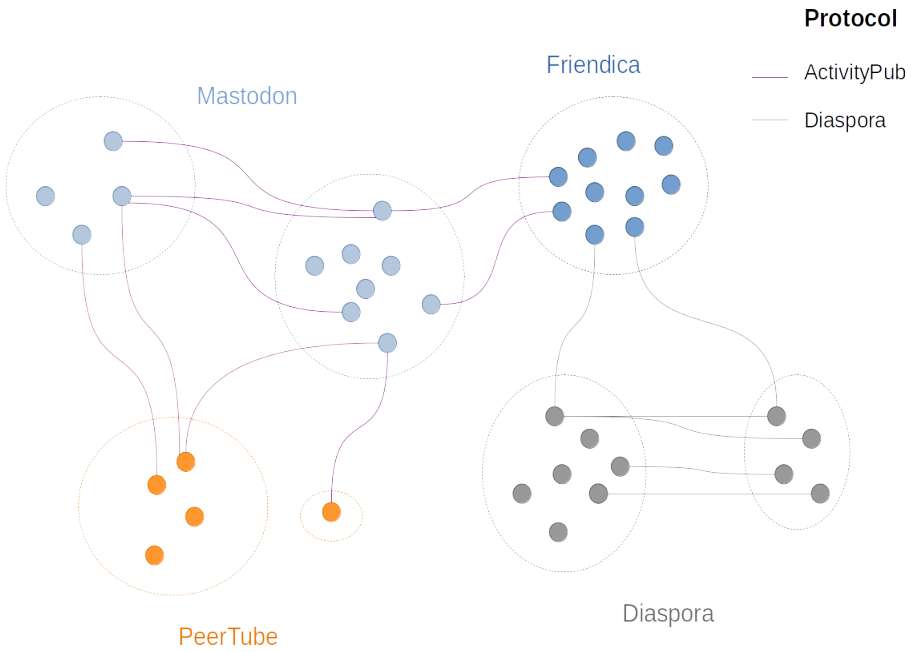


Figure 3.1: Several social networks from Fediverse[1]

that Fediverse offers a solution to the decentralization of social networks. Many are already part of this decentralization and use specific protocols to communicate with each other. Among them, one communication protocol becomes very popular in recent years: ActivityPub.

3.1.1 Fediverse

The name Fediverse comes from the mixture of two words, "Federation" and "Universe", which describe what it is all about. Indeed, Fediverse refers to the federation of a multitude of servers that host free and open software. We can find social networks, microblogging sites, video hosts, etc. The result is an abundant universe where the servers work independently but are interconnected and able to communicate with each other. Social networks created according to the DEVI approach are integrated into this federation.

Figure 3.1 illustrates in a simplified way what makes up the Fediverse. We find Mastodon, PeerTube (a Youtube-like service), Friendica, and Diaspora (the last two are, like Mastodon, horizontal social networks). As we can see, these platforms are decentralized. Indeed, several servers exist and work together to form one social network in general (as a reminder, such servers are called the instances of the

[1]From `https://en.wikipedia.org/wiki/File:Federated_social_media.png`, by Stéphane Guillou under licence CC BY-SA 4.0

social network). For example, two Mastodon members can use different instances. Since these are connected, they will still be able to communicate together. Besides, the other important point to note in the figure is that all these social networks are interconnected. They can communicate with each other and are, therefore, interoperable. For example, a Fiendica user likes a video channel on PeerTube and would like to be notified when a new one is posted. He does not have to use another account on PeerTube: since both of the social networks are connected, the user can follow the video channel from his Friendica's account.

Additionally, we find two different communication protocols in Figure 3.1: ActivityPub and Diaspora. Fediverse actually allows the use of other protocols such as OStatus, Zot, DFRN... Indeed, teams of developers and researchers have tried to develop protocols that can be widely used and meet a maximum of needs. Among them, ActivityPub seems to have taken the lead: in May 2020, nearly 70% of the nodes composing the universe implement this protocol.

Many networks are already present within Fediverse, some are innovating, and others offer alternatives to the social networks we all know. Additionally, activity within this universe seems to be increasingly important in recent years. In 2018, there were only a few tens of thousands of users, but the situation has changed. Today, there are more than 1.3 million active users, with almost 5 million accounts spread over more than 5 thousand instances[2]. Therefore, Fediverse seems to be an emerging and ideal context in which to create a decentralized social network.

3.1.2 ActivityPub

SocialWG, the Social Web Working Group, is a group that is part of W3C[3]. This one stands for World Wide Web Consortium, and it is an international standardization organism that has promoted protocols such as HTML, XML, PNG... Active until 2018, SocialWG was working on the implementation of social web technologies. This working group is behind the ActivityPub[4]communication protocol.

ActivityPub is an open protocol for decentralized social networks. For a few years, it is mostly implemented by the different existing platforms of the Fediverse. This makes the difference between ActivityPub and other protocols. Moreover, in 2018, ActivityPub has become a standard recommended by the W3C. With data sent in a JSON-based format, the protocol allows communication from client to server as well as directly between servers. This is very helpful in the case of a decentralized network where nodes must also be able to exchange information.

[2]Data regularly updated and available from the Fediverse website: `https://fediverse.party/`

[3]`https://www.w3.org/`

[4]`https://activitypub.rocks/`

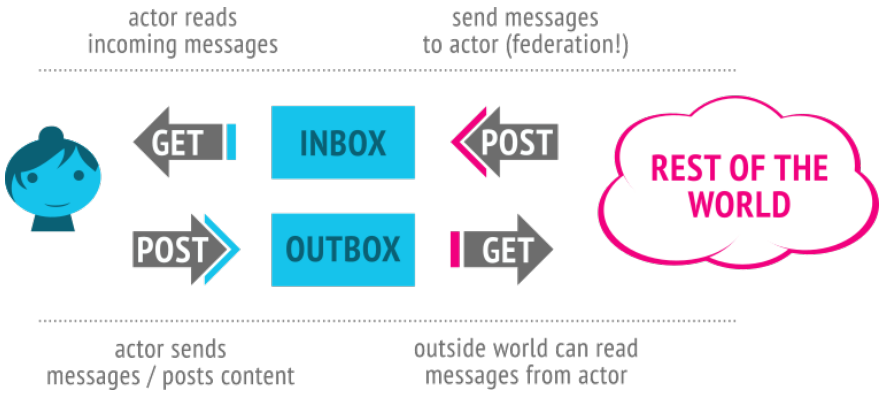


Figure 3.2: Possible interactions with ActivityPub[5]

Communication

Figure 3.2 illustrates all possible interactions between a client and a server, but also directly between servers. We notice that it is possible to perform four different operations:

- A client can send messages to its *outbox*
- A client can retrieve the messages received on his *inbox*
- A server can post messages on an actor's *inbox*
- A server can retrieve messages that an actor has posted. This is done directly from the actor's *outbox* and only with messages for which the server has permission (the simplest example is an actor who has posted public messages).

It is worth noting that within ActivityPub, a message is called an *Activity*, and that several activity types can be used, such as: *Create* to send a new activity, *Follow* so that an actor subscribes to another actor, *Like* so that an actor shows his appreciation towards a specific activity, etc.

ActivityStreams 2.0 Data Format

All exchanged activities respect the ActivityStreams format. This one can facilitate the communication of objects within social web applications. It provides a JSON-based syntax, which makes it both human-readable and parsable by any computer program. More precisely, ActivityStreams is compliant with the JSON-LD standard. This term stands for "JavaScript Object Notation for Linked Data" and is an

[5]From `https://activitypub.rocks/`, by W3C Social Web Working under licence CC BY-SA 4.0

```
{
    "@context": "https://www.w3.org/ns/activitystreams",
    "summary": "Martin created an image",
    "type": "Create",
    "actor": "http://www.test.example/martin",
    "object": "http://example.org/foo.jpg"
}
```

Listing 3.1: Basic object in ActivityStreams format

extension of the JSON format. Basically, it allows developers to structure data and to share these structures across different sites. A field will always be present within a JSON-LD object: `@context`. This one must provide the meaning of each possible key of the JSON-LD object in question.

Listing 3.1 gives the example of an object expressed in ActivityStreams format. As expected, we find the field `@context`, and a link defines it. This link refers to all the ActivityStreams documentation, which explains how such an object is structured and defines all the possible fields that can be used.

In our example, several fields are defined and describe the case of a user who has created an image. If this object is very elementary, it turns out that, in reality, ActivityStreams offers many types ("*Create*" for listing 3.1) and a multitude of fields. Moreover, most ActivityStreams fields can also be expressed by a new ActivityStreams object. The result is a very extensible way to represent any data. In 2017, the W3C recommended this new open format.

Actors

When sending an object, it should be possible to find out who is the author of the object. ActivityStreams has provided a field for this purpose: *actor*. We also find the *actor* field in the given example (listing 3.1), and we can see that it is a URI. This field can actually be completed either by a nested object or by a link to a second object. In both cases, it will have to provide all the information specific to the object's author.

In Listing 3.2, we find the profile of an actor representing a person called Alyssa. The object specifies it is a "Person" because ActivityStreams offers different possible definitions for the type of actor. For example, it can be an application, an organization, a service, etc. The ActivityPub documentation strongly recommends the definition of two fields:

- *inbox*: a reference that allows the user to receive content
- *outbox*: a reference that allows the user to post content

```
{
    "@context": "https://www.w3.org/ns/activitystreams",
    "type": "Person",
    "id": "https://social.example/alyssa/",
    "name": "Alyssa P. Hacker",
    "preferredUsername": "alyssa",
    "summary": "Lisp enthusiast hailing from MIT",
    "inbox": "https://social.example/alyssa/inbox/",
    "outbox": "https://social.example/alyssa/outbox/",
    "followers": "https://social.example/alyssa/followers/",
    "following": "https://social.example/alyssa/following/",
    "liked": "https://social.example/alyssa/liked/"
}
```

Listing 3.2: Example of an Actor in ActivityPub

Although some are recommended, all other fields are not mandatory. For example, the *followers* field provides a collection of actors who follow Alyssa, and the *liked* field lists all the content Alyssa liked.

Therefore, we can notice that this protocol allows interoperability between several social networks. If user A wants to send a message to user B, it is no longer necessary for both to come from the same social network. Indeed, if A and B use social networks that implement ActivityPub, A's server will only have to extract information from B's profile (notably his inbox address) in order to send him a message.

Example

Let's take a concrete case to understand the transmission process of an activity. Suppose Alyssa has lent a book to her friend Ben and wants to know if he has finished it. Alyssa is part of an instance whose domain is *social.example* and Ben *chatty.example*

Alyssa posts on his *outbox* the following Activity: listing 3.3. This is an activity (of type *Create*) that contains an object (of type *Note* and with a simple text as content). Alyssa's *outbox* will then examine the different fields that define the recipients. This way, the server gets a list of each actor's identifier (only Ben in the example), and can request their profile. By using the appropriate field from each recipient's profile, the *outbox* can then retrieve the addresses of all the corresponding inboxes. As a result, Alyssa's *outbox* is now able to post the Activity in Ben's *inbox*. Later, when Ben reads the messages received on his *inbox*, he will see Alyssa's message and will be able to reply to it through a similar process.

```
{
    "@context": "https://www.w3.org/ns/activitystreams",
    "type": "Create",
    "id": "https://social.example/alyssa/posts/a29a6843-9feb-4c74-
        ↪ a7f7-081b9c9201d3",
    "to": ["https://chatty.example/ben/"],
    "actor": "https://social.example/alyssa/",
    "object": {
        "type": "Note",
        "id": "https://social.example/alyssa/posts/49e2d03d-b53a-4c4c
            ↪ -a95c-94a6abf45a19",
        "attributedTo": "https://social.example/alyssa/",
        "to": ["https://chatty.example/ben/"],
        "content": "Say, did you finish reading that book I lent you
            ↪ ?"
    }
}
```

Listing 3.3: Example of a posted activity

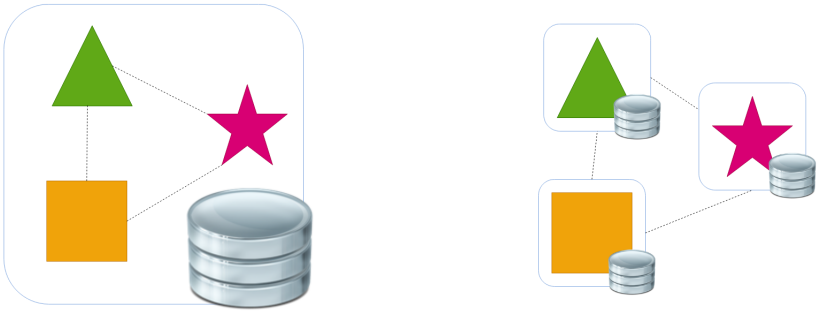
3.2 Modularity, extensibility and scalability

Among all the objectives set, we know that in the DEVI approach, social networks must be vertical. Indeed, we want them to address a specific community and offer adequate services. Furthermore, if two different platforms want to provide the same functionality, we should avoid having to do the same work twice.

By developing these functionalities in a modular way, it becomes possible and straightforward to reuse them. Besides, with the appropriate foundation, a social network becomes extensible: for any new need, the developers only have to find the corresponding module (or create it if it does not yet exist) in order to add it to the platform. We will also see that this division into modules improves the scalability of the application.

Additionally, CQRS and Event-Souring are two patterns that review the traditional approach of the database structure. The former reconsiders and separates the write and read parts of an application. The latter suggests storing data as a sequence of events over time rather than finite states. These two new software engineering principles usually go hand in hand: the combination provides better scaling and more comfortable use of data for programmers.

We will see a concrete example using microservices, CQRS and Event sourcing. It is an application whose architecture is both modular and extensible. It will not only be easy to add functionalities but also to scale only the services that need it.



(a) Monolithic (b) Microservices

Figure 3.3: Types of software architecture
(where shapes represent functionalities)

3.2.1 Microservices

In general, design and development teams tend to create applications according to a monolithic architecture. This term describes a system made up of a single block where the functionalities are developed together. They are interconnected and interdependent. Figure 3.3a illustrates this architecture. We can see a unique block that exchanges with a single database. This combination forms the application in its totality. The architecture has some positive points:

- It has simplicity in development, testing, and deployment. Indeed, each of these steps is done directly with a single block.
- All data are present in the same place, several services using the same data can share them.
- If different functionalities need to communicate, it directly occurs within the same application.

However, the monolithic structure also generates negative aspects:

- Updates quickly become tedious: correcting an error, improving the code, or adding a new feature, all require full knowledge of the system. This requirement can generate a significant additional workload.
- If a problem occurs within a service, the whole application may crash.
- Scaling can only be done with the entire application. This can lead to a waste of resources and money.

With a microservice architecture, the result is immediately more modular and scalable. As shown in Figure 3.3b, services become isolated, independent, and autonomous components. Of course, if developers want to test the application as a whole, this new architecture can make the task more complicated. Also, since each module has its database, we can obtain some cases of data redundancy. However:

- Since a module is smaller, it's easier to take charge of it in order to make changes. Moreover, only this module will have to be redeployed. The rest of the application will not experience any downtime.
- If the application has new needs, it is sufficient to create new modules. There is, therefore, absolutely no need to worry about the code already present.
- It is possible to scale only the services that are heavily used. The scaling of the application thus becomes tailor-made.
- It enables a polyglot architecture. For each microservice, the developers can choose a programming language and a database adapted to the needs.
- It provides some benefits from a management perspective, with smaller teams in charge of smaller amounts of code.

So, given the objectives set and the advantages provided by this new way of structuring a program, we can conclude that microservices are very suitable for the DEVI approach to social networks. Indeed, each functionality can be developed in the form of microservices. Therefore, it is easier for developers to reuse a service, and a heavily used feature can be directly scaled.

3.2.2 Command Query Responsibility Segregation

In many cases, an application requires the use of a database. Therefore, design and development teams need to think about possible models and interactions. The solution is usually built on the traditional CRUD system, i.e., being able to create, read, update, and delete data based on the same models. However, there are often differences in the requirements for read and write data representations. Also, read and write workloads are unusually symmetrical (in an application, we rarely have a one-to-one relation between read and write operations).

There is a pattern that presents a solution to the several negatives aspects of the traditional approach: the Command and Query Responsibility Segregation (CQRS). This new approach proposes to separate the read part from the write one completely. Commands are used to update data in the application. They have their models and are generally based on tasks, actions, specific to the application domain ("buy this item", "vote for this time slot", etc.). The read models, on the

other hand, will propose a view, a state, resulting from the succession of commands received. Only the queries allow us a client read these views. They do not allow to modify data.

Hence, it is possible to take advantage of new positive aspects:

- The separation of write and read can be done by using different modules: one for the write operations, the other for the read ones. As explained with the microservice architecture, it is then possible to scale only the one that needs it.
- The design of the models no longer has to take into account both write and read requirements. Each model can be considered separately and respectively optimized for commands and queries.
- Read models aim to provide a view of the commands received. It becomes possible to design these models in order to summarize data effectively. A model can be designed so that a query does not generate more complex operations within database (joining tables, grouping values, etc.).

Since this pattern offers advantages in terms of performance and modularity, we can consider CQRS to be relevant in meeting the objectives we have set.

3.2.3 Event sourcing

In traditional databases, the application saves models and makes them evolve according to the different requests. Again, this is done according to the CRUD functions. However, if an application is heavily used, it can cause delays or even errors. Indeed, it becomes possible that several users simultaneously attempt to update the same object. This overload can cause performance decreases and potential conflicts.

Event sourcing is a pattern that reviews data storage. This new approach proposes to no longer store the final state of a model. Instead, it stores the sequence of state-changing events that led the application to the current final state. The logs stored (i.e., the events) focus on business entities of the application. Such a database is called an event store. Moreover, such a store can act as a message broker: il allows the services to subscribe to events. This way, subscribers receive all new logs and can process them if necessary.

Event sourcing offers several advantages:

- The application has a history that justifies the final state of the returned objects, which can, for example, be very helpful in the case of an audit.

- It is possible to replay a part of the stored events to find a previous final state.
- With the use of microservices, a recently added module can quickly be active. Since the event store can replay the events, the module can read all the previous logs and process them to propose the final states directly (as if the new service was present from the beginning).
- Together with CQRS, the event store will record the commands, and the read part will consume the different events in order to create final states based on the read model.

The list of the various advantages shows the interest of this pattern if it is coupled with CQRS and microservices. It is, therefore, interesting in our new approach to social network development.

3.2.4 A concrete example

Let's suppose a web application that offers its users the possibility to publish products for sale. It is also possible for users to buy these products. Additionally, a user can subscribe to a type of object so that he will automatically receive all the offers on it by email. Buyers can make payments directly from the site. An email is also sent to a seller when he makes a sale.

This application can be built according to an architecture that uses microservices, CQRS, and Event sourcing. Indeed, we find in figure 3.4 the separation between the write and read part (CQRS). We notice that the commands (write side) are recorded in an Event Store. In this example, the publication of events is done through the Event Bus. On the read side, we find four microservices in charge of consuming the events. Depending on those events, the modules perform actions and create new views. Finally, to interact with the application, the client uses an API, which is actually the gateway of the whole architecture.

When a vendor posts a new product, he sends a command to the application. The information is stored and transmitted in the form of an event via the event-bus. The "offer projection" microservice then takes note of this new product and adds it to its database.

When a buyer searches for a product, he queries the service that lists each item. If he decides to buy it, he sends a command. In this case, the published event is handled by the "payment action" microservice. We notice that this service is also able to send commands. Indeed, after the purchase, this service sends a command,

[6]From `https://www.meetup.com/HumanTalks-Paris/events/256079042/` by Yoann Gotthilf under licence CC BY 3.0

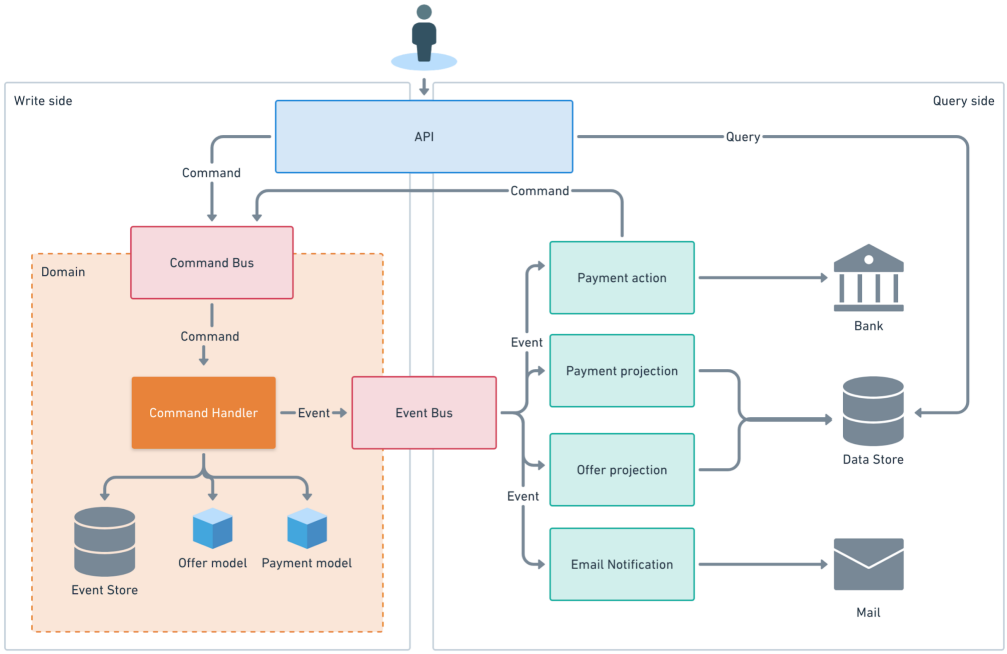


Figure 3.4: Example of an application using microservices, CQRS and Event sourcing[6]

meaning that a purchase has been made. Again, the information is relayed in the form of an event (for example, "product purchased") with all the necessary details. We can then imagine that the microservices "offer projection" and "email notification" consume this event: the former to mark the product as sold or even to delete it from the catalog, the latter to send an email to the seller.

Chapter 4

Proof of concept

We now move on to the second part of this thesis. In the first chapters, we were first able to set our research objectives. Then, we analyzed more precisely the characteristics of the problem to be solved as well as the different objectives that this thesis should achieve. Finally, the previous chapter sets out the elements (such as new software engineering techniques, a communication protocol, etc.) that make it possible to create a new social network according to the DEVI approach.

The second part makes a transition from the DEVI approach creation to its use. Indeed, different software engineering techniques have been presented separately. The development of a proof of concept illustrates how it is possible to articulate them all together. After that, the next chapter will also discuss the different software tools that allow such an implementation.

This chapter sets the context in which a new social network will be created and provides several analyses. It is a platform for youth movements. Since the DEVI approach proposes to create a vertical social network, the first section defines the targeted social community and their different needs in order to identify several services that can be implemented. Then, the section on architecture will illustrate how to articulate all the techniques and the services of the platform. Eventually, the last sections describes the APIs and gives some additional explanations on how the proof of concept works.

4.1 The scout social network case

Many organizations for youth movements are active across Belgium. There is, for example, the French-speaking Scouts Federation of Belgium[1], which counts more than 50,000 members in about 400 units. Figure 4.1 shows the geographical distribution of all these units. Every weekend, these groups gather in their Scout

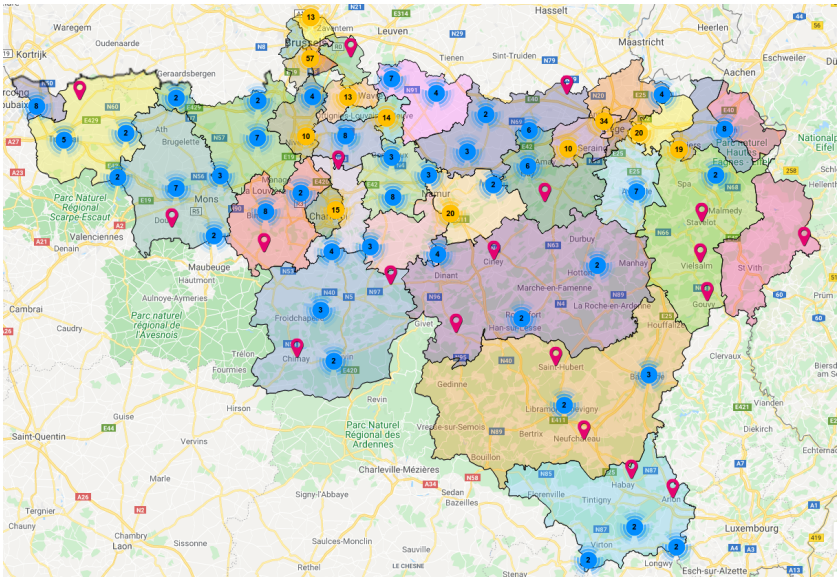


Figure 4.1: Geographical distribution of Scout units in Wallonia (Belgium)[2]

local and take part in activities prepared by their leaders.

Most of these young people are already present on social networks such as Facebook or Instagram. Indeed, they use these platforms in various ways, such as chatting with each other or sharing photos. Many units also use these platforms as a means of communication with parents and children (e.g., to schedule a meeting).

Therefore, designing a social network specifically for scouts may be relevant. The platform could offer functionalities adapted to their needs. Moreover, as shown in Figure 4.1, this community already has a form of decentralization: each member belongs to a unit, all these units are spread throughout Belgium, and the units constitute the Scout Federation. These reasons justify the creation of a social network based on the DEVI approach as a proof of concept.

4.1.1 An implementation based on DEVI approach

Since we are developing a decentralized social network, we need a communication protocol that connects all the instances. Moreover, if this network is integrated into the Fediverse, it is better if the social network can interact with a majority of the other social networks. The appropriate solution is for the platform to implement the ActivityPub protocol.

Also, this web application targets a specific social community: the scouts. We

[1] `https://lesscouts.be/`

[2] `https://lesscouts.be/parents/trouver-une-unite.html`

can then develop a vertical social network that also offers personalized services based on their needs. To do so, we can list and analyze several situations that the application can manage:

- First of all, we know that the scouts gather almost weekly. It can be useful for the leaders to have the number of children present at the next meetings. That way, they can prepare an appropriate animation. Also, the leaders often need to have meetings so that they can discuss the year and the camp.

  In both cases, an agenda is useful. A leader can then send a list of dates to the parents. Parents vote for these dates to confirm or not to their children's attendance. The chief could also use this module to schedule the meetings with his staff more efficiently.

- Chiefs sometimes organize special activities that require parents to travel to a location other than the usual meeting place. Furthermore, a special weekend and a camp occur every year and often require a long round trip for the parents. To avoid wasting time or for ecological reasons, they sometimes organize carpooling. In this case, the parents need everyone's contact details and have to arrange themselves. The social network can offer a service to automate the organization of carpooling trips.

- As said before, the leaders organize a weekend and a camp every year. This requires them to find the best building or meadow to host them. The leaders have to look at several sites to find the offer that suits them best. The developed platform could, therefore, also offer a module to publish places for rent. This service would make it possible to bring all the properties together under the same social network and would help the leaders to contact some owners.

4.1.2 Other similar cases

If the proof of concept of this thesis is limited to a social network for scouts, we can still see similarities with other cases.

We previously mentioned a social network for musicians. They often need to rehearse and therefore have to choose a date that suits each member. The first module could be reused to meet this need.

We also discussed the case of a social network for football clubs. We know the players have to travel almost every week through the region, or even the country, for competitions. The carpooling module could, hence, be reused to facilitate the organization of these trips.

There are groups and forums for people who like to travel. They share all the useful tips, the proper places, etc. A social network for travelers could propose a

module so that owners could offer their property as a waypoint. The third module would then be very suitable for this new use.

In any case, the verticality of a social network dramatically increases its usefulness to these members. We also note that modularity can allow these modules to be directly reused. In the long term, it may become possible to imagine any social network and develop it by using a library containing many modules already implemented.

4.2 Architecture

The challenge lies in the components and techniques it has to integrate. The architecture must include microservices, the implementation of the ActivityPub protocol, the read and write separation as described by CQRS, and an event store. Besides, we don't want clients to be able to use only the primary functions of a social network. They must also be able to benefit from several functionalities: an agenda service, a carpooling service, and a rental service. Also, in the previous chapter, ActivityPub is mainly presented as a communication protocol used by two people who want to communicate. Here, these people must interact with several automated services. Therefore, the second challenge is to adapt the communication between the clients and the modules. For this purpose, ActivityPub can be exploited since it also offers server-to-server communication. It is then possible to automate servers so that they exchange information and allow a service to operate.

Figure 4.2 illustrates the implemented architecture. This architecture contains the three services to help the scouts' community, and each one is composed of several components: two databases, an outbox, an inbox, and a querier. The ActivityPub protocol is implemented thanks to these outboxes and inboxes. The former is used for a user to post a message: it stores the posted message and forwards it to the recipient. The latter is in charge of receiving the messages. Also, each service applies the event sourcing pattern: when a message is received, the inboxes publish some events on the event store, and the queriers subscribe to these events. These queries are then able to consume the events and store some information in their databases. They also use an automatic system, called "secretary", to post messages on users' inbox. This way, a secretary can inform a user about specific news (e.g., for a driver to know that a new user has joined the carpooling).

4.2.1 Use of an existing foundation

The previous thesis (presented in subsection 2.2.2) is used as a foundation. The architecture of this foundation consists of a gateway and a multitude of microservices.

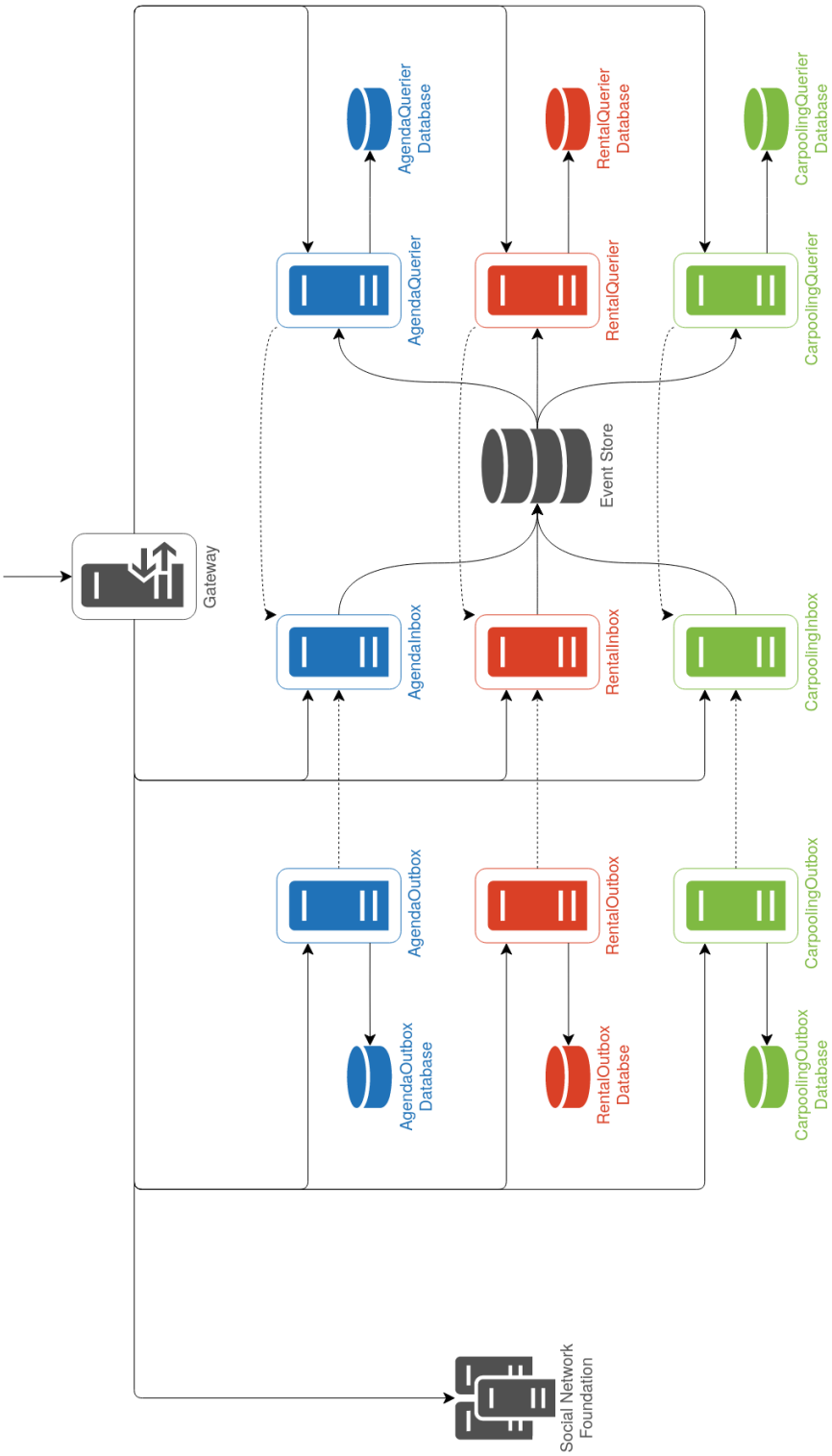


Figure 4.2: Proof of concept architecture

These ones are represented by the "Social Network Foundation" component on figure 4.2. Through the gateway, a client can use all microservices and enjoy every feature of a social network: he can create an account, publish, like, follow, share, or even block content. Since the implementation is extensible, the current thesis is an overlay of this previous work. Indeed, we can implement features and link them to the gateway.

4.2.2 Modules as extensions

Thanks to the gateway and the modularity of the services developed here, extending the social network becomes more convenient. Indeed, these services are autonomous and independent, which makes them easy to reuse: all that is required is to connect them to the gateway and reconfigure this one so that it can transfer the new message types. As a result, the modules become extensions of the initial social network. Also, we can notice that the three services are built according to the same architecture: an outbox, an inbox, a querier, and two databases. Each of these is actually a microservice. As expected, this allows a tailor-made scaling. For example: if the querier of the rental service is heavily requested, it is enough to scale only this one to make the social network adequately adapted to the traffic.

Additionally, the modules are composed of servers that must be able to send and receive data. With the DEVI approach, the solution is that all these servers implement the ActivityPub protocol. This choice standardizes the communication and makes all modules fully interoperable.

Send content

The outboxes allow users to send content. Each one is linked to a database so that the activities posted can be recorded and found later. After that, the server can send a new message to the corresponding inboxes. Each arrow in figure 4.2 represents a possible interaction from one component to another. However, we can notice that some of them are dotted: they mean that communication can happen through two different instances of the Fediverse. For example, an outbox could directly send the activity to the inbox if both sender and receiver use the same social network instance. Otherwise, the outbox has to send the activity to another instance. When such a request arrives, the gateway will handle it and will directly forward the new message to the appropriate inbox. This is also the reason why the gateway is linked to each inbox.

[2]Written by Gary Lovisetto under the supervision of Pr. Etienne Riviere and available in open-source: `https://github.com/glovise15/MasterThesis`

Process and query data

When an inbox receives activities, it generates events that are forwarded to the event store. This one stores them, and since the queriers subscribe to the store, they can consume the events they are interested in. Thanks to its database, each querier will then build aggregated versions, store them, and enable users to send queries to get them.

4.2.3 Application-type actors

One link has not yet been explained in figure 4.2: the one that connects the queriers with the inboxes. The difference between the interactions of the initial social network and the ones with the extensions is the following: the second does not only allow you to send messages, like them, etc. Rather, it makes possible to interact with a service. For this purpose, the module must have a data processing system. For example, a user sends an agenda to several people to schedule a meeting. If the participants send their choices to the organizer, and the organizer has to process the results himself, the service is completely useless. Therefore, the agenda module has to process the organizer's message and interact with the participants.

The solution is to create an intermediary and automated actor (in the sense of ActivityPub). As a result, when this same organizer wants to create an agenda, he will actually send a message to an application-type actor that represents the agenda service. Such an actor is called a secretary in this thesis. The message then generates two steps within the querier: first, the creation of the agenda in database. Secondly, this virtual secretary (i.e., the service), informs the participants of this agenda so that they can vote. After that, they can send their choices to this secretary, and the latter keeps an updated agenda version in its database.

Each of the three services uses this application-type actor system. Besides, as said before, a dotted arrow indicates an interaction between two servers, which are not necessarily from the same social network instance. On figure 4.2, the arrow that links the queries of the inboxes is dotted. Indeed, in our example, the organizer may add participants from different social network instances. Therefore, the secretary may send messages to the inboxes of these other instances.

4.3 Modules API

A module is composed of five microservices (an inbox, an outbox, a querier, and two databases). However, we do not yet know how to use them and how they work in more detail. Hence, it is now interesting to determine which HTTP requests can be sent to each microservices.

Since the three services follow a similar architecture and functioning, this section focuses on only one of them: the Agenda module. However, the second subsection still provides general information about the API of the two other modules.

4.3.1 Agenda API

Thanks to the architecture, we know that a client or server can, through the gateway, send HTTP requests to the outbox, the inbox, and the querier. Table 4.1 shows all the possibilities.

4.3.2 Carpooling and Rental API

The outbox and inbox API of the Carpooling and Rental services are based on the same logic as the Agenda API. It is, therefore, not useful to list each request with all the information. Instead, Table 4.2 summarizes the different possible actions to interact with each of the services. It is then sufficient to use the appropriate URNs: `/carpooling/<action>`, `/carpooling/secretary/<action>`, `/rental/<action>`, `/rental/secretary/<action>`, respectively the outbox and inbox of the carpooling and rental modules. The rest of the specifications do not change.

However, there are differences in the queriers. The Agenda module is indeed different from the Carpooling and Rental modules. In the first case, it is an object shared between defined actors. In the second case, it is public announcements that the actors must be able to find. For this purpose, the queriers propose requests to make searches. For example, an actor can send a query to the Carpooling querier specifying: a place and date of departure, a place and date of arrival. The result will be a list of all matching carpools (with a margin of one day for the departure and arrival dates). The actor will then be able to contact the secretary and perform some actions according to Table 4.2 (for example, he will ask the secretary to add him to the passengers list).

4.4 Interaction through secretary

Subsection 4.2.3 introduced the use of application-type actors. This automated secretary system receives all messages, processes them, and sends new ones. This section provides a more detailed explanation of how it works. Again in order to avoid redundancy, this section only focuses on the Agenda service since the three services follow a similar architecture and functioning.

The querier allows clients to send a request to obtain the secretary's profile. List 4.1 shows the result of this HTTP request. As expected, this is an application-

Table 4.1: Agenda Module API

OUTBOX			
Method	**URN**	**Parameters**	**Description**
POST	`/agenda/create`	ActivityStreams object	Create an agenda
POST	`/agenda/close`	ActivityStreams object	Set the chosen date and close the agenda
POST	`/agenda/open`	ActivityStreams object	Reopen an agenda and cancel the chosen date
POST	`/agenda/vote`	ActivityStreams object	Vote for some dates
POST	`/agenda/withdraw`	ActivityStreams object	Cancel a vote
POST	`/agenda/reset`	ActivityStreams object	Remove all votes
GET	`/agenda/:id`	id, a message identifier	Get an ActivityPub message sent from this outbox
INBOX			
Method	**URN**	**Parameters**	**Description**
POST	`/agenda/secretary/create`	ActivityStreams object	Ask for the secretary to create an agenda
POST	`/agenda/secretary/close`	ActivityStreams object	Ask for the secretary to close an agenda
POST	`/agenda/secretary/open`	ActivityStreams object	Ask for the secretary to open an agenda
POST	`/agenda/secretary/vote`	ActivityStreams object	Send a vote to the secretary
POST	`/agenda/secretary/withdraw`	ActivityStreams object	Ask for the secretary to cancel a vote
POST	`/agenda/secretary/reset`	ActivityStreams object	Ask for the secretary to cancel all votes
POST	`/agenda/news`	ActivityStreams object	For the secretary to post messages about a specific agenda
QUERIER			
Method	**URN**	**Parameters**	**Description**
GET	`/agenda/with/:id`	id, the actor identifier	Get agendas with actor as organizer or participant
GET	`/agenda/content/:id`	id, the agenda identifier	Get the content of a specific agenda
GET	`/agenda/secretary`	-	Get the secretary's profile
GET	`/agenda/message/:id`	id, the message identifier	Get an ActivityPub message sent by the secretary
GET	`/agenda/news/new/:id`	id, the actor identifier	Get new messages received by an actor
GET	`/agenda/news/old/:id`	id, the actor identifier	Get old messages received by an actor

Table 4.2: List of interactions with the Carpooling and Rental modules

CARPOOLING		RENTAL	
Action	**Description**	**Action**	**Description**
create	Create a carpooling	*create*	Create a property
join	For an actor to join a carpooling	*delete*	Delete a property
leave	For an actor to leave a joined carpooling	*update*	Update some property's fields
		book	Book a property
manage	For the driver to accept or reject the joined actors	*cancel*	Cancel a booking
		accept	For the owner to accept a booking
close	For the driver to mark the carpooling as unavailable	*reject*	For the owner to refuse a booking
		comment	For a tenant to comment the property
open	For the driver to mark the carpooling as available	*close*	For the owner to mark the property as unavailable
		open	For the owner to mark the property as available

```
{
    "@context": "http://www.w3.org/ns/activitystreams",
    "id": "http://172.18.0.1:3205/agenda/secretary",
    "type": "Application",
    "name": "Agenda module secretary",
    "summary": "In charge of processing all messages concerning the
       ↪ agenda module (domain http://172.18.0.1)",
    "inbox": "http://172.18.0.1:3204/agenda/secretary"
}
```

Listing 4.1: Profile of the agenda secretary

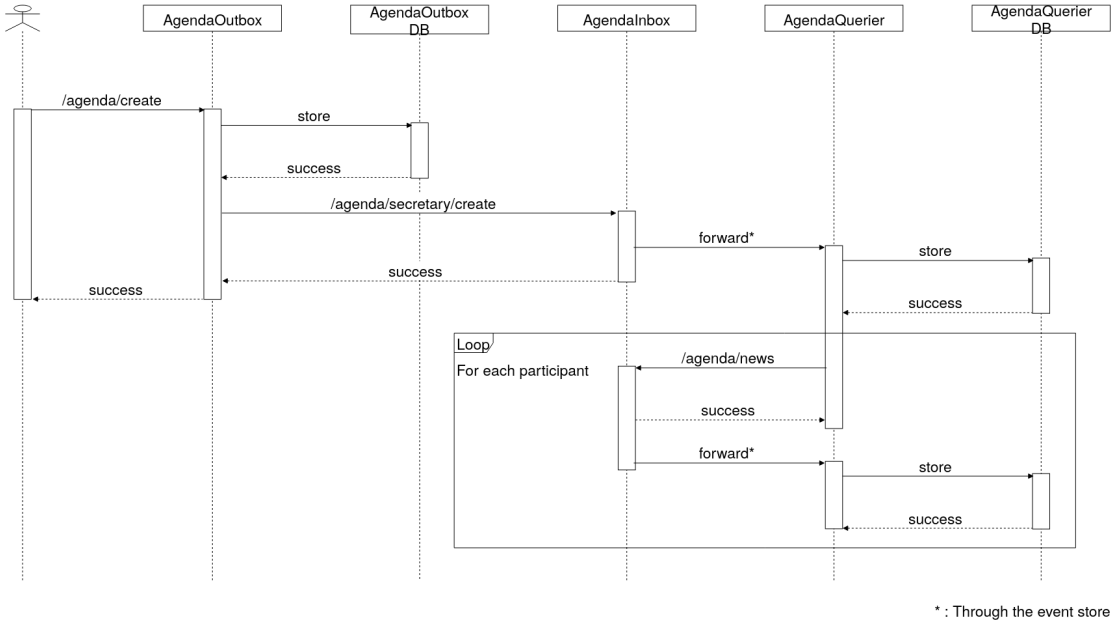


Figure 4.3: Sequence diagram of an agenda creation

type actor. This actor is the secretary and actually refers to all the automatic processing that occurs within the service.

Users of the agenda module never interact directly with each other. All communications must go through this secretary. This is the reason why, in the API, almost all paths in the inbox are dedicated to this application-type actor. The secretary is also in charge of keeping each user up to date. In the inbox, an API address does not concern the secretary: `/agenda/news`. It is used to post news about an agenda to one or more participants. Indeed, when data are processed in the querier, information ("agenda created", "agenda reset", etc.) is communicated to the participants involved. Only the secretary has the right to post news to this address.

Example Let's take the case of a user who creates an agenda. The sequence diagram in figure 4.3 shows each of the steps. The first one is, of course, the sending of the dates and participants by the organizer. As said just before, this message is intended for the secretary. The outbox adds an identifier to the message (so that it can be retrieved later), saves it in database and finally sends it to the secretary's inbox.

Once the message is received, the inbox generates an event through the event store so that the appropriate querier can handle it: this querier saves a new agenda in its database. It then sends the identifier of this new object to the participants.

For this purpose, the querier ("the secretary") posts a message on each participant's inbox. This message contains a flag to indicate that an agenda has been created and the identifier of this agenda.

After that, the inboxes receive the message and generate an event. The appropriate queriers consume this event and record the existence of the new message. As listed in the querier API, it will then be possible for participants to send an HTTP request to get the new notifications. They will know that an agenda has been created and will have its identifier so that they can find it. Eventually, they will have the opportunity to send a message to the secretary to vote.

4.4.1 Wrapped JSON object as Note content

Each time a client or server posts content, it sends an HTTP request of type POST. The API indicates that the body of this request always contains an ActivityStreams object. As explained earlier, this object is called an activity and provides a nested object. Examples: a "Like" activity whose content is a link to a publication, a "Create" activity whose content is a Note object, etc. This structure is used to respect the conditions given by the ActivityPub protocol.

However, a user always sends messages to and receives messages from the secretary (the application). Since this one is an automated system, the content of the messages must be written in a machine-processable format. For this reason, all messages are notes that wrap a JSON object.

For example, Listing 4.2 is a Create-type activity sent to the Agenda module's secretary. The activity's type indicates to the receiver that it is a new post. It contains a nested object, which is a Note-type one with content in JSON format. This content lists some dates, the participants, a name, and a description. Since the activity is posted on inbox URN `/agenda/secretary/create`, the secretary has to interpret the note content in order to save a new agenda and inform each participant. The participants' inboxes will then receive, from the secretary, a new Note-type object whose content is also expressed in JSON format. Listing 4.3 gives an example of such content: we find the flag to indicate that an agenda has been created, the agenda's URL, and the actor's identifier who generated this action.

```
{
    "@context": "https://www.w3.org/ns/activitystreams",
    "type": "Create",
    "to": "http://172.18.0.1:3205/agenda/secretary",
        ⋮
    "object": {
        "@context": "https://www.w3.org/ns/activitystreams",
        "type": "Note",
            ⋮
        "content" : {
            "name": "Meeting INGI",
            "description": "Important meeting about sinf11ba'\''s
                ↪ program (+- 1h)",
            "dates": [
                "2020-03-15T14:00:00.000Z",
                "2020-03-16T14:00:00.000Z",
                "2020-03-17T14:00:00.000Z",
                "2020-03-18T14:00:00.000Z"
            ],
            "with": [
                "http://172.18.0.1:3106/actor/get/user02-45df",
                "http://172.18.0.1:3106/actor/get/user03-45df",
                "http://172.18.0.1:3106/actor/get/user04-45df"
            ]
        }
    }
}
```

Listing 4.2: Example of a Create-type activity sent to the Agenda's secretary

```
{
    "url": 'http://172.18.0.1:3205/agenda/content/168385c0-a354-11ea
        ↪ -88d6-793a532d9716',
    "from": 'http://172.18.0.1:3106/actor/get/user01-45df',
    "type": 'created'
}
```

Listing 4.3: Content of a Note-type object sent by the Agenda's secretary

Chapter 5

Implementation & Evaluation

In the previous chapter, we saw how to arrange several new software engineering techniques to create a proof of concept. We described the architecture, the possible interactions, etc. Therefore, the next step is the implementation of the three services. For this purpose, different choices are possible.

Each microservice is developed using the Javascript programming language. This one allows, for instance, to easily manipulate objects in ActivityStreams format. Moreover, these microservices can run independently within containers. The Docker tool makes it easy to create them and offers easy scaling.

The DEVI approach uses two different types of storage. The outbox stores the messages it transfers. The querier consumes events and, then, also stores information. Both components use MongoDB databases. These are NoSQL document-oriented databases. Regarding the event store, the implementation is based on an open-source tool called Eventstore. This store records the events and allows microservices to subscribe to these events.

Finally, some tools also allow the implementation to be tested. The combination of Mocha and Chai makes it possible to create several scenarios in order to verify all interactions with the different modules.

All the tools listed above are choices made for this implementation. Therefore, this chapter aims to explain in detail why they have been chosen, how they work, their advantages, etc.

5.1 JavaScript code

As said before, the ActivityStreams format extends the JSON format. This one stands for "JavaScript Object Notation". Indeed, the JavaScript language allows us to manipulate this kind of object natively. Hence, it seems quite logical to use this language to develop the proof of concept. In other programming languages, it would

be necessary to use appropriate and not always optimized libraries. Moreover, this language is very well known in the developer world and continues to be regularly updated. If a developer wants to understand how a module works, we can consider that he will quickly be able to take it in hand and make modifications as he wishes.

5.1.1 Node.js

JavaScript is considered to be a client language and is not suitable for implementing back-end modules. So we can do this, we need to use Node.js[1]. This is an environment that enables the use of JavaScript on the server-side. We then keep the advantages of this language while implementing each of the modules. Moreover, there is even a vast library of tools that can be imported within an implementation. This avoids having to recreate code for existing features. Instead, we can focus on the more essential parts of the project. Besides, Node.js is a trendy environment. A team of developers will probably be able to easily reuse a module, understand it, and modify it.

5.1.2 Express.js

Since services are composed of servers, it is necessary to handle all HTTP requests exchanged. In an environment based on Node.js, Express.js[2]is a framework that will facilitate this management. Express.js allows us to create a complete API easily : it offers a whole range of features such as creating the routes of a server, manipulating the parameters and header of HTTP requests, providing responses in different formats... Besides, Express.js is a common framework. Therefore, most developers are directly able to understand how the server API is built.

5.2 Containers

The objectives set out in Chapter 2 emphasize the modularity and scalability of the application. To achieve this, the architecture is split into a multitude of microservices. So, a module is built thanks to the associations of several microservices. These are programs that run independently. It is, therefore, possible for each of them to run within a container.

As shown in Figure 5.1, a container is a virtualization at the operating system level that creates an isolated execution space. This container actually generates an environment with the source files, libraries, tools, etc. required for a given program to run in it. This system offers several advantages. First of all, a container provides

[1] `https://nodejs.org/en/`
[2] `http://expressjs.com/`

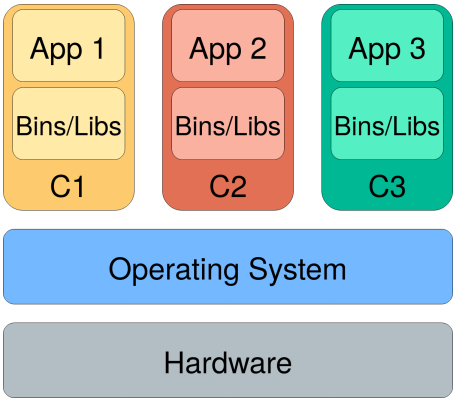


Figure 5.1: Containerized applications

great portability: since it contains everything needed to run, we can quickly redeploy it to another machine, or even to another operating system. Second, when an application is under heavy load, scaling consists merely of duplicating the container and distributing the workload.

5.2.1 Docker

The open-source Docker[3]project facilitates the creation and management of containers. In Figure 5.1, it would appear as a layer between the operating system and all containers. Docker actually enables the creation of lightweight containers and their automated deployment. So, this tool makes it both faster and easier to run them or move them to another host.

In order for Docker to create a container, it needs a configuration written in a specific file: the `Dockerfile`. This file enables, for example, the import of an already existing file system, the import of application source code, the definition of environment variables, etc. Also, we know that a container is an isolated space. Therefore, if we want to send network requests to the application, it is necessary to open ports. It is also in the `Dockerfile` that such parameters are defined. Thanks to this configuration file, Docker is then able to build an image, i.e., a file system containing the application and all the necessary dependencies. Later, this image will be used to generate the container.

Finally, Docker provides a vast library[4]of images. Among these, we can find *mongo*, *node*, *mysql*, *openjdk*, etc. It is even possible for a user to upload his images for reuse. We are, therefore, able to put our modules online so that other social networks can use them.

[3]`https://www.docker.com/`
[4]`https://hub.docker.com/`

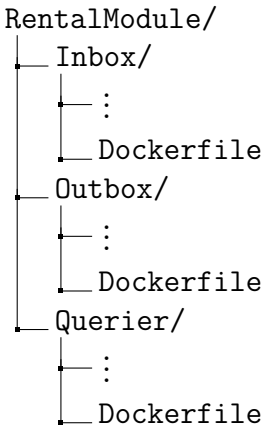


Figure 5.2: Rental module file tree

Docker Compose

Although Docker makes it easy to create a container, the work remains tedious if the application has a lot of them. In the architecture of the social network for scouts, a single service consists of five microservices, i.e., five containers (considering that the event store is already present): an outbox, its database, an inbox, a querier and its database.

One solution allows us to automate the creation of several containers in a single command: Docker Compose[5]. This tool needs a simple file in YML format that lists all the required containers, with a few associated parameters. For example: for each container, it is necessary to specify the image to download from the library or the path to the *Dockerfile*. Then, in one command, Docker Compose will build the images, create the containers, and execute them.

Furthermore, it becomes fast and easy to connect a module to its social network. For example: suppose a team of developers wants to add the Rental module to its social network using the source code and not the images. Figure 5.2 shows part of the file tree of this module. The task becomes very straightforward: they need to download the Rental service (i.e., the `RentalModule` folder in the tree structure), add in the YML file the path to each *Dockerfile* to generate the images, and set the few necessary environment variables. After that, one command will build the images, create the new containers, and execute them. As a result, the social network will provide the Rental functionality.

So, this technique emphasizes the modularity of the services: adding them becomes a quick and easy task.

[5] `https://docs.docker.com/compose/`

5.3 Event store

The open-source Eventstore[6]database is used to apply the event sourcing pattern. This event store provides the ability to list events in different streams. This feature is useful since it allows each microservice to publish or subscribe only to the streams it is interested in.

The store can run on most platforms and offers a basic HTTP API. Also, the official website publishes clients that the community has developed. Therefore, developers who are creating a new module can easily find the appropriate client and interact directly with the event store (instead of having to implement each interaction via HTTP requests). These clients are implemented in several languages (Java, Ruby, Python, Node.js, etc.), which makes the event store compatible with most projects. Eventstore also allows the configuration of projections, i.e., a sequence of instructions that occur when an event is added. This can, for example, be used to create event statistics directly from the store.

As a result, the social network has a history of all the events that have taken place. This history justifies the current state of the platform. Additionally, developers can, for example, put the platform back into a state that took place in the past. It is also possible for a new microservice to replay a stream from the Eventstore. This way, it can process all old events and be entirely up to date.

5.4 NoSQL Databases

Many databases are relational. All developers have the knowledge to use them. Additionally, such databases have very comprehensive interfaces. SQL, the "Structured Query Language", is a standard and well-known language to manipulate them.

However, relational databases can also cause problems. Indeed, they do not scale well for large data sizes. They also do not have good elasticity, i.e., they do not automatically scale in real-time and according to needs. Besides, these databases do not support well the recording and querying of data that are not strictly structured.

With the DEVI approach, the databases must have proper scalability. Also, we have seen several examples of messages sent using the ActivityPub protocol: the content of these messages varies widely. Therefore, it is not possible to anticipate a well-defined data structure. The solution is the use of NoSQL ("Not only SQL") databases. These offer both better scalability and elasticity. Moreover, they are very well suited to store and query data with varying structures.

[6]`https://eventstore.com/`

```
{
    "_id": String,
    "name": {type: String, required: true},
    "description": {type: String, default: ""},
    "province": {type: String, required: true},
    "city": {type: String, required: true},
    "capacity": {type: Number, required: true, min: 1},
    "price": {type: Number, min: 0, required: true},
    "owner": {type: String, required: true},
    "showers": {type: Number, required: true, min: 0},
    "meadow": {type: Boolean, required: true},
        ⋮
}
```

Listing 5.1: (Rental Module) Extract of the property schema

5.4.1 MongoDB

The queriers and outboxes of each service are MongoDB[7]databases. It is a document-oriented storage system. More precisely, it is a key-value store whose documents are in JSON format. With MongoDB, the documents of the same collection can have different structures. Moreover, this store allows us to nest them. Since ActivityStreams is a JSON-based format and is very extensible, this database is very well suited.

MongoDB also offers other advantages. For example, it has interfaces in many languages. Therefore, if they want to implement a new module, developers are not restricted in the choice of programming language. The store also provides the ability to index documents in order to improve the performance of some queries. Finally, MongoDB offers proper scaling, even for large data sizes. Regarding the elasticity, there is the MongoDB Atlas[8]tool that allows the automatic scaling. All these reasons show that MongoDB meets our criteria and is well suited as a database for our modules.

Mongoose

Of course, most of data exchanged are in ActivityStreams format and do not follow a specific structure. Nevertheless, queriers consume events from the event store to build finite states. These states always have the same structure and can, therefore, be based on a fixed model.

Let's take the example of the Rental module. This module lists different

[7]https://www.mongodb.com/

[8]https://www.mongodb.com/cloud/atlas

properties so that interested users can rent them. In this case, the querier requires specific information to register a new property. Therefore, since the structure will always be the same, it may be relevant to model it. Mongoose[9]is precisely a library of MongoDB object modeling for Node.js. It is an overlay that abstracts part of the MongoDB code and simplifies some uses. Moreover, it allows programmers to create schemas easily.

The listing 5.1 shows a part of what is a property in the querier database. As shown, Mongoose enables the definition of different fields and validations (value type, mandatory value, minimum value, etc.). Moreover, when a validation is not respected, the method returns an error message with the necessary details. In this case, the querier is then able to send a message to the appropriate actor. This message contains a flag indicating that it is an error and a description of this one. Mongoose is thus very useful thanks to its API and helps to ensure the respect of precise data structures.

5.5 Testing

It is mandatory to test the implementation to ensure that it works properly. For this purpose, it is possible to simply send requests to the servers and check the answers obtained. For example, Insomnia[10]can be a useful tool: it has a graphical user interface and allows a user to generate all types of HTTP requests quickly. However, it can take time to test complete scenarios. Besides, if the response is relatively large, analyzing each field becomes tedious. Therefore, it makes more sense to automate all tests.

5.5.1 Mocha and Chai

Mocha[11] is a JavaScript test framework, and it can run on Node.js. It allows the implementation of asynchronous tests. This is very convenient since, in the case of a web application, the tests must interact with servers and wait for answers. Also, Mocha allows you to decompose tests into a suite of cases, define instructions to be performed before or after the tests, and returns a full report of the errors detected.

Chai[12]is an assertion library that can be paired with any JavaScript test framework. It provides a rich library of assertions that are based on natural language. Therefore, the developer can easily write or read tests based on Chai. Furthermore, Chai has a library of plugins for testing specific objects. For instance,

[9]`https://mongoosejs.com/`
[10]`https://insomnia.rest/`
[11]`https://mochajs.org/`

```
describe("secretary's message announces a new agenda", function () {
            ⋮
        it("should correspond to data sent by actor[0]", function (
            ↪ done) {
            agenda.should.have.property("agendaID");
            agenda.should.have.property("name", newAgenda.name);
            agenda.should.have.property("participants");
            agenda.participants.should.be.an("array").that.have.
                ↪ lengthOf(4);
                ⋮
        });
});
```

Listing 5.2: Example of a test using Mocha and Chai

Chai HTTP provides features to verify the content of HTTP responses.

The combination of Mocha and Chai thus makes it possible to implement tests sequences for each service. Listing 5.2 shows an example. Thanks to Mocha, the use of keywords such as `describe` and `it` makes the sequences very clear. The body of the test uses Chai's assertions and is also very easy to understand.

Each module is tested on several points: format validations (for example, creating a carpool with a negative price cannot work), permissions (for example, a user cannot modify a property he does not own), or entire scenarios.

Example A test checks a scenario in the Agenda module: an actor asks the secretary to create a new agenda. Tests verify that the organizer and participants have all received a message. Then, tests also check that the agenda link works, and that the format and content of the agenda are correct. After that, the participants vote one or more times, some cancel their votes, and finally, the organizer closes the agenda. For each step, the tests check HTTP responses, messages sent to actors, and new versions of the agenda.

[12] `https://www.chaijs.com/`

Chapter 6

Conclusion

The DEVI approach provides a solution for developing open and alternative social networks. First, the new social networks implemented are vertical. This means that each social network targets a specific social community in order to offer appropriate functionalities. Thanks to the microservice architecture, each service is developed separately, so it is easy to export them for reuse. This isolation also makes it easy to import new services. Finally, this architecture allows the scaling to be tailor-made.

The applications created are also decentralized: several instances of a social network can exist. Moreover, they are part of a federation of social networks called the Fediverse. It is, therefore, possible for users from different platforms to communicate with each other. The use of the ActivityPub protocol as a standard makes all instances directly interoperable.

The combination of CQRS and event sourcing patterns is successful. First of all, it allows the developers to adapt the read and write requirements of the application. Within a social network, the number of commands is rarely equal to the number of requests, so scaling can also be tailored. Then, the application has a history of all the events that occurred. This can be used to retrieve a state in the past. It also allows a new microservice to reread all the events so that it is directly up to date.

Thanks to all these new software engineering techniques, the implementation of a proof of concept has become possible. Indeed, by using tools such as Node.js, MongoDB, or even Docker, it was possible to develop a social network for the French-speaking Scouts Federation of Belgium. The vertical platform offers three different services to its users, and these services can be easily reused: an Agenda service, a Carpooling service, and a Rental service. Implemented according to the DEVI approach, this new application respects the objectives set: a Decentralized, Extensible, Vertical, and Interoperable social network.

6.1 Future work

Since the implementation of the social network for scouts is done on the servers side, it is now possible to develop an entire user interface. This new step would concretize not only this work but also the previous thesis that provided the foundation. Moreover, it would probably allow the UI developers to propose improvements for the services APIs.

In the proof of concept, each service's architecture is the same: an outbox and its database, an inbox, a querier and its database. Besides, the outbox and inbox APIs of each service also have similarities. Therefore, it would be interesting to know if a framework can generalize the design of a service. This framework could then generate the whole structure common to any module so that developers can focus on the essential parts of the service.

With the DEVI approach, the development of services is done within a module. It is then possible for another social network to import an existing service in order to incorporate it into the application. Also, Docker provides a way to export images of the microservices on a platform so that they can be downloaded. A library offering a multitude of modules developed using the DEVI approach is, therefore, conceivable. If a team of developers wants to create a new social network, they could then use this library to quickly and easily add features.

References

Dominic Betts, Julian Dominguez, Grigori Melnik, Fernando Simonazzi, and Mani Subramanian. *Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and maintainability with Windows Azure*. Microsoft patterns & practices, 2013.

Erin Shepherd Amy Guy Evan Prodromou Christopher Lemmer Webber, Jessica Tallon. Activitypub. Technical report, W3C, 2018. URL `https://www.w3.org/TR/activitypub/`.

Andrzej Debski, Bartlomiej Szczepanik, Maciej Malawski, Stefan Spahr, and Dirk Muthig. In search for a scalable & reactive architecture of a cloud application: Cqrs and event sourcing case study. *IEEE Software*, (99), 2017.

Martin Fowler. Event sourcing, Dec 2005. URL `https://martinfowler.com/eaaDev/EventSourcing.html`.

Martin Fowler. CQRS, Jul 2011. URL `https://martinfowler.com/bliki/CQRS.html`.

Evan Prodromou James Snell. Activitystreams 2.0. Technical report, W3C, 2017. URL `https://www.w3.org/TR/activitystreams-core/`.

Sam Newman. *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.

Florian Rademacher, Jonas Sorgalla, and Sabine Sachweh. Challenges of domain-driven microservice design: a model-driven perspective. *IEEE Software*, 35(3), 2018.

Chris Richardson and Floyd Smith. Microservices: from design to deployment. *Nginx Inc*, 2016.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**